EMPIRICAL APPROACH TO THE COMPLEXITY OF HARD PROBLEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Eugene Nudelman

October 2005

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Yoav Shoham Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Andrew Ng

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Bart Selman
(Computer Science Department, Cornell University)

Approved for the University Committee on Graduate Studies.

To my parents and grandparents

Abstract

Traditionally, computer scientists have considered computational problems and algorithms as artificial formal objects that can be studied theoretically. In this work we propose a different view of algorithms as natural phenomena that can be studied using empirical methods. In the first part, we propose a methodology for using machine learning techniques to create accurate statistical models of running times of a given algorithm on particular problem instances. Rather than focus on the traditional aggregate notions of hardness, such as worst-case or average-case complexity, these models provide a much more comprehensive picture of algorithms' performance. We demonstrate that such models can indeed be constructed for two notoriously hard domains: winner determination problem for combinatorial auctions and satisfiability of Boolean formulae. In both cases the models can be analyzed to shed light on the characteristics of these problems that make them hard. We also demonstrate two concrete applications of empirical hardness models. First, these models can be used to construct efficient algorithm portfolios that select correct algorithm on a per-instance basis. Second, the models can be used to induce harder benchmarks.

In the second part of this work we take a more traditional view of an algorithm as a tool for studying the underlying problem. We consider a very challenging problem of finding a sample Nash equilibrium (NE) of a normal-form game. For this domain, we first present a novel benchmark suite that is more representative of the problem than traditionally-used random games. We also present a very simple search algorithm for finding NEs. The simplicity of that algorithm allows us to draw interesting conclusions about the underlying nature of the problem based on its empirical performance. In particular, we conclude that most structured games of interest have either pure-strategy equilibria or equilibria with very small supports.

Acknowledgements

None of the work presented in this thesis would have been possible without many people who have continuously supported, guided, and influenced me in more ways than I can think of.

Above all I am grateful to Yoav Shoham, my advisor. Yoav gave me something invaluable for a graduate student: an enormous amount of freedom to choose what I want to do and how do I want to do it. I felt his full support even when my research clearly took me to whole new fields, quite different from what I thought I would do working with Yoav. Freedom by itself can be dangerous. I was also fortunate to have strict expectations of progress to make sure that I move along in whatever direction I chose. I felt firm guidance whenever I needed it, and could always tap Yoav for solid advice. He never ceased to amaze me with his ability to very quickly get to the heart of any problem that was thrown at him, immediately identify weakest points, and generate countless possible extensions. I felt that Yoav would be a good match for me as an advisor when I aligned with him during my first quarter at Stanford; after five years this conviction is stronger than ever.

It is impossible to overstate the influence of my good friend, co-author, and officemate Kevin Leyton-Brown. I would be tempted to call him my co-advisor if I did not, in the course of our relationship, witness his transformation from a long-haired second-year Ph.D. student striving to understand the basics of AI for his qual to a successful and respected professor, an expert in everything he works on. A vast portion of the work presented in this thesis was born out of endless heated arguments between Kevin and myself; arguments that took place over beers, on ski runs, in hotels, and, of course, during many a late night in our office — first in person, and, later, over the

phone. These could be long and frustrating — sometimes due to our stubbornness and sometimes because initially we did not really understand what we were talking about; all were very fruitful in the end. Kevin taught me a great deal about research, presentation of ideas, the workings of the academic world, attention to minute details such as colors and fonts, as well as ways to fix those, the list goes on. Nevertheless, it is our endless late-night debates from which you could see Understanding being born that I'll miss the most.

The work culminating in this thesis started when Yoav sent Kevin and myself to Cornell, where we met with Carla Gomes, Bart Selman, Henry Kautz, Felip Mañà, and Ioannis Vetsikas. There Carla and Bart told us about phase transitions and heavy-tailed phenomena, and Kevin talked about combinatorial auctions. I learned about both. This trip inspired us to try to figure out a way to get similar results for the winner determination problem, even after it became quite clear that existing approaches were infeasible. I am very grateful to Yoav for sending me on this trip when it wasn't at all obvious what I would learn, and it was quite probable that I wouldn't contribute much. That trip defined my whole research path.

I'd like to express special thank you to Carla Gomes and Bart Selman, who have been very supportive over these years. They followed our work with interest ever since the first visit to Cornell, always happy to provide invaluable advice and to teach us about all the things we didn't understand.

Needless to say, a lot of work contained here has been published in various forms and places. I was lucky to have a great number of co-authors who contributed to these publications. Chapters 2 and 3 are based mostly on ideas developed with Kevin Leyton-Brown. They are based on material that previously appeared in [Leyton-Brown et al. 2002; Leyton-Brown et al. 2003b; Leyton-Brown et al. 2003a] with some ideas taken from [Nudelman et al. 2004a]. Galen Andrew and Jim McFadden contributed greatly to [Leyton-Brown et al. 2003b] and [Leyton-Brown et al. 2003a]. Rámon Béjar provided original code for calculating the clustering coefficient.

Chapter 4 is based on [Nudelman et al. 2004a], joint work with Kevin Leyton-Brown, Alex Devkar, Holger Hoos, and Yoav Shoham. I'd like to acknowledge very helpful assistance from Nando de Freitas, and our indebtedness to the authors of

the algorithms in the `SATzilla` portfolio. This work also benefited from helpful comments by anonymous reviewers.

Chapter 6 is based on [Nudelman et al. 2004b], which is joint work with Kevin Leyton-Brown, Jenn Wortman, and Yoav Shoham. I'd especially like to acknowledge Jenn's contribution to this project. She single-handedly filtered vast amounts of literature, distilling only the things that were worth looking at. She is also responsible for a major fraction of GAMUT's code. I'd also like to thank Bob Wilson for identifying many useful references and for sharing his insights into the space of games, and Rob Powers for providing us with implementations of multiagent learning algorithms.

Finally, Chapter 7 is based on [Porter et al. to appear][1], joint work with Ryan Porter and Yoav Shoham. I'd like to particularly thank Ryan, who, besides being a co-author, was also an officemate and a friend. From Ryan I learned a lot about American way of thinking; he was also my only source for baseball and football news. After a couple of years, he was relocated to a different office. Even though it was only next door, in practice that meant many fewer non-lunchtime conversations — something that I still occasionally miss. Ryan undertook the bulk of implementation work for this project while I was tied up with GAMUT, which was integral to our experimental evaluation. Even when we weren't working on a project together, Ryan was always there ready to bounce ideas back and forth. He has had definite influence on all work presented in this thesis. Returning to Chapter 7, I'd like to once again thank Bob Wilson, and Christian Shelton for many useful comments and discussions.

One definite advantage of being at Stanford was constant interaction with a lot of very strong people. I'd like to thank all past and present members and visitors of Yoav's Multiagent group; all of my work benefited from your discussions, comments, and suggestions. Partly due to spacial proximity, and, partly, to aligned interests, I also constantly interacted with members of DAGS—Daphne Koller's research group. They were always an invaluable resource whenever I needed to learn something on pretty much any topic in AI. I'd also like to mention Bob McGrew, Qi Sun, and Sam Ieong (another valued officemate), my co-authors on [Ieong et al. 2005], which is not part of this thesis. It was very refreshing to be involved in something so

[1] A slightly shorter version has been published as [Porter et al. 2004].

non-experimental.

I was very lucky to count two other members in the department, Michael Brudno and Daniel Faria, among my close friends. Together, we were able to navigate through the CS program, and celebrate all milestones. They were always there whenever I needed to bounce new ideas off somebody. They also exposed me to a lot of interesting research in areas quite distant from AI: computational biology and wireless networking. More importantly, sometimes they allowed me to forget about work.

I would also like to thank members of my Ph.D. committees, without whom neither my defense, nor this thesis would have been possible: Andrew Ng, together with Yoav Shoham and Bart Selman on the reading committee, and Serafim Batzouglu and Yossi Feinberg on orals.

The work in this thesis represents enormous investment of computational time. I have come to regard the clusters that I used to run these experiments as essentially my co-authors; they certainly seem to have different moods, personalities, their personal ups and downs. Initial experiments were run on the unforgettable "zippies" in Cornell, kindly provided to us by Carla and Bart. Eventually, we built our own cluster — the "Nashes". I'm extremely grateful to our system administrator, Miles Davis, for keeping Nashes healthy from their birth. His initial reaction, when we approached him about building the cluster, was: "It's gonna be so cool!". And it was cool ever since; Nashes have been continuously operating for several years, with little idle time.

The work in this thesis was funded by a number of sources. Initially, I was funded by the Siebel Scholar fellowship. Most of the work, however, was funded by the NSF grant IIS-0205633 and DARPA grant F30602-00-2-0598. Our trip to Cornell, and the use of the "zippies" cluster were partially funded by the Cornell Intelligent Information Systems Institute.

Outside of Stanford, I count myself lucky in having lots of friends who always provided me with means to escape the academic routine. There are too many to mention here without leaving out somebody important; scattered around the globe. You know who you are! Thank you!

Finally, I'd like to thank my parents, family, and Sasha, though I can hardly even begin to express the gratitude for all the support I've had all my life. My parents, in

particular, are responsible for giving me the kick that I needed to start applying to grad schools, even though they well realized how hard it would be on them to have me gone. They always provided a firm base that I could count on, and a home I could come back to. Without them, I simply would not be.

Looking back, I am very glad that it snowed in Toronto on a warm April day five years ago, prompting me to choose Stanford after long deliberation; it has been an unforgettable journey ever since.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Complexity

Fundamentally, this thesis is about complexity. Complexity became truly inherent in computer science at least since it was in some sense formalized by Cook [1971] and Levin [1973]; in reality, it has been a concern of computer science since the inception of the field. I would argue that the mainstream perspective in computer science (which, by no means, is the only existing one) is heavily influenced by a particular view that really goes back to the logical beginnings of CS, the works of Church and Turing, if not earlier. One of the first (and, certainly, true) things that students are taught in the "foundational" CS courses is that we can think of the computational problems as *formal* languages — *i.e.,* sets of strings with certain properties. Algorithms, then, become simply mappings from one set of strings to another. The work of Cook [1971] firmly cemented the dimension of time (read – complexity) to concrete realizations of such mappings, but it didn't change the fact that algorithms are formal artificial objects. The fallacy that often follows this observation lies in the fact that formal or artificial objects *must* be studied by analytical formal methods.

There is another perspective that seems to be dominant at least among theoretically-inclined CS researchers. Often the algorithms are viewed as somehow being secondary to the computational problems. A well-established computer scientist once even said to me that "algorithms are merely tools, like microscopes, that allow us to study the underlying problem". This is certainly also a very much valid and useful point of view. Indeed, in the second part of this thesis, we'll subscribe to this view ourselves. I hope to demonstrate, at least via the first part, that yet again, this is not the only possible view.

Let us examine closer some ramifications that the views described above had on computer science, and, in particular, on the study of complexity. First, the view of algorithms as being secondary causes most work to focus on the *aggregate* picture of complexity; *i.e.,* complexity of a problem as a whole, and not of particular instances. Indeed, the complexity of a single instance is simply undefinable irrespectively of an algorithm, for one can always create an algorithm that solves any *particular* instance in constant time by a simple lookup. In the most classical case this aggregation takes form of the *worst-case* complexity (*i.e.,* the max operator). A slightly more informative view is obtained by adding to the mix some (usually, uniform) *probability distribution* over problem instances or some randomness to the algorithms. This leads to the notion of *average-case* complexity — still an aggregation, with max replaced by the expectation. In certain cases instead of a distribution a *metric* can be imposed, leading to such notions as *smoothed* complexity. None of these notions are concerned with individual instances.

This problem of not being concerned with minutiae is compounded by the formal approach. Instead of specifics, a lot of work focuses on the asymptotics — theoretical bounds and limits. One cannot really hope to do much better, at least without fixing an algorithm. For example, the notion of the worst-case problem instance is meaningless for the same reason as above: we can always come up with an algorithm that would be tremendously efficient on any particular instance. Unfortunately, the practical implications of such theoretical bounds sometimes border on being absurd. Here is an anecdote from a fellow student Sergei Vassilvitskii, one of many such. He was examining some well-cited (and, apparently, rather insightful) work that showed that certain peer-to-peer systems achieve very good performance when being used by *sufficiently* many people. Out of curiosity, Sergei actually calculated the number of users at which presented bounds would take effect. It came out to the order of 10^{27} people — probably more than the universe will bear in any foreseeable future.

Besides possibly very exciting and useful proof techniques, it is not clear what can be drawn from such work.

1.2 Empirical Complexity

In no way do I wish to suggest that traditional CS undertakings are useless or futile. On the contrary, our understanding of the state of the world has been steadily advancing. In the end we *are* dealing with formal objects, and so formal understanding of computation is still necessary. However, in this thesis we'll take a *complementary* view of complexity that overcomes some of the shortcomings listed above.

In order to get to this complementary view, we are going to make one important philosophical (or, at least, methodological) distinction. We are going to think of both computational problems and algorithms as *natural*, not artificial, phenomena. In the first part of this thesis our fundamental subject of study is going to be a triple consisting of a space of possible problem instances, a probability distribution over those instances, and an algorithm that we wish to study. In the second part we are going to take a slightly more traditional view and treat algorithms as tools for studying the underlying problem (though these tools will turn out to be very useful as algorithms).

Once we take on this perspective, one course of action readily suggests itself. We should take a hint from natural sciences and approach these "natural" phenomena with *empirical* studies. That is, running experiments, collecting data, and mining that data for information can give us a lot of insight; insight that, as we'll see, can later be used to come up with better formal models and shine light on important new research directions. In a sense, empirical approach will allow us to study a different kind of complexity, which we'll call *empirical complexity*.

Definition 1.1 *The* **empirical complexity** *of an instance $\mathcal{I}$ with respect to an (implementation of an) algorithm $\mathcal{A}$ is the actual running time of $\mathcal{A}$ when given $\mathcal{I}$ as input.*

Empirical complexity has also been variously called **typical-case complexity** and

empirical hardness.

This new notion of complexity leads to a complementary perspective in several different directions. First, this allows for a *comprehensive*, rather than aggregate, view, since we are now working on the scale of instances. Second, after going to the level of particular implementations, we can start making statements about *real* running times, as opposed to bounds and limits.

Perhaps the most important distinction is that this view will allow us to get a better handle on input *structure*, as opposed to the traditional input *size*. After all, already Cook [1971], in his discussion of the complexity of theorem-proving procedures, suggested that time dependent only on the input size is too crude of a complexity measure, and that additional variables should also play a role. It seems that for the most part, problem size just stuck since then. While in the limit size might be the only thing that matters, where empirical complexity is concerned structure becomes paramount. For example, throughout many experiments with the instances of combinatorial auctions winner determination problem (see Chapter 3), I never saw a clear dependence of running times on input size. No matter what size we tried, we would always see trivial instances that took fractions of a second to solve, as well as instances that took more than our (extremely generous) patience allowed. Though the hardest instances probably did get harder with size, the picture was not clear for any reasonable-sized input that we could generate. It might not take 10^{27} participants to have a tangible effect in this case, but it is clear that understanding the dependence on the elusive "problem structure" is crucial.

1.3 Contributions and Overview

The most important contribution of this thesis is in demonstrating how the empirical approach to CS complements a more traditional one. In doing so, a number of much more concrete and tangible results have been obtained. These include a novel methodology for approaching empirical studies, identification of structural properties that are relevant to hardness in two specific domains, introduction of a new testbed and novel algorithms, and, as the ultimate result, a multitude of important new research directions.

The rest of this thesis is broken into two parts. The first one takes to heart the definition of empirical complexity and demonstrates how one can study it with respect to particular algorithms. In that part we present and validate a general methodology for these kinds of studies. The second part takes a closer look at the domain of computational game theory. Via that domain, it demonstrates how algorithms, together with the experimental mindset, can help to uncover interesting facts about the underlying problem domain.

1.3.1 Algorithms as Subjects

In Chapter 2 we propose a new approach for understanding the empirical complexity of $\mathcal{NP}$-hard problems. We use machine learning to build regression models that predict an algorithm's runtime given a previously unseen problem instance. We discuss techniques for interpreting these models to gain understanding of the characteristics that cause instances to be hard or easy. We also describe two applications of these models: building algorithm portfolios that can outperform their constituent algorithms, and generating test distributions to focus future algorithm design work on problems that are hard for an existing portfolio. We also survey relevant literature.

In Chapter 3 we demonstrate the effectiveness of all of the techniques from Chapter 2 in a case study on the combinatorial auctions winner determination problem. We show that we can build very accurate models of the running time of CPLEX — the state-of-the-art solver for the problem. We then interpret these models, build an algorithm portfolio that outperforms CPLEX alone by a factor of three, and tune a standard benchmark suite to generate much harder problem instances.

In Chapter 4 we validate our approach in yet another domain — random k-SAT. It is well known that the ratio of the number of clauses to the number of variables in a random k-SAT instance is highly correlated with the instance's empirical hardness. We demonstrate that our techniques are able to automatically identify such features. We describe surprisingly accurate models for three SAT solvers — `kcnfs`, `oksolver` and `satz`— and for two different distributions of instances: uniform random 3-SAT with varying ratio of clauses-to-variables, and uniform random 3-SAT with fixed ratio of clauses-to-variables. Furthermore, we analyze these models to determine which features are most useful in predicting whether a SAT instance will be hard to solve. We also discuss the use of our models to build `SATzilla`, an algorithm portfolio for SAT. Finally, we demonstrate several extremely interesting research directions for the SAT community that were highlighted as a result of this work.

1.3.2 Algorithms as Tools

In Chapter 5 we explain the relevance of game theory to computer science, give a brief introduction to game theory, and introduce exciting game-theoretic computational problems.

In Chapter 6 we present GAMUT[1], a suite of game generators designed for testing game-theoretic algorithms. We explain why such a generator is necessary, offer a way of visualizing relationships between the sets of games supported by GAMUT, and give an overview of GAMUT's architecture. We highlight the importance of using comprehensive test data by benchmarking existing algorithms. We show surprisingly large variation in algorithm performance across different sets of games for two widely-studied problems: computing Nash equilibria and multiagent learning in repeated games.

Finally, in Chapter 7 we present two simple search methods for computing a sample Nash equilibrium in a normal-form game: one for 2-player games and one for n-player games. Both algorithms bias the search towards supports that are small and balanced, and employ a backtracking procedure to efficiently explore these supports. We test these algorithms on many classes of games from GAMUT, and show that they perform well against the state of the art — the Lemke-Howson algorithm for 2-player games, and Simplicial Subdivision and Govindan-Wilson for n-player games. This conclusively demonstrates that most games that are considered "interesting" by researchers must posses very "simple" Nash equilibria.

[1]Available at `http://gamut.stanford.edu`

Part I

Algorithms as Subjects

Chapter 2

Empirical Hardness: Models and Applications

In this chapter we expand on our discussion of the need for having good statistical models of runtime. We present a methodology for constructing and analyzing such models and several applications of these models. Chapters 3 and 4 validate this methodology in two domains, combinatorial auctions winner determination problem and SAT.

2.1 Empirical Complexity

It is often the case that particular *instances* of $\mathcal{NP}$-hard problems are quite easy to solve in practice. In fact, classical complexity theory is never concerned with solving a given problem instance, since for every instance there always exists an algorithm that is capable of solving that particular instance in polynomial time. In recent years researchers mostly in the artificial intelligence community have studied the *empirical* hardness (often called *typical-case complexity*) of individual instances or distributions of $\mathcal{NP}$-hard problems, and have often managed to find simple mathematical relationships between features of problem instances and the hardness of a problem. Perhaps the most notable such result was the observation that the ratio of the number of clauses to the number of variables in random k-SAT formulae exhibits strong correlation with both the probability of the formula being solvable, and its the apparent hardness [Cheeseman et al. 1991; Selman et al. 1996]. The majority of such work has focused on decision problems: that is, problems that ask a yes/no question of the form, "Does there exist a solution meeting the given constraints?".

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?". These problems are clearly different from decision problems, since they always have solutions. In particular, this means that they cannot give rise to phenomena like phase transitions in the probability of solvability that were observed in several $\mathcal{NP}$-hard problems. 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 the value of the objective function $\geq x$?" This approach has yielded promising results when applied to MAX-SAT and TSP. Unfortunately, it fails when the expected value of the solution depends on input factors irrelevant to hardness (*e.g.,* in MAX-SAT scaling of the weights has effect on the value, but not on the combinatorial structure of the problem). Some researchers have also tried to understand the empirical hardness of optimization problems through an analytical approach. (For our discussion of the literature, see Section 2.5.1.)

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 also have other weaknesses. 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. Thus, if we are to better understand empirical hardness of instances of such problems, a new experimental approach is called for.

The idea behind our methodology in some sense came from the basic goal of artificial intelligence research: if we cannot analyze the problem either empirically, or theoretically, ourselves, why not make computers do the work for us? More precisely, it is actually possible to apply *machine learning* techniques in order to *learn* parameters that are relevant to hardness. Philosophically, this approach to the study of complexity is reminiscent of the classical approach taken in natural sciences. When natural phenomena (problems and algorithms in our case) are too complicated to understand directly, we instead attempt to collect a lot of data and measurements, and then mine it to create *statistical* (as opposed to analytical) models[1].

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. For example, in case of the combinatorial auctions winner determination problem (WDP) (see Chapter 3), this will allow auctioneers to know how long an auction will take to clear. More generally, 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 will have to be used instead.

Second, it has often been observed that algorithms for $\mathcal{NP}$-hard 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 3.4, and SAT in Figure 4.6.) However, little is known about what causes these instances to vary so substantially in their empirical hardness. In Section 2.3 we explain how analyzing our runtime models can shine light on the sources of this variability, and in Chapters 3 and 4 we apply these ideas to our case studies. This sort of analysis could lead to changes in problem formulations to reduce the chance of long solver runtimes. Also, better understanding of high runtime variance could serve as a starting point for

[1]We note that this methodology is related to approaches for statistical experiment design (see, *e.g.,* [Mason et al. 2003; Chaloner and Verdinelli 1995]).

improvements in algorithms that target specific problem domains.

Empirical hardness models also have other applications, which we discuss in Section 2.4. First, we show how accurate runtime 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 which complement the existing state of the art. These ideas are validated experimentally in Chapter 3.

2.2 Empirical Hardness Methodology

We propose the following methodology for predicting the running time of a given algorithm on individual instances drawn from some arbitrary distribution.

1. **Select an algorithm of interest.**

2. **Select an instance distribution.** Observe that since we are interested in the investigation of *empirical* hardness, the choice of distribution is fundamental — different distributions can induce very different algorithm behavior. It is convenient (though not necessary) for the distribution to come as a set of parameterized generators; in this case, a distribution must be established over the generators and their parameters.

3. **Define problem size (or known sources of hardness).** Problem size can then be held constant to focus on unknown sources of hardness, or it can be allowed to vary if the goal is to predict runtimes of arbitrary instances.

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

5. **Collect data.** Generate a desired number of instances by sampling from the distribution chosen in step 2, setting the problem size according to the choice

made in step 3. For each problem instance, determine the running time of the algorithm selected in step 1, and compute all the features selected in step 4. Divide this data into a training set and a test set.

6. **Learn a model.** Based on the training set constructed in step 5, 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.

2.2.1 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 which is produced through the application of this methodology will be algorithm-specific, and thus can never directly provide information about a problem domain which transcends 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 2.3.) We do not consider the algorithm-specificity of our techniques to be a drawback — it is not clear what algorithm-independent empirical hardness would even mean — but the point deserves emphasis.

While Chapter 3 focuses only on deterministic algorithms, we have also had success in using our methodology to build empirical hardness models for randomized search algorithms (see Chapter 4). Note that our methodology *does not* apply as directly to incomplete algorithms, however. When we attempt to predict an algorithm's running time on an instance, we do not run into an insurmountable problem when the actual running time varies from one invocation to another. For incomplete algorithms, however, even the notion of running time is not always well defined because the algorithm can lack a termination condition. For example, on an optimization problem such as the WDP, an incomplete algorithm will not know when it has found the optimal allocation. On a decision problem such as SAT, an incomplete algorithm will know that it can terminate when it finds a satisfying assignment, but will never know when it has been given an unsatisfiable instance. We expect that techniques similar to those presented here will be applicable to incomplete algorithms; however, this is a topic for future work.

In principle, it is equally possible to predict some other measure of empirical hardness, or even some other metric, such as solution quality. While we've also had some success with the latter in the Traveling Salesman problem domain, in this thesis we'll focus exclusively on the running time as it is the most natural and universal measure.

2.2.2 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 thesis we consider instances that were created by artificial instance generators; however, real-world instances may also be used. (Indeed, we did the latter when constructing `SATzilla` (see Section 4.5 in Chapter 4.) 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.

It is often the case that in the literature on a particular computational problem, a wide variety of qualitatively different instance distributions will 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 distribution is chosen at random and then an instance is drawn from the 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.

2.2.3 Step 3: Defining Problem Size

Some sources of empirical hardness in $\mathcal{NP}$-hard problem instances are already well understood; in particular, as problems get larger they also get harder to solve. However, as we illustrate when we consider this step in our case study (Section 3.3 in Chapter 3), there can be multiple ways of defining problem size for a given problem. Defining problem size is important when the goal for building an empirical hardness model is to understand what *previously unidentified* features of instances are predictive of hardness. In this case we generate all instances so that 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 build an empirical hardness model that applies to problems of varying size; however, even in this case we must define the way in which problem size varies in our instance distribution, and hence problem size must be clearly defined. Another advantage of having problem size defined explicitly is that its relationship to hardness may be at least approximately known. Thus, it might be possible to tailor hypothesis spaces in the machine learning step to make direct use of this information.

2.2.4 Step 4: Selecting Features

An empirical hardness model is a mapping from a set of features which 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;

researchers must use domain knowledge to identify properties of instances that appear likely to provide useful information. However, we did discover that a lot of intuitions can be generalized. For example, many features that proved useful for one constraint satisfaction or optimization problem can carry over into another. Also heuristics or simplified algorithms often make good features.

The good news is that techniques *do* exist for building good models even if the set of features provided includes 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 tries to use 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 which have the possibility to prove useful, and allow models to select among them.

We recommend that features that are extremely highly correlated with other features or extremely uninformative (*e.g.,* they always take the same value) be eliminated immediately, on the basis of some small initial experiments. Features which are not (almost) perfectly correlated with other features should be preserved at this stage, but should be re-examined if problems occur in Step 6 (*e.g.,* numerical problems arise in the training of models; models do not generalize well).

We do offer two guidelines to restrict the sorts of features that should be considered. First, we only consider features that can be generated from *any* problem instance, without knowledge of how that instance was constructed. For example, we do not use parameters of the 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 problem instance.

2.2.5 Step 5: 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 which is introduced in Step 4 increases the dimensionality of the learning problem, a very large amount of data may be required for the construction of good models. Fortunately, problem instances are available in large quantities, so the size of a dataset is often limited only by the amount of time one is willing to wait for it. This tends to encourage the use of large parallel computer clusters, which are luckily becoming more and more widely available. Of course, it is essential to ensure that hardware is identical throughout the cluster and that no node runs more jobs than it has processors.

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 fraction of a second to solve, hard instances of the same size may take many hours. (We see this sort of behavior in our WDP case study, for example, in Section 3.5.1.) Since runtimes will often be distributed exponentially, it may be infeasible to wait for every run to complete. Instead, it may be necessary to cap runs at some maximum amount of time.[2] 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, but capping should always be performed cautiously.

Finally, we have found data collection to be logistically challenging. When experiments involve tens of processors and many CPU-years of computation, jobs will crash, data will get lost, and it will become necessary to recover from bugs in feature-computation code. In the work that led to this thesis, we have learned a few general lessons. (None of these observations are especially surprising — in a sense, they all boil down to a recommendation to invest time in setting up clean data collection methods rather than taking quick and dirty approaches.) First, enterprise-strength queuing software should be used rather than attempting to dispatch jobs using home-made scripts. Second, data should not be aggregated by hand, as portions of experiments

[2]In the first datasets 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 dataset.

will sometimes need to be rerun and such approaches will become unwieldy. Third, for the same reason the instances used to generate data should always be kept (even though they can be quite large). 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.

2.2.6 Step 6: Building Models

Our methodology is agnostic on the choice of a particular machine learning algorithm to be used to construct empirical hardness models. Since the goal is to predict runtime, which is a continuous-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 we ultimately became convinced that they were less appropriate. (For a discussion of some of the reasons that we drew this conclusion, see Section 2.5.2.) Because of our interest in being able to analyze our models and in keeping model sizes small (*e.g.,* so that models can be made publicly available as part of an algorithm portfolio), we have avoided approaches such as nearest neighbor or Gaussian processes; however, there may be applications for which these techniques are the most appropriate.

There are a wide variety of different regression techniques; the most appropriate for our purposes perform supervised learning[3]. Such techniques choose a function from a given hypothesis space (*i.e.,* a space 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.

[3] A large literature addresses these statistical techniques; for an introduction see, *e.g.,* [Hastie et al. 2001].

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 i^{th} feature and the w's are free variables, and has as its error metric root mean squared error (RMSE). Geometrically, this procedure tries to construct a hyperplane in the feature space that has the closest ℓ_2 distance to data points. Linear regression is a computationally appealing procedure because it reduces to the (roughly) cubic-time problem of matrix inversion.[4] 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'll see in Section 2.3.

While we will discuss other regression techniques later in Section 2.5, we will present linear regression as our baseline machine learning technique.

Choosing an Error Metric

Linear regression uses a squared-error metric, which corresponds to the ℓ_2 distance between a point and the learned hyperplane. Because this measure penalizes outlying points superlinearly, it can be inappropriate in cases where data contains many outliers. Some regression techniques use ℓ_1 error (which penalizes outliers linearly); however, optimizing such error metrics often requires solution of a quadratic programming problem.

Some error metrics express an additional preference for models with small (or even zero) coefficients to models with large coefficients. This can lead to 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.

[4] In fact, the worst-case complexity of matrix inversion is $O(N^{log_2 7}) \doteq O(N^{2.807})$.

Choosing a Hypothesis Space

Although linear regression seems quite limited, it can actually 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 of the original features. For example, in order to learn a model which is a quadratic function of the features, the feature set can be augmented to include all pairwise products of features. A hyperplane in the resulting much-higher-dimensional space corresponds to a quadratic manifold in the original feature space. The key problem with this approach is that the size of the new set of features is the square of the size of the original feature set, which may cause the regression problem to become intractable (*e.g.,* because the feature matrix cannot fit into memory). There is also the more general problem that using a more expressive hypothesis space can lead to overfitting, because the model can become expressive enough to fit noise in the training data. Thus, in some cases it can make sense to add only a subset of the pairwise products of features; *e.g.,* only pairwise products of the k most important features in the linear regression model. Of course, we can use the same idea to reduce many other nonlinear hypothesis spaces to linear regression: all hypothesis spaces which can be expressed by $\sum_i w_i g_i(\mathbf{f})$, where the g_i's are arbitrary functions and $\mathbf{f} = \{f_i\}$.

Sometimes we want to consider hypothesis spaces of the form $h\left(\sum_i w_i g_i(\mathbf{f})\right)$. For example, we may want to fit a sigmoid or an exponential curve. When h is a one-to-one function, we can transform this problem to a linear regression problem by replacing the response variable y in the training data by $h^{-1}(y)$, where h^{-1} is the inverse of h, and then training a model of the form $\sum_i w_i g_i(\mathbf{f})$. On test data, we must evaluate the model $h\left(\sum_i w_i g_i(\mathbf{f})\right)$. One caveat about this trick is that it distorts the error metric: the error-minimizing model in the transformed space will not generally be the error-minimizing model in the true space. In many cases this distortion is acceptable, however, making this trick a tractable way of performing many different varieties of nonlinear regression. In this thesis, unless otherwise noted, we use exponential models ($h(y) = 10^y$; $h^{-1}(y) = \log_{10}(y)$) and logistic models ($h(y) = 1/(1 + e^{-y})$; $h^{-1}(y) = \ln(y) \ln(1-y)$ with values of y first mapped onto the interval $(0, 1)$). Because logistic functions have a finite range, we found them particularly useful for modeling capped runs.

It seems that exponential (and, similarly, logistic) models provide a better hypothesis space for predicting running times than linear models do. The problem is that in our data we encounter a large number of very easy instances, with small runtimes. In order to fit these, linear regression necessarily must sometimes predict negative values, resulting in bad overall performance. With the exponential models, these negative predictions get appropriately transformed into small positive values after exponentiation.

2.3 Analyzing Hardness Models

In the previous section we have explained how it is possible to learn a statistical model that accurately predicts algorithm's runtime on given instances. For some applications accurate prediction is all that is required. For other applications it is necessary to *understand* what makes an instance empirically hard. In this section we explain one way to interpret our models.

2.3.1 Evaluating the Importance of Variables

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 then models can include larger-than-necessary coefficients with different signs. For example, suppose that we have to identical and completely unimportant features $f_i \equiv f_j$. Then all models where $w_i = -w_j$ are equally good, even if in some of them $w_i = 0$, while in others $|w_i| \to \infty$.

A better approach is to force models to contain fewer variables, on the principle that the best low-dimensional model will involve only relatively uncorrelated features since 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 four heuristic methods: *forward selection*, *backward elimination*, *sequential replacements* and *least-angle regression (LAR)*.

Forward selection starts with an empty set, and greedily adds the feature that, combined with the current model, makes the largest reduction to cross-validated error. Backward elimination starts with a full model and greedily removes the features that yields the smallest increase in cross-validated error. Sequential replacement is like forward selection, but also has the option to replace a feature in the current model with an unused feature. [5] Finally, the recently introduced LAR [Efron et al. 2002] algorithm is a shrinkage technique for linear regression that can set the coefficients of sufficiently unimportant variables to zero as well as simply reducing them; thus, it can be also be used for subset selection.

Since none of these techniques is guaranteed to find the optimal subset, we combine them together by running all and keeping the model with the smallest cross-validated (or validation-set) error.

Besides being an important tool in model analysis, this procedure can be iterated to aid model construction as follows. For example, if we have a 100 base features, the full quadratic model would contain on the order of 5000 terms, which is very costly to train. Instead, one can first use subset selection techniques to select a subset of few (say 30) base features that is most predictive. Then, we can compute all quadratic terms involving just those 30 features to obtain a reasonable second-order model. We can once again, apply subset selection techniques on this new model to analyze

[5] 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].

relative variable importance of quadratic terms.

2.3.2 Cost of Omission

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*. That is, to evaluate $score(f_i)$ we can train a model using all features except for f_i and report the resulting increase in (cross-validated) prediction error compared to the full model. To make scores more meaningful, we scale the cost of omission of the most important feature to 100 and scale the other costs of omission in proportion. Notice, that this would not work in the presence of highly-correlated features: if $f_i \equiv f_j$ and both are very useful, then dropping either one will not result in any increase in the error metric, leading us to believe that they are useless.

We must discuss 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 size k, and this subset has validation-set error that is close to that 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 a small number of features. It must be stressed, however, that this does not amount to an argument that choosing the subset X is *necessary* for good performance in a subset of size k. Because variables are very often at least somewhat correlated, there may be other sets 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 draw conclusions about the variables that are *present* in small, well-performing subsets, but we must be very careful in drawing conclusions about the variables that are *absent*.

2.4 Applications of Empirical Hardness Models

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 $\mathcal{NP}$-hard problems, because runtimes for these algorithms are often highly variable from instance to instance. When algorithms exhibit high runtime variance, one is faced with the problem of deciding which algorithm to use; Rice dubbed this the "algorithm selection problem" [Rice 1976]. 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. By far, the most common approach to algorithm selection has been to measure different algorithms' performance on a given problem distribution, and then to use only the algorithm having the lowest average runtime. This approach, to which we refer as "winner-take-all," has driven recent advances in algorithm design and refinement, but has resulted in the neglect of many algorithms that, while uncompetitive on average, offer excellent performance on particular 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, once we have rejected 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.

2.4.1 The Boosting Metaphor

Boosting is a machine learning paradigm due to Schapire [1990] and widely studied since. Although we do not make use of any technical results from the boosting literature, we take 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.

Of course the analogy to boosting is imperfect; we discuss differences in Section 2.5.4.

2.4.2 Building Algorithm Portfolios

In the presence of accurate algorithm-specific models of the empirical hardness of given distributions of problem instances, we can build portfolios of multiple algorithms in a very straightforward way:

1. Train a model for each algorithm, as described in Section 2.2.

2. Given an instance:

    (a) Compute feature values.

    (b) Predict each algorithm's running time using runtime models.

    (c) Run the algorithm predicted to be fastest.

Overall, while we will show experimentally that our portfolios can dramatically outperform the algorithms of which they are composed, our techniques are also deceptively simple. For discussion and comparison with other approaches in the literature, please see Section 2.5.2.

We now turn to the question of enhancing our techniques specifically for use with algorithm portfolios.

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[6]. The portfolio can start by computing the easiest features, and iteratively compute the next set only if the expected benefit to selection exceeds the cost of computation. More precisely:

1. For each set S_j learn or provide a model $c(S_j)$ that estimates time required to compute it. Often, this could be a simple average time scaled by input size.

2. Divide the training examples into two sets. Using the first set, train models $M_1^i \ldots M_l^i$, 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 which 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 on S_k. The second set should be used to avoid training the difference models on data to which the runtime models were fit.

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

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.

Alternative Performance Measures

From the machine learning point of view, the task faced by algorithm portfolios is that of discrimination among algorithms. Thus, in principle, many standard discriminative machine learning algorithms (instead of our regression models) could be used. What makes this task different from standard classification is that we might be interested in different, non-trivial, performance (and, hence, error) measures.

Average runtime is an obvious measure of portfolio performance if one's goal is to minimize computation time over a large number of instances. Even with this measure it is clear that standard discrimination methods may not be appropriate: it is much more important to be correct on cases where wrong choice is very costly, than on cases where all algorithms have runtimes relatively close to each other. Since we use regression models that minimize root mean squared error on predictions of runtime, they appropriately penalize 20 seconds of error equally on instances that take 1 second or 10 hours to run. That is, they penalize the same *absolute error* in the same way regardless of the magnitude of the instance's runtime. Standard off-the-shelf classifications techniques face the same difficulty.

However, another motivation is to achieve good *relative error* on every instance regardless of its hardness — we might thus consider that a 20 second error is more significant on a 1 second instance than on a 10 hour instance. Let n be the number of instances; let r_i^p and r_i^* be the portfolio's selection runtime and the optimal selection runtime, respectively, on instance i. One measure that gives an insight into the portfolio's selection quality (regardless of hardness) is *percent optimal*:

$$\frac{1}{n}|\{i : r_i^* = r_i^p\}| \tag{2.1}$$

Another measure of relative error is *average percent suboptimal*:

$$\frac{1}{n}\sum_i \frac{r_i^p - r_i^*}{r_i^*}. \tag{2.2}$$

In this thesis we concentrate on the average runtime as the most natural metric.

Capping Runs

The portfolio methodology 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 in our approach, gathering perfect data for every instance can take an unreasonably long time. 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. 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. This approach is safe if the captime is chosen so that it is (almost) always significantly greater than the minimum of the algorithms' runtimes; if not, it might 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 a 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.

2.4.3 Inducing Hard Distributions

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. In order to understand what this means it is first essential to choose a distribution D that reflects the problems that will be encountered in practice. 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 importance of a region of problem space is proportional to the amount of time the current portfolio spends working on instances in that region. 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 (let $D \cdot h_f(x) = \frac{D(x)h_f(x)}{\int Dh_f}$). However, it is problematic to sample from $D \cdot h_f$: D may be non-analytic (an instance generator), 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. This method works best when another distribution is available to guide the sampling process toward hard instances. 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. We can generate instances from $D \cdot h_f$ in the following way:[7]

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 probability density function, h_p.

2. Generate a large number of instances from $D \cdot h_p$. Observe that we *can* sample from this distribution: h_p is a polynomial, so we can sample from it directly;

[7]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 doesn't require us to normalize H_f and allows us to output an instance after generating a constant number of samples.

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.

Observe that if h_p turns out to be helpful, hard instances from $D \cdot h_f$ will be encountered quickly. Even in the worst case where h_p directs the search away from hard instances, observe that we still sample from the correct distribution because the weights are divided by $h_p(s)$.

In practice, D may be factored as $D_g \cdot D_{p_i}$, where D_g is a distribution over otherwise unrelated instance generators with different parameter spaces, and D_{p_i} is a distribution over the parameters of the chosen instance generator i. In this case it is 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 $H_{f,i}$ and normalize each to a pdf $h_{f,i}$.[8] 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.

[8] However, the experimental results presented in Figures 3.31–3.33 use hardness models H_f trained on the whole dataset rather than using models trained on individual distributions. Learning new models could be expected to yield even better results.

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.

Inducing "Realistic" Distributions

It is important for our portfolio methodology that we begin with a "realistic" D: that is, a distribution accurately reflecting the sorts of problems expected to occur in practice. Care must always be taken to construct a generator or set of generators producing instances that are representative of problems from the target domain. Sometimes, it is possible to construct a function R_f that scores the realism of a generated instance based on features of that instance; such a function can sometimes encode additional information about the nature of realistic data that cannot easily be expressed in a generator. If a function R_f is provided, we can construct D from a parameterized set of instance generators by using R_f in place of H_f above and learning r_p in the same way we learned h_p. Given these distributions, the techniques described in the previous section are guaranteed to generate instances with increased average realism scores.

2.5 Discussion and Related Work

2.5.1 Typical-Case Complexity

Early work [Selman et al. 1996; Cheeseman et al. 1991] considered the empirical performance of DPLL-type solvers running on uniform random k-SAT instances, finding 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, it was 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[9]. Similar phenomenon has also been observed in other decision problems such as quasigroup completion [Gomes and Selman 1997]. This well-publicized finding led to increased enthusiasm for the idea of studying algorithm performance experimentally, using the same tools as are used to study natural phenomena. Follow-up work that took a closer look at runtime distributions includes [Gomes et al. 2000], which demonstrated 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.

Over the past decade, the study of empirical hardness has complemented the theoretical worst-case analysis of algorithms, leading to interesting findings and concepts. 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. Backbone has also been extended to optimization problems [Slaney and Walsh 2001], although it is often difficult to define for arbitrary problems and can be costly to compute. 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].

This flurry of activity also prompted more theoretical approaches. Kolaitis [2003] defined and studying "islands of tractability" of hard problems. Analytical attempts to understand the empirical hardness of optimization problems include Zhang [1999], who performed average case theoretical analysis of particular classes of search algorithms. Though his 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 has made theoretical contributions — describing a

[9] Though [Coarfa et al. 2000] pointed out that this is quite algorithm-specific, as well as that runtimes are still exponential for higher ratios.

polynomial/exponential-time transition in average-case complexity — and shed light on real-world problems. Korf and Reid [1998] predict 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 are required: *e.g.,* that the branching factor is constant and node-independent, and that edge costs are uniform throughout the tree.

2.5.2 Algorithm Selection

It has long been understood that algorithm performance can vary substantially across different classes of problems. 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. The main methodological difference with our approach is that we recognize that the actual step of algorithm selection is trivial and need not be learned, if we can accurately learn the performance measure (time, in our case).

We explain our choice of methodology by relating it to other approaches for algorithm selection that have been proposed in the literature.

Parallel Execution

One tempting alternative to portfolios that select a single algorithm is the parallel execution of all available algorithms. 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. 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 portfolio that executes all algorithms in parallel on instance x will always take time $nt^*(x)$.

In some domains, parallel execution *can* be a very effective technique. Gomes and Selman [2001] proposed such an approach for incomplete SAT algorithms, using the term *portfolio* to describe a set of algorithms run in parallel. In this domain runtime depends heavily on variables such as random seed, making it difficult to predict; thus, parallel execution is likely to outperform a portfolio that chooses a single algorithm. In such cases it is possible to extend our methodology to allow for parallel execution. We can add one or more new algorithms to our portfolio, with algorithm i standing as a placeholder for the parallel execution of k_i of the original algorithms; in the training data i would be given a running time of k_i times the minimum of its constituents. This approach would allow portfolios to choose to task-swap sets of algorithms in parts of the feature space where the minimums of individual algorithms' runtimes are much smaller than their means, but to choose single algorithms in other parts of the feature space. Our use of the term "portfolio" may thus be seen as an extension of the term coined by Gomes and Selman, referring to a set of algorithms and a strategy for selecting a subset (perhaps one) for parallel execution.

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. Any standard classification algorithm (*e.g.,* a decision tree) could be used to learn which algorithm to choose given features of the instance and labeled training examples. The problem is that such non-cost-sensitive classification algorithms use the wrong error metric: 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 runtimes of the chosen and fastest algorithms is large than when it is small. This is just what happens when our decision criterion is to select the smallest prediction among a set of regression models that were fit to minimize root mean squared error.

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; 2002] used classification to predict runtime of CSP and SAT solvers with inherently high runtime variance (heavy tails). Despite its similarity to our portfolio methodology, this approach suffers from the same problem as described above. First, the learning algorithm does not use an error function that penalizes 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 runtime 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.

Markov Decision Processes

Perhaps most related to our methodology is work by Lagoudakis and Littman [2000; 2001]. They worked within the Markov decision processes (MDP) framework, and concentrated on recursive algorithms (*e.g.,* sorting, DPLL), sequentially solving the algorithm selection problem on each subproblem. This work demonstrates encouraging results; however, its generality is 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 'black-box' commercial or proprietary algorithms, which are often among the most competitive. Second, solving the algorithm selection problem recursively requires that the value functions be very inexpensive to compute; in our case studies we found that more computationally expensive features were required for accurate predictions of runtime.

Finally, these techniques can be undermined by non-Markovian algorithms, such as those using clause learning, taboo lists, 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 sequential decision-making.

Experts Algorithms

Another area of research which is somewhat related to algorithm selection is that of "experts algorithms" (*e.g.,* [de Farias and Megiddo 2004]). The setting studied in this area is the following. An agent must repeatedly act in a certain environment. It has access to the 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. The estimation step is in principle similar to our use of regression models to predict algorithm performance. The main difference between our work and the area of experts algorithms is that the latter do both learning and selection online. As a result, estimation often simply takes form of historical averages, and a lot of work goes into the study of the exploration-exploitation tradeoff.

Different Regression Approaches

Lobjois and Lemaître [1998] select among several simple branch-and-bound algorithms based on a prediction of running time. 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.

Since our goal is to discriminate among algorithms, it might seem more appropriate to learn models of pairwise differences between algorithm runtimes, rather than models of absolute runtimes. For linear regression (and the forms of nonlinear regression used in our work) it is easy to show that the two approaches are mathematically equivalent. Nonetheless, it can still be useful to think of portfolios as using models of pairwise differences, as the models are ultimately used for discrimination rather than prediction.

2.5.3 Hard Benchmarks

It is widely recognized that the choice of test distribution is important for 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 are known to be hard for most SAT solvers.

Recently, there has been a conscious effort in the SAT community to provide generators for hard instances. For example, Achlioptas et al. [2004] and Jia et al. [2005] hide pre-specified solutions in random formulae that appear to be hard. Jia et al. [2004] generate random hard formulae based on statistical physics spin-glass models, once again highlighting the connection between physical phenomena and phase transitions in SAT.

2.5.4 The Boosting Metaphor Revisited

Although it is helpful, our analogy to boosting is clearly not perfect. 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; the 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.

2.6 Conclusion

This chapter layed out the methodology that we can use to construct empirical hardness models. In Chapters 3 and 4 we validate this methodology by actually applying it in two different domains. In Chapter 3 we present the combinatorial auctions winner determination problem. We'll discuss features that can be used to describe instances of that problem, and demonstrate that accurate models can indeed be learned. In Section 3.7 we validate both our methodology for constructing algorithm portfolios and for inducing hard instance distributions. In Chapter 4 we demonstrate that good models can also be constructed in SAT domain. In that chapter we focus more on analyzing hardness models, and as a result highlight several exciting research directions for the SAT community.

Chapter 3

The Combinatorial Auctions Winner Determination Problem

In this chapter we will apply methodologies introduced in Chapter 2 to the winner determination problem, in order to both validate them and to shed some light onto the nature of the WDP.

3.1 Introduction

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: agents may bid for bundles of goods, with the guarantee that these bundles will be allocated "all-or-nothing". (For an introduction to the topic, see *e.g.,* [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 valuation for the goods contained in the bundle. They may also allow agents to specify that they consider some goods to be *substitutable*, *e.g.,* agents can 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 naturally arising in conjunction with combinatorial auctions. The goal of the WDP is to allocate sets of goods among the bidders given their bids, while optimizing seller's revenue. Formally, the WDP turns out to be $\mathcal{NP}$-hard. However, as has been observed by many researchers in the past, the WDP appears to be relatively easily solvable on *realistic* (as opposed to random) inputs.

The WDP thus seems to be a perfect candidate for the study of empirical complexity. Yet, it is a good example of a problem that is ill-suited for either existing experimental or theoretical approaches. 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.

The rest of this chapter is organized as follows. In Section 3.2 we describe the WDP, algorithms used to solve it, and the testbed on which these algorithms can be evaluated. In Section 3.3 we discuss the definition of the problem size for WDP instances. In Section 3.4 we describe the features that can be used to characterize a WDP instance. Then, in Section 3.5 we demonstrate that it is indeed possible to construct accurate runtime models for the WDP algorithms. Finally, in Section 3.7 we show how these hardness models can be used to both construct algorithm portfolios, and to induce much harder test distributions.

3.2 The Winner Determination Problem

In a combinatorial auction (see *e.g.,* [de Vries and Vohra 2003; Cramton et al. 2006]), a seller is faced with a set of price offers for various bundles of goods, and his aim is to allocate the goods in a way that maximizes his revenue. The *winner determination problem* (WDP) is choosing the subset of bids that maximizes the seller's revenue, subject to the constraint that each good can be allocated at most once. This problem is formally equivalent to weighted set packing.

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.

Problem 3.1 *The* Winner Determination Problem *is:*

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

3.2.1 WDP Algorithms

There are many WDP algorithms from which to choose, as much recent work has addressed this problem. A very influential early paper was [Rothkopf et al. 1998], but it focused on tractable subclasses of the problem and addressed computational approaches to the general WDP only briefly. The first algorithms designed specifically for the general WDP were published at IJCAI in 1999 [Fujishima et al. 1999; Sandholm 1999]; the authors of these papers subsequently improved and extended upon their algorithms in [Leyton-Brown et al. 2000b; Sandholm et al. 2001]. First-generation WDP solvers such as CASS [Fujishima et al. 1999] made use of classical AI heuristic search techniques, structuring their search by branching on goods. (In contrast, Sandholm's first-generation solver [Sandholm 1999] branched on bids.)

More recently, there has been an increasing interest in solving the WDP with branch-and-bound search, 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). In 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]. (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 also 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 in general, and CPLEX in particular, as the preferred approach to 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 the CABOB algorithm, and the latter is not publicly available. As we discuss in later sections, the experimental results in this chapter are based on several different datasets. Three of these datasets were used in our previously published conference papers on empirical hardness models [Leyton-Brown et al. 2002; Leyton-Brown et al. 2003b; Leyton-Brown et al. 2003a]; another represents new work which first appears in this chapter. Because a new version of CPLEX became available before we began to construct the last dataset, we upgraded our CPLEX software at this point. Thus, for our fixed-sized datasets we used CPLEX 7.1, and for the variable-sized dataset we used CPLEX 8.0.

3.2.2 Combinatorial Auctions Test Suite

The wealth of research into algorithms for solving the WDP created a need for many instances on which to test these algorithms. To date few unrestricted combinatorial auctions have been held, and little data has been publicly released from those auction that *have* been held. Thus, researchers have mostly evaluated their WDP algorithms using artificial distributions.

Legacy Data Distributions

Along with the first wave of algorithms for the WDP, seven distributions were proposed in [Sandholm 1999; Fujishima et al. 1999; Hoos and Boutilier 2000]. They 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 distributions select *which* goods to include in a bid uniformly at random without replacement. We give here the names that were used to identify them as "legacy" distributions in [Leyton-Brown et al. 2000a]; we use these names hereafter.

- **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. The price is drawn uniformly from $[0, 1]$.

- **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 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.

We used five of these distributions in our experiments. For reasons explained in Section 3.3, we did not use the distributions $L1$ and $L5$.

CATS Distributions

The above distributions were criticized in several ways, perhaps most significantly for lacking economic justification (see, *e.g.,* [Leyton-Brown et al. 2000a; Anderson et al. 2000; de Vries and Vohra 2003]). This criticism was significant because the WDP is simply 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 with the WDP in particular. To focus algorithm development more concretely on combinatorial auctions, Leyton-Brown et al. [2000a] introduced a new set of benchmark distributions called the Combinatorial Auction Test Suite (CATS). By modeling bidders explicitly and creating bid amounts, sets of goods and sets of substitutable bids from models of bidder valuations and models of problem domains, CATS distributions were aimed to serve as a step towards a realistic set of test distributions. (For example, note that none of the legacy distributions introduce any structure in the choice of *which* goods are included in a bundle; this is one way that the CATS distributions differ.)

In this chapter we consider four CATS distributions: regions, arbitrary, matching and scheduling. We provide a high-level description of each distribution; however, a more formal definition of each distribution is beyond the scope of this chapter. For a detailed description of CATS please refer to [Leyton-Brown et al. 2000a].

- **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, 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. 2000a] no efforts were made to tune the distributions to provide hard instances. In practice, some researchers have remarked that some CATS problems are comparatively easy (see *e.g.,* [Gonen and Lehmann 2000; Sandholm et al. 2001]). In Section 3.5.1 we show experimentally that some CATS distributions are always very easy for CPLEX, while others can be extremely hard. We come back to the question of whether these distributions could be made computationally harder in Section 3.7.2.

3.3 The Issue of Problem Size

Step 3 of our methodology in Section 2.2 prescribes us to identify known sources of hardness. Some sources of empirical hardness in $\mathcal{NP}$-hard problem instances are well understood. The reason for this step is to understand what *other* features of instances are predictive of hardness, so we hold these parameters constant, concentrating on variations in other features.

For the WDP, it is well known that problems become harder as the number of goods and bids increases.[1] 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.

Definition 3.2 *Bid b_i is* dominated by *bid b_j if $g(b_j) \subseteq g(b_i)$ and $p(b_i) \leq p(b_j)$.*

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

[1] 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].

if artificial distributions such as the legacy and CATS distributions generate a large number of dominated bids, it is possible that the apparent size of problems given as input to WDP algorithms could be much larger than the size of the problem that leaves the preprocessor. This means that we might be somewhat misled about the core algorithms' scaling behaviors — we might inadvertently learn about the preprocessors' scaling behavior as well.

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 has a substantial impact on algorithm behavior, or whether the relationship between the average number of non-dominated 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; however, the legacy distributions were much more variable. Figure 3.3 shows the number of non-dominated bids generated as a function of the total number of bids generated for the seven legacy distributions. In these experiments each data point represents an average over 20 runs. Bids were generated for an auction having 64 goods, and we terminated bid generation once 2000 non-dominated bids had been created.

Because of the generator-specific variation in the number of non-dominated bids generated in a fixed number of raw bids, we concluded that raw bids was not a reliable algorithm-independent proxy for the number of non-dominated bids. We therefore defined problem size as the pair (*number of goods, number of non-dominated bids*).

Of course, many other polynomial-time preprocessing steps are possible; *e.g.,* 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 poorer performance in terms of the number of bids discarded in a given amount of time. In any case, the results above suggest that strict domination checking

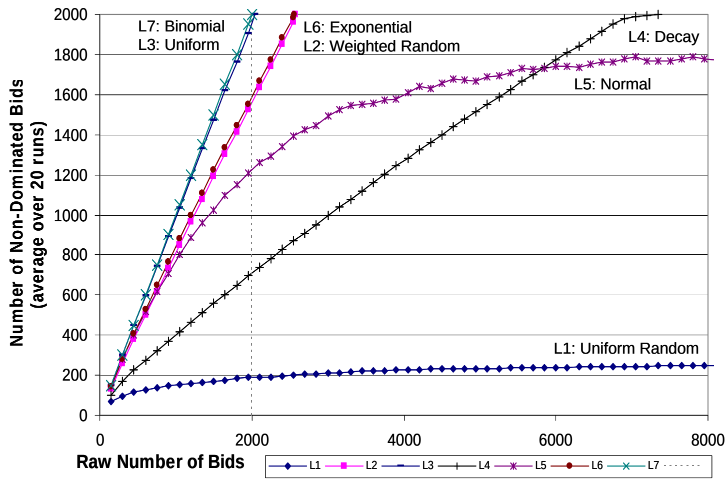


Figure 3.1: Non-Dominated Bids vs. Raw Bids.

should not be disregarded, since distributions differ substantially in the ratio between the number of non-dominated bids and the raw number of bids. Observe that if we want to be able to generate any given number of non-dominated bids then we will be unable to use the distributions L1 and L5, because they often fail to generate a target number of non-dominated bids even after millions of bids were created. (This helps to explain why researchers have found that their algorithms perform well on L1 (see *e.g.,* [Sandholm 1999; Fujishima et al. 1999]) and L5 (see *e.g.,* [Hoos and Boutilier 2000].)

3.4 Describing WDP Instances with Features

As described in Section 2.2, we must characterize each problem instance with a set of features.

We determined 35 features which we thought could be relevant to the empirical hardness of the WDP, ranging in their computational complexity from linear to cubic time. After having generated feature values for all our problem instances, we examined our data to identify redundant features. After eliminating these, we were left with 28 features, which are summarized in Figure 3.2. Of course for the variable size data the numbers of goods and bids were also used as two additional features. We

Bid-Good Graph Features:

1-4. **Bid nodes 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 node degrees.

Bid Graph Features:

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

10-15. **Node degree statistics**: maximum, minimum, standard deviation, first and third quartiles, and the median of the node degrees.

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**: One of the measures 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 the 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 (normalized).

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**: $\text{stdev}(p(b_i)/\sqrt{|g(b_i)|})$.

Figure 3.2: Four Groups of Features.

describe our features in more detail below, and also mention some of the redundant features that were eliminated.

There are two natural graphs associated with each instance; schematic examples of these graphs appear in Figure 3.3. 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: extremal and average degrees and their standard deviations for each group of nodes. The average number of goods per bid was perfectly correlated with another feature, and so did not survive our feature selection.

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 true for all CSPs, the BG captures a lot of useful information about the problem instance. Our second

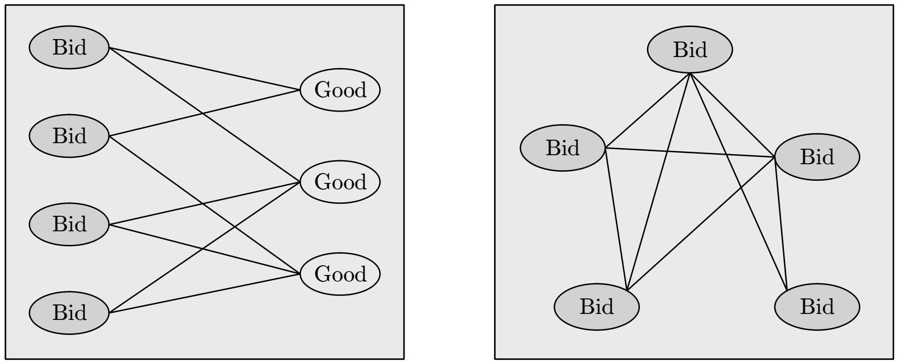


Figure 3.3: Examples of the Graph Types Used in Calculating Features 1–19: Bid-Good Graph (left); Bid Graph (right).

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 is decomposable into simpler instances, but found that virtually every instance consisted of a single component.[2]

The third group of features is calculated from the solution vector of the linear programming (LP) relaxation of the WDP. Recall that the WDP can be formulated as an integer program (Problem 3.1). To obtain the LP relaxation, we simply drop integrality constraints, and solve the resulting LP.

We calculate the *integer slack* vector by replacing each component x_i with $|0.5 - x_i|$. 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.

Our last 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

[2]It would 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 the 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.

schemes in [Gonen and Lehmann 2000].

3.5 Empirical Hardness Models for the WDP

Since the purpose of this work 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 both on fixed and variable-sized data.

For our fixed-size experiments we generated three separate data sets of different problem sizes, to ensure that our results were not artifacts of one particular choice of problem size. The first data set contained runtimes of CPLEX, CASS, and GL on instances of 1000 bids and 256 goods each, with a total of 4466 instances (roughly 500 instances per distribution). The second data set with 1000 bids and 144 goods had runtimes of CPLEX on a total of 4500 instances; the third data set with 2000 bids and 64 goods contained CPLEX's runtimes on 4493 instances. Where we present results for only a single fixed-size data set, the first data set was always used. All of our fixed-size CPLEX's runtime data was collected by running CPLEX's version 7.1 with preprocessing turned off.[3] 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 estimate that we consumed approximately 3 years of CPU time collecting this data. We also computed our 35 features for each instance. (Recall that feature selection took place after all instances had been generated.)

Our variable-size dataset was collected at a later time. The number of bids was randomly selected from the interval $[50, 2000]$, and the number of goods was drawn from $[40, 400]$[4]. We obtained runtimes for both CPLEX and CASS on 7146 of these

[3]When the work described in this chapter was performed, CPLEX 7.1 was the latest version. Unfortunately, it's not easy to rerun 3 CPU-years worth of experiments! On the bright side, limited experiments suggest that CPLEX 8.0 is not a huge improvement over CPLEX 7.1, at least for our WDP benchmark distributions.

[4]Due to the limitations imposed by CATS generators, for some instances the number of goods

instances (roughly 800 instances per distribution). CPLEX version 8.0 with default parameters was used to run these experiments. This time we capped CPLEX based on runtime rather than the number of nodes. The timeout for CPLEX was set to 1 CPU-week (and was reached on 2 instances from the **arbitrary** distribution). The timeout for CASS was set to 12 hours (the reason being that in our experience CASS was not likely to solve an instance within any reasonable time given that it already ran this long). The variable-size dataset 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 dataset, with CASS taking 6 hours on average. The second dataset took almost 9 CPU-years to collect!

3.5.1 Gross Hardness

Figure 3.4 shows the results of 500 runs for each distribution on problems with 256 goods and 1000 non-dominated bids, indicating the number of instances with the same order-of-magnitude runtime — *i.e.,* $\lfloor \log_{10}(runtime) \rfloor$. Each instance of each distribution had different parameters, each of which was sampled from a range of acceptable values. Figure 3.5 contains the same data for the variable-size dataset.

We can see that several of the CATS distributions are quite easy for CPLEX, and that others vary 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. This gives rise to the question of whether we can tune CATS so that in addition to generating "realistic" instances, it also generates the hardest possible instances? We present techniques that answer this question in Section 2.4.3.

3.5.2 Linear Models

As a baseline for other learning approaches, we performed 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 be learned. For

and bids actually fell outside of these ranges.

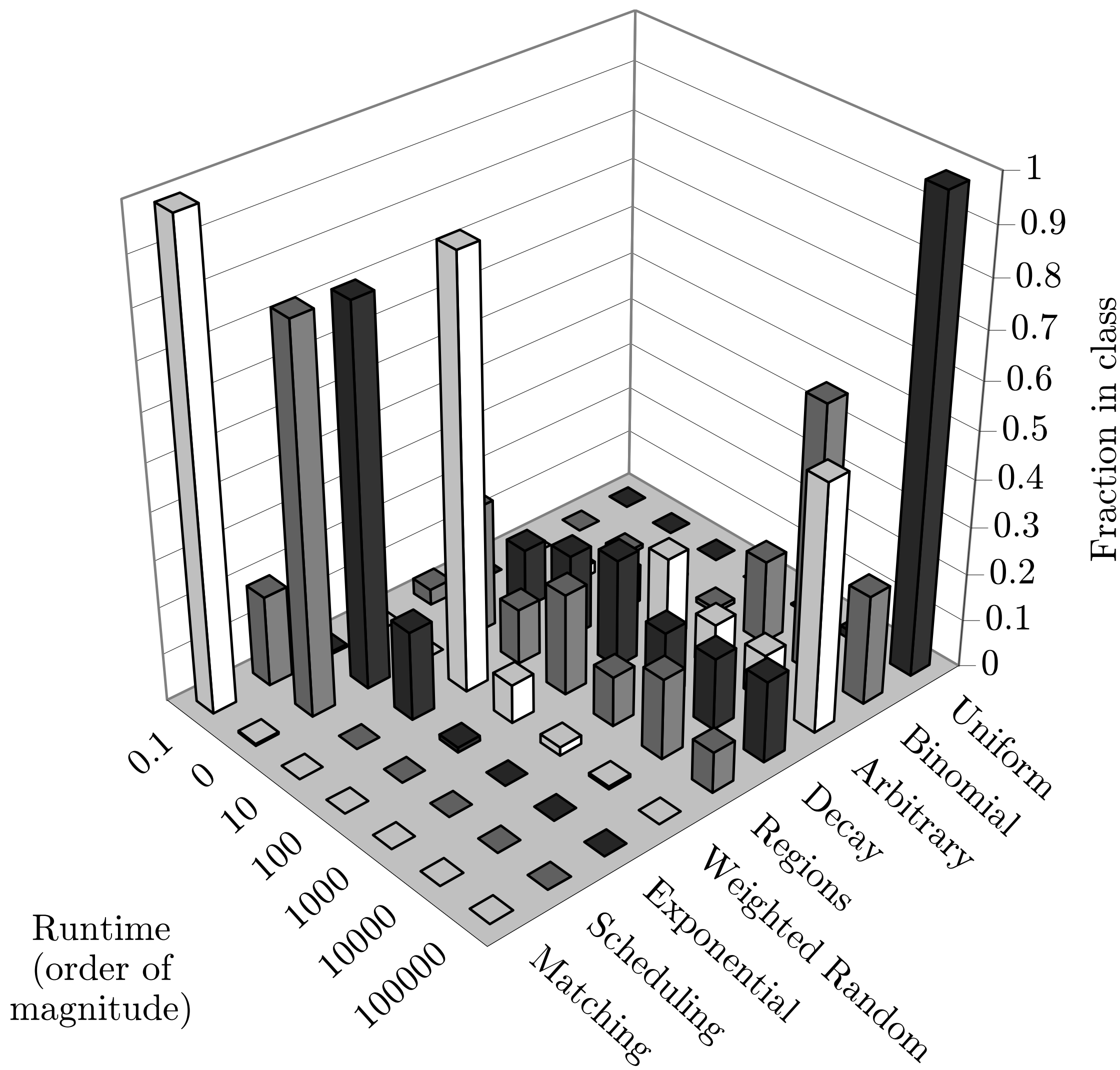


Figure 3.4: Gross Hardness, 1000 Bids/256 Goods.

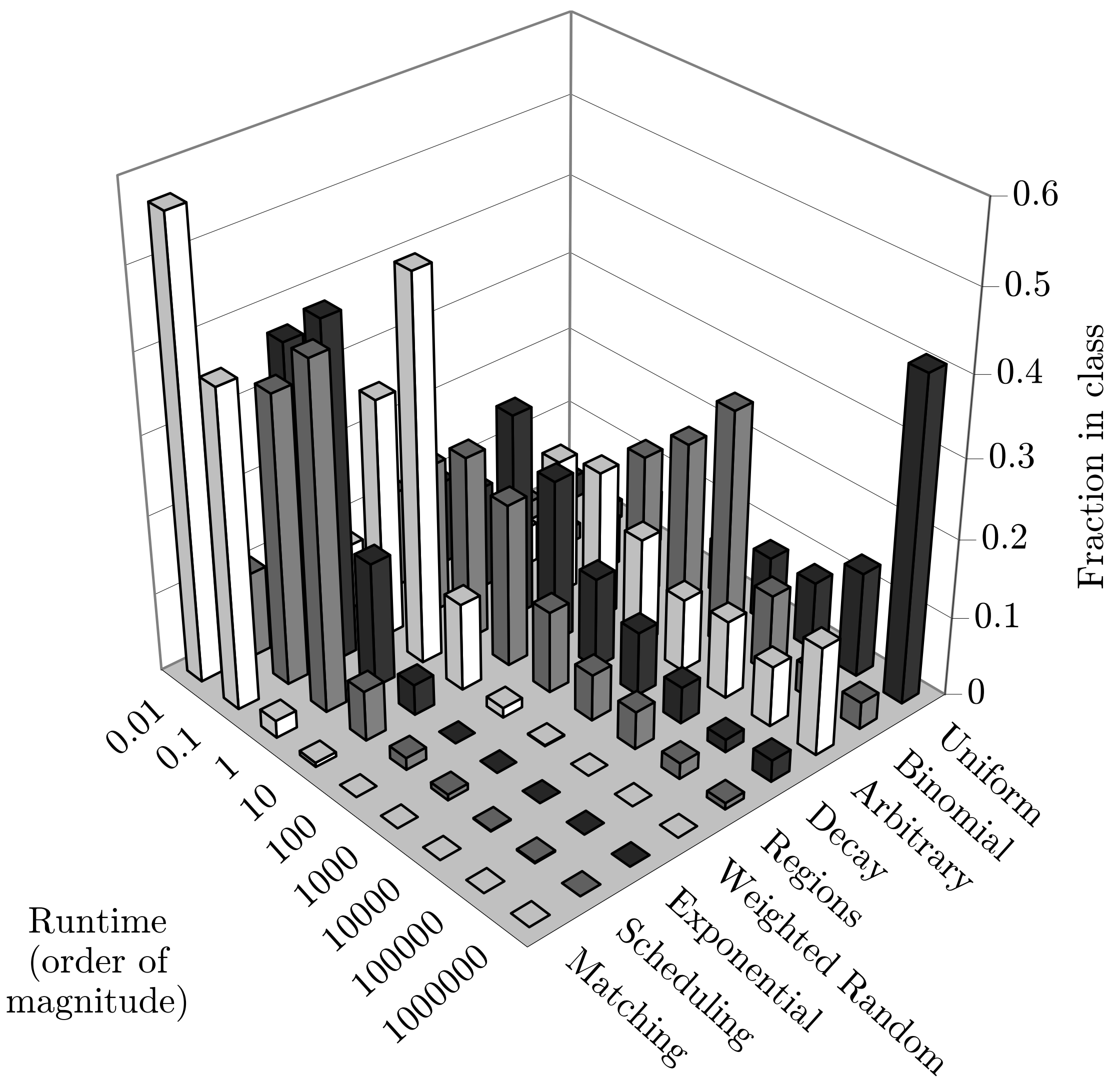


Figure 3.5: Gross Hardness, Variable Size.

Data point	Mean Abs. Err.	RMSE	Adj-R^2
1000 Bids/256 Goods	0.4294	0.5666	0.9351
1000 Bids/144 Goods	0.4000	0.5344	0.9162
2000 Bids/64 Goods	0.3034	0.4410	0.9284
Variable Size	0.9474	1.2315	0.6881

Table 3.1: Linear Regression: Errors and Adjusted R^2.

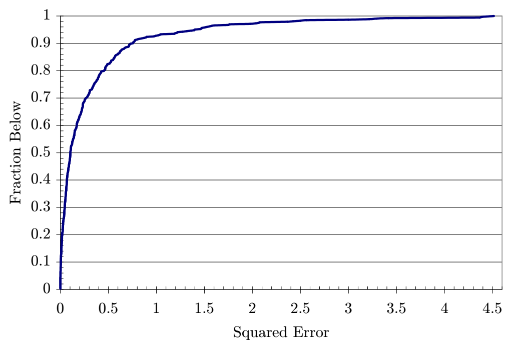


Figure 3.6: Linear Regression: Squared Error (test data, 1000 Bids/256 Goods).

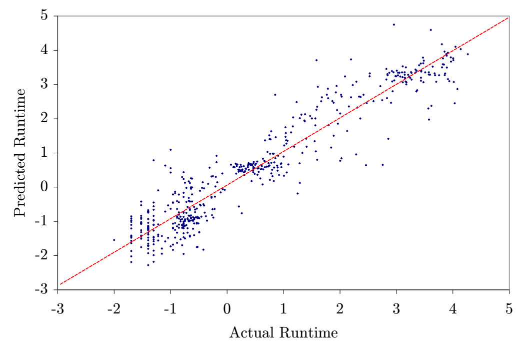


Figure 3.7: Linear Regression: Prediction Scatterplot (test data, 1000 Bids/256 Goods).

this experiment we chose the exponential hypothesis with the logarithm of CPLEX running time as our response variable — the value to be predicted — rather than absolute running time, because we wanted the model to be penalized according to whether the predicted and actual values had the same order of magnitude. 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 don't apply the inverse transformation $h(y)$ when reporting error metrics. Recall, also, that linear models by their nature have to provide a lot of negative predictions, which are clearly not very accurate.

In Table 3.1 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 a penalty for

Figure 3.8: Linear Regression: Squared Error (test data, Variable Size.)

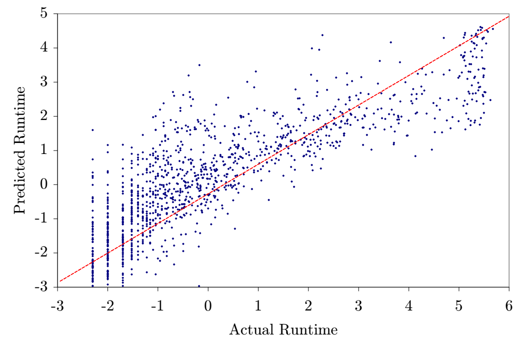


Figure 3.9: Linear Regression: Prediction Scatterplot (test data, Variable Size).

cases when the amount 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 entirely correct for overfitting; nevertheless, it can be an informative measure when presented along with test set error. Figure 3.6 shows the cumulative distribution of squared errors on the test set for the 1000 bids, 256 goods dataset. 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 3.7 shows a scatterplot of predicted $\log_{10}$ runtime vs. actual $\log_{10}$ runtime. Figures 3.8 and 3.9 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, these results show that our linear models would be able to do a good job of classifying instances into the bins shown in Figures 3.4 and 3.5, despite the fact that they are not given the distribution from which each instance was drawn: 93% of the time the log running times 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). On the variable-size data, 62% of instances were predicted correctly to an order of magnitude.

Data point	**Mean Abs. Err.**	**RMSE**	R^2
1000 Bids/256 Goods	0.2043	0.3176	0.9849
1000 Bids/144 Goods	0.2471	0.3678	0.9725
2000 Bids/64 Goods	0.1974	0.3352	0.9652
Variable Size	0.4222	0.7245	0.9448

Table 3.2: Quadratic Regression: Errors and Adjusted R^2.

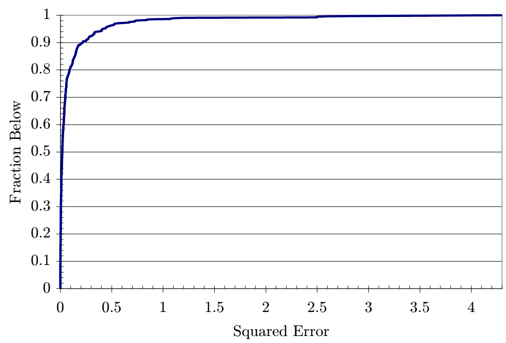


Figure 3.10: Quadratic Regression: Squared Error (test data, 1000 Bids/256 Goods).

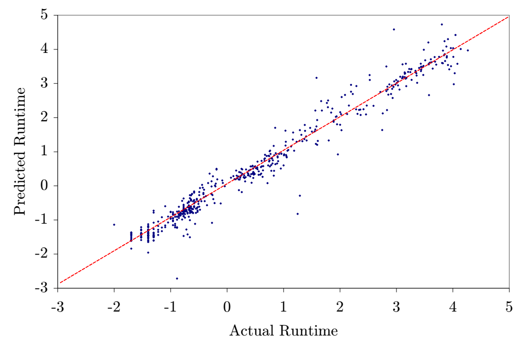


Figure 3.11: Quadratic Regression: Prediction Scatterplot (test data, 1000 Bids/256 Goods).

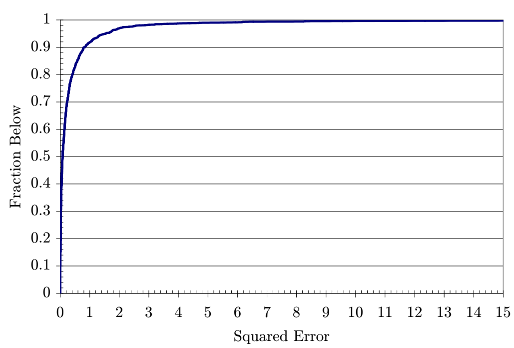


Figure 3.12: Quadratic Regression: Squared Error (test data, Variable Size).

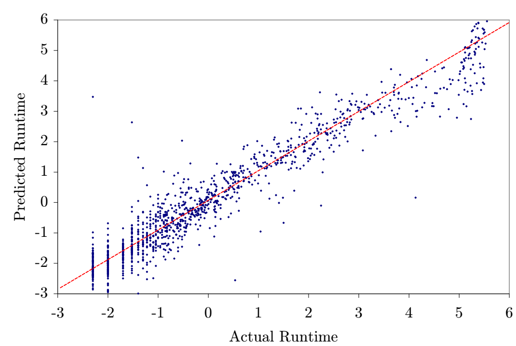


Figure 3.13: Quadratic Regression: Prediction Scatterplot (test data, Variable Size).

3.5.3 Nonlinear Models

We tried quadratic regression as a more sophisticated machine learning technique[5]. In order to construct these models, we first computed all pairwise products of features, including squares, as well as retained the original features, for a total of 434 features in the fixed-size datasets and 495 features in the variable-sized one. However, this introduced a lot of redundant features, as even the original features had a lot of correlation. In order to improve numerical stability and accuracy, we performed a preprocessing step that allowed us to get rid of some of these useless features. For this we employed the following iterative process. At each step for each feature we constructed a regression model that predicted its value using the remaining features. For some features this resulted in an almost perfect prediction accuracy on the training set, indicating that they are redundant. We used R^2 as a measure of this prediction quality. At each step we dropped the feature that had the highest R^2 and then repeated the whole process. We stopped when all of the R^2 values were below a conservative threshold of 0.99999. Even with such a stopping criterion, this eliminated a lot of features. On the 1000 bids, 256 goods dataset we were left with 312 features out of 434, and on the variable-size dataset with 436 out of 495. In our experience, this preprocessing step also resulted in better and more stable models.

For all of our datasets quadratic models gave considerably better error measurements on the test set and also explained nearly all the variance in the training set, as shown in Table 3.2. As above, Figures 3.10, 3.11, 3.12, and 3.13 show cumulative distributions of the squared error and scatterplots of predicted $\log_{10}$ runtime vs. actual $\log_{10}$ runtime for fixed-size and variable-size data. Comparing these figures to Figures 3.6, 3.7, 3.8, and 3.9 confirms our judgment that quadratic models are substantially better overall. The cumulative distribution curves clearly lie well above the corresponding curves for linear models. In particular, quadratic models would

[5]We also explored another nonlinear regression technique, 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. The RMSE on our MARS models differed from the RMSE on our second-order model only in the second decimal place; as MARS models can be unstable and difficult to interpret, we focus on our second-order model.

classify 98% of test instances in the fixed-size dataset and 92% of instances in the variable-sized dataset correctly to within an order of magnitude.

3.6 Analyzing the WDP Hardness Models

The results summarized above demonstrate that it is possible to learn a model of our features that accurately predicts the logarithm of 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. In this section we set out to interpret our models according to the process described in Section 2.3.

We used three subset selection techniques (forward and backward selection, and sequential replacement) to obtain good subsets with given numbers of features. We plotted subset size (from 1 to the total number of variables) versus the RMSE of the best model built from a subset of each size. We then chose the smallest subset size at which there was little incremental benefit gained by moving to the next larger subset size. We examined the features in the model, and also measured each variable's cost of omission — the (normalized) difference between the RMSE of the model on the original subset and a model omitting the given variable.

Despite the fact that our quadratic models strongly outperformed the linear models, we analyze both sets of models here. The reason for this is that we can learn different things from the different models; for example, the linear models can sometimes give rise to simpler intuitions since they do not depend on products of features.

Linear Models

Figure 3.14 shows the RMSE of the best subset containing between 1 and 28 features for linear models on the 1000 bids, 256 goods dataset. Due to our use of heuristic subset selection techniques, the subsets shown in Figure 3.14 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

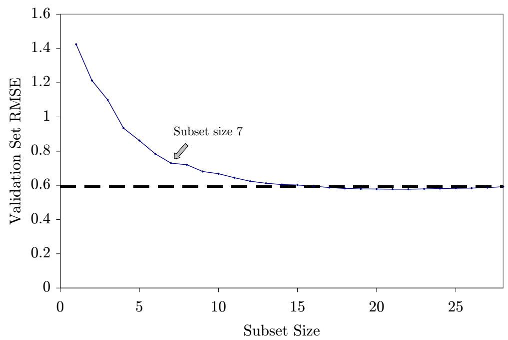


Figure 3.14: Linear Regression: Subset Size vs. RMSE (1000 bids/256 Goods).

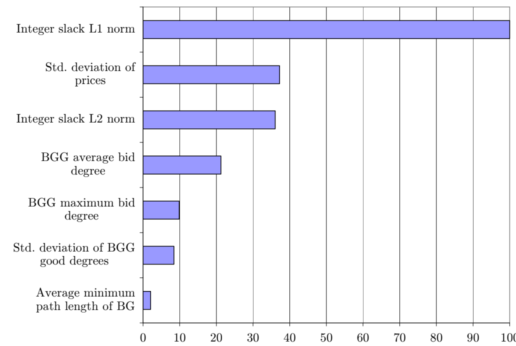


Figure 3.15: Linear Regression: Cost of Omission for Subset Size 7 (1000 Bids/256 Goods).

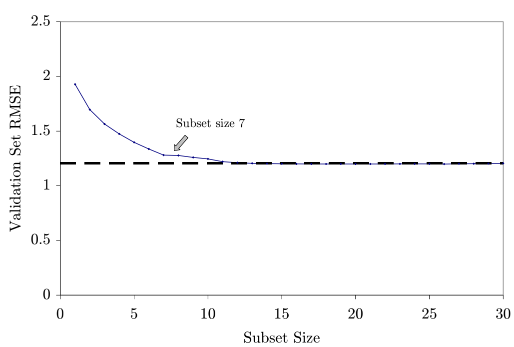


Figure 3.16: Linear Regression: Subset Size vs. RMSE (Variable Size).

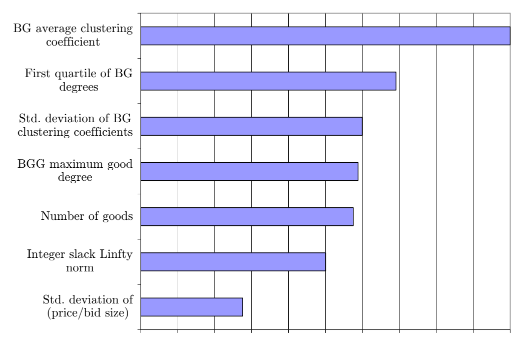


Figure 3.17: Linear Regression: Cost of Omission for Subset Size 7 (Variable Size).

the model with seven 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 are relatively highly correlated with these seven. Figure 3.15 shows the features in this model and their respective costs of omission (scaled to 100).

First, we must note that this set of features is rather different from the one described in [Leyton-Brown et al. 2002]. This is due to the fact that in current work we have employed a slightly different machine learning pipeline: we've added a couple of extra features and used a small ridging parameter in regression. Also, because we have switched our regression software, we were unable to run the exhaustive feature selection algorithm. This observed difference stresses the important point discussed in Section 2.3 of Chapter 2: good subsets of each size identified by our techniques need not be unique. Nevertheless, by looking closer at the kinds of features that comprise these subsets, we can clearly discern some common themes.

The most overarching conclusion we can draw from this data and our previous observations is that structural features and features based on the LP relaxation are the most important ones. The most important feature in the model analyzed in Figure 3.15 appears to be the ℓ_1 norm of the linear programming slack vector. This is very intuitive; CPLEX employs LP relaxation as its guiding heuristic for the branch-and-cut search. Thus, integrality gap represents the quality of the heuristic and, consequently, the hardness of the problem. The closely related ℓ_2 norm also appears in the model. The second most important feature deals with actual weights in our optimization problem; it looks at the spread of different prices. Intuitively, if bid prices are close to each other it must be harder to choose one bid over the other, and so optimization depends much more on the rest of the structure of the problem instance. The remaining four features in this model all deal with this structure. The average and the maximum number of goods contained in each bid, and the standard deviation of the number of bids in which each good participates in some sense measure the local structure of the bid-good graph. The average minimum path length in the bid graph, on the other hand, is a more global measure of the constraint structure. Qualitatively, this mix of LP relaxation features and both global and local structural features is consistently present in every similarly-sized subset that our techniques have identified as being important.

Figures 3.16 and 3.17 demonstrate the linear model subset selection for our variable-size dataset. For this dataset, we also picked a seven-variable model. Notice that while the actual feature names are different, we can observe the same trends as we have observed in the fixed-size case. We still see an LP-based feature, though it has now been replaced with the ℓ_∞ norm. We also see a feature that measures actual prices, now normalized to better account for varying bid sizes. Clustering coefficient is a measure of average cliquiness of BG; this feature gives an indication of how local the problem's constraints are. We also see structural features related to the number of conflicts in the BG as well the maximum number of bids in which a good participates.

Since it is well-known that problem hardness depends on the input size, it is not surprising to see size features appear in the model. The only size feature that made it into the seven-variable model is the number of goods. This is not to be unexpected, since it is actually known that complexity of the WDP can scale exponentially in the number of goods, but only polynomially in the number of bids [Sandholm 2002]. While the number of bids definitely still has an effect on hardness, it appears much smaller than the effect of problem structure[6]. We also note that these observations indirectly validate our techniques: we were able to automatically identify the number of goods as being a more fundamental factor than the number of bids in determining problem hardness.

Nonlinear Models

We now consider second-order models. Figure 3.18 describes the best subsets containing between 1 and 80 features for second-order models on the fixed-size dataset. We observe that allowing interactions between features dramatically improved the

[6]When studying the effect of input size on runtime, we have attempted to conduct scaling experiments, trying to increase problem size as much as possible. We have 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 doesn't strongly manifest itself till size is increased well beyond what we would consider reasonable or feasible. In particular, this lead us to treat problem size features as being fundamentally the same as structural features, rather than come up with specialized hypotheses that treat input size separately.

accuracy of our very small-subset models; indeed, our 4-feature quadratic model outperformed the full linear model.

Figure 3.19 shows the costs of omission for the variables from the best eight-feature subset. We first note that many second-order features were selected, with only two being first-order. As in the case of our linear model, we observe that the most critical features are structural. The clustering coefficient and node degrees are very prominent; we also see a more global measure of constrainedness — average node eccentricity of the BG. Interestingly, it appears that most important is the interaction between LP relaxation features and features related to the BG node degrees.

Figures 3.20 and 3.21 show a similar picture for our variable-size dataset. On this dataset a two-feature quadratic model slightly outperformed the full linear model. These two features were the product of the number of bids with the ℓ_1 norm of the LP slack vector and the product of the average node eccentricity of the BG with the standard deviation of the clustering coefficients of nodes in the BG.

For a more informative comparison, we selected a larger nine-feature model (Figure 3.21). The features present in this model exhibit many similarities to those in the fixed-size dataset model and the linear model. We still see LP-based features, global structural features such as BG radius and average minimum path lengths, as well as more local node degree statistics. We also see some marked differences from the linear model. Most importantly, the number of bids, once allowed to interact with other features, becomes much more prominent. One possible explanation for this is that the number of bids provides a unifying scaling factor for many features.

Discussion

We can look at the features that were important to our quadratic and linear models in order to gain understanding about how these models work. The importance of the LP relaxation norms is quite intuitive: 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. Various BG node degree features describe the overall constrainedness of the problem. Generally, we would expect that very

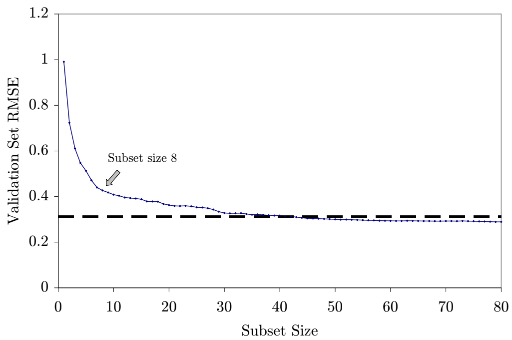


Figure 3.18: Quadratic Regression: Subset size vs. RMSE (1000 Bids/256 Goods).

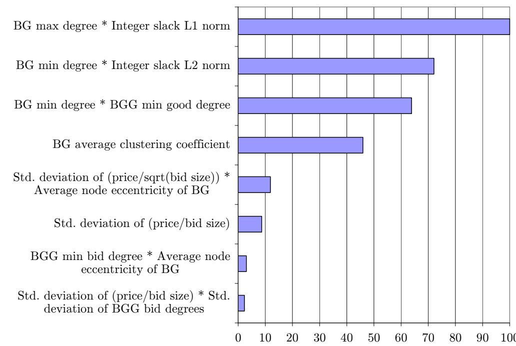


Figure 3.19: Quadratic Regression: Cost of Omission for Subset Size 8 (1000 Bids/256 Goods).

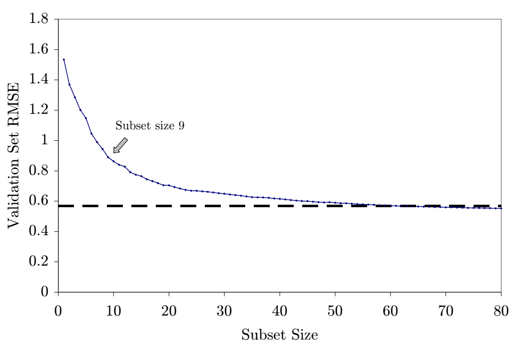


Figure 3.20: Quadratic Regression: Subset Size vs. RMSE (Variable Size).

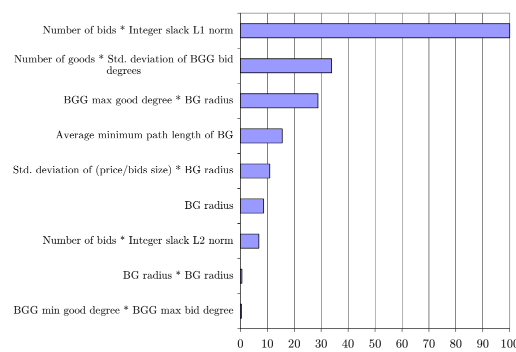


Figure 3.21: Quadratic Regression: Cost of Omission for Subset Size 9 (Variable Size).

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. It seems 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 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 max, min, average and standard deviation of the number of constraints in which each variable is involved; they indicate how quickly the search space can be expected to narrow as variables are given values (*i.e.,* as bids are assigned to or excluded from the allocation). Similarly, 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. Thus, we can now understand the role of the most important feature in our eight-feature fixed-size quadratic model: the product of the maximum BG node degree and the integer slack ℓ_1 norm. Note that this feature takes a large value only when both components are large; the explanations above show that problems are easy for CPLEX whenever either of these features has a small value. Since both features are relatively uncorrelated on our data, their product gives a powerful prediction of an instance's hardness. Similar intuitions clearly also apply to the variable-sized models.

3.7 Applications of the WDP Hardness Models

The results in previous sections demonstrated that it is indeed possible to construct accurate models of runtime according to the methodology of Chapter 2. We now to the question of validating applications of these models, as described in Section 2.4.

3.7.1 Algorithm Portfolios

In this section, we consider portfolio performance on two datasets: the fixed size instances with 1000 bids and 256 goods, and our variable size dataset. On the fixed

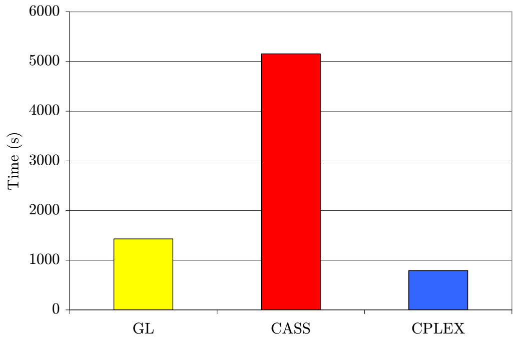


Figure 3.22: Algorithm Runtimes (1000 Bids/256 Goods).

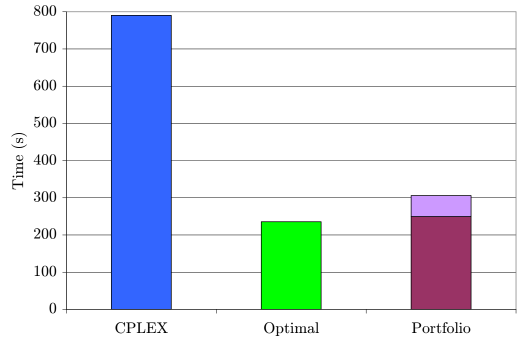


Figure 3.23: Portfolio Runtimes (1000 Bids/256 Goods).

size dataset we used CPLEX 7.1, GL, and CASS (see Section 3.2.1 for their descriptions). For the variable-size dataset we only considered CPLEX 8.0 and CASS, since it turned out that marginal utility of adding GL to a portfolio is low.

First, we used the methodology from Section 2.2 to build regression models for GL and CASS. Figures 3.22, 3.23, compare the average runtimes of our three algorithms (CPLEX, CASS, GL) to that of the portfolio on our fixed-size dataset.[7]. These averages were computed over instances on which *at least one* of the algorithm didn't time out, and thus include some cap times. Therefore, the bars in reality represent *lower bounds* on average runtimes on these datasets for constituent algorithms.

CPLEX would clearly be chosen under winner-take-all algorithm selection on both datasets. 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 are much slower than CPLEX on average, the portfolio outperforms CPLEX by roughly a factor of 3. Moreover, neglecting the cost of computing features, our portfolio's selections on average take only 8% longer to run than the optimal selections.

Figures 3.24 and 3.25 show the frequency with which each algorithm is selected in the ideal portfolio and in our portfolio on the fixed-size dataset. They illustrate the

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

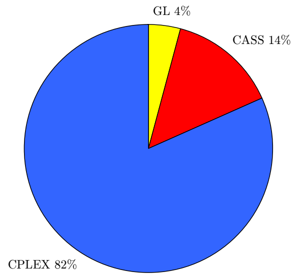


Figure 3.24: Optimal Selection (1000 Bids/256 Goods).

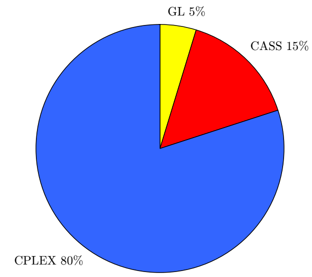


Figure 3.25: Portfolio Selection (1000 Bids/256 Goods).

quality of our algorithm selection and the relative value of the three algorithms. It turns out that CASS is often significantly uncorrelated with CPLEX, and that most of the speedup in our portfolio comes from choosing CASS on appropriate instances. Observe that the portfolio does not always make the right choice (in particular, it selects GL and CASS slightly more often than it should). However, most of the mistakes made by our models occur when both algorithms have very similar running times; these mistakes are not very costly, explaining why our portfolio's choices have a running time so close to the optimal. It thus performs very well according to all of the metrics mention in Section 2.4.2. This highlights an important point about our portfolio methodology: algorithm selection is an easier problem than accurate runtime prediction, since it's relatively easy to discriminate among algorithms when their runtimes differ greatly, and accuracy is less important when their runtimes are similar.

Figures 3.26, 3.27, 3.28, and 3.29 show portfolio performance on the variable-size dataset. At first glance, the average gain of using a portfolio appear less dramatic here. This is partially caused by the fact that CPLEX is able to solve significantly harder instances than CASS, and thus the average runtime for the portfolio tracks CPLEX's runtime much closer than that of CASS. However, as Figure 3.28 demonstrates, it is still the case that CASS is faster on roughly a quarter of the instances, and the portfolio often correctly selects CASS instead of CPLEX. When one is concerned with

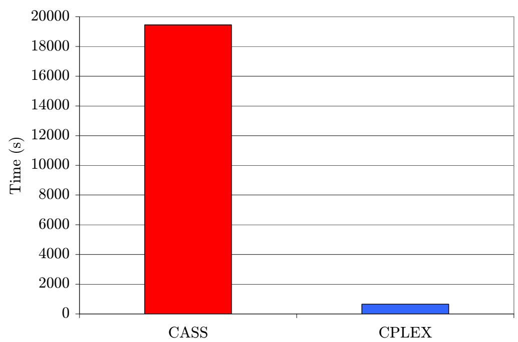


Figure 3.26: Algorithm Runtimes (Variable Size).

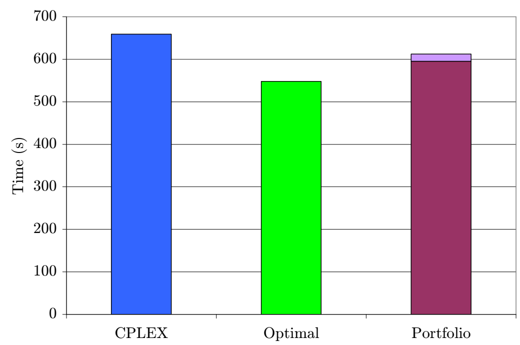


Figure 3.27: Portfolio Runtimes (Variable Size).

solving a *particular* instance at hand, the average performance becomes a meaningless metric, and employing a portfolio still makes sense.

Observe that our variable importance analysis from Section 3.6 gives us some insight about why an algorithm like CASS is 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 bid graph 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 feasible search space size. Like many search algorithms, CASS often benefits whenever 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'. We suspect that our techniques could be even more effective in other settings.

We have also experimentally validated our methodology for smart feature computation. Figure 3.30 shows the performance of the smart feature computation discussed in Section 2.4.2, with the upper part of the bar indicating the time spent computing features. Compared to computing all features, we reduce overhead by almost half

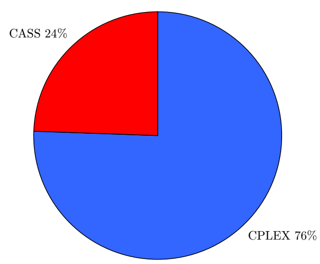


Figure 3.28: Optimal Selection (Variable Size).

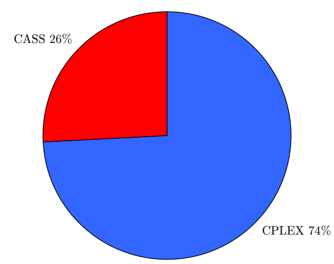


Figure 3.29: Portfolio Selection (Variable Size).

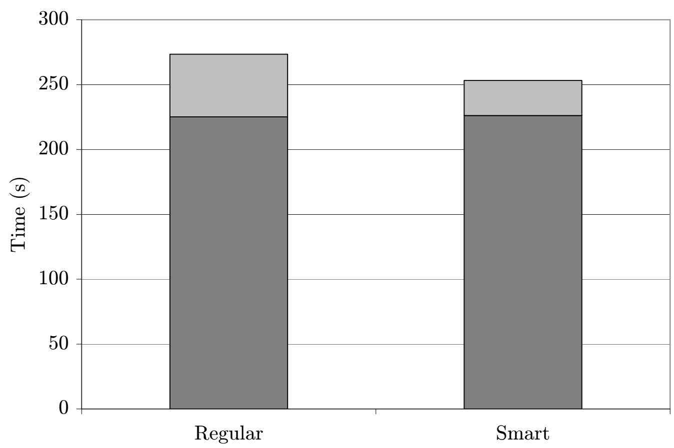


Figure 3.30: Smart Features.

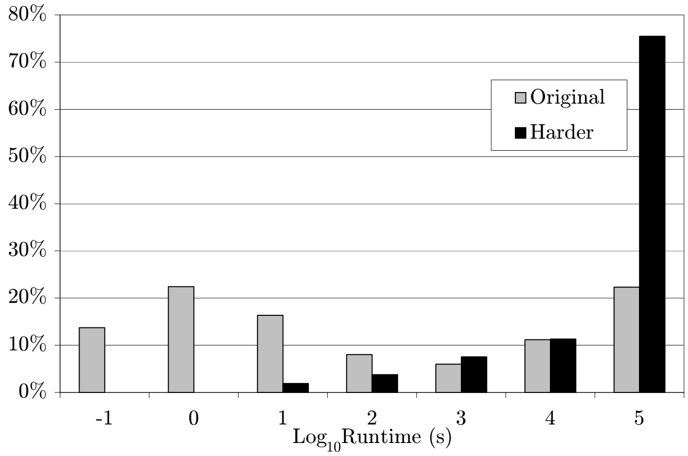


Figure 3.31: Inducing Harder Distributions.

with nearly no cost in running time[8].

3.7.2 Inducing Hard Distributions

Now that we have demonstrated that algorithms portfolios improve performance of WDP algorithms, we turn to the question of testing new WDP algorithms, as described in Section 2.4.3.

Due to the wide spread of runtimes in our composite distribution D (7 orders of magnitude) and the high accuracy of our model h_f, it is quite easy for our technique to generate harder instances. We present the histograms of the optimal portfolio runtimes on the 1000 bids, 256 goods dataset for the original and harder distribution in Figure 3.31[9]. Because our runtime data was capped, there is no way to know if the hardest instances in the new distribution are harder than the hardest instances in

[8]This figure was obtained using models from [Leyton-Brown et al. 2002], which are slightly different from the models used for the rest of the results in this chapter. The performance of those models is virtually identical to that of the new ones.

[9]Because generating figures in this section requires collecting runtimes on newly generated instances, these results are based on models used in [Leyton-Brown et al. 2002]. As mentioned previously, the performance of those models is very similar to that of the newer ones. We also note that while in Section 2.4.3 we suggested that for a composite distribution such as ours it is possible to learn hardness models individually, for these results H_f was trained on the whole dataset. Learning new models would probably yield even better results.

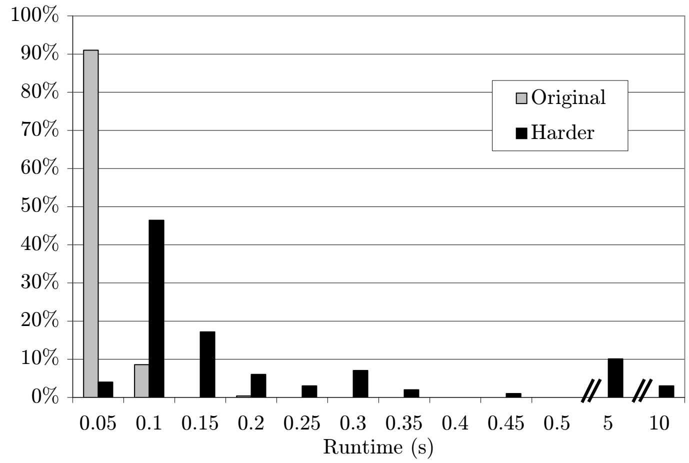


Figure 3.32: Matching.

the original distribution; note, however, that very few easy instances are generated. Instances in the induced distribution came predominantly from the CATS *arbitrary* distribution, with most of the rest from L3.

To demonstrate that our technique also works in more challenging settings, we sought a different distribution with small runtime variance. As described in Section 3.5.1, there has been ongoing discussion in the WDP literature about whether those CATS distributions that are relatively easy could be configured to be harder. We consider two easy distributions with low variance from CATS, *matching* and *scheduling*, and show that they indeed can be made much harder than originally proposed. Figures 3.32 and 3.33 show the histograms of the runtimes of the ideal portfolio on the 1000 bids, 256 goods dataset 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.

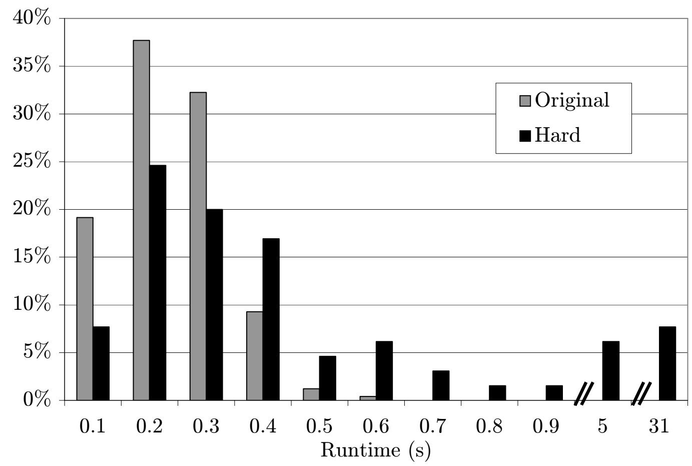


Figure 3.33: Scheduling.

3.8 Conclusion

In this chapter, we performed an extensive experimental investigation into the empirical hardness of the combinatorial auctions winner determination problem. We identified structural, distribution-independent features of WDP instances and showed that, somewhat surprisingly, they contain enough information to predict CPLEX's running time with high accuracy.

We demonstrated that a portfolio composed of CPLEX, CASS and GL outperformed CPLEX alone by a factor of 3 — despite the fact that CASS and GL are *much* slower than CPLEX on average. We were also able to induce test data that was much harder for our portfolio, and were even able to make specific CATS distributions much harder.

Perhaps more importantly, results in this chapter experimentally validated methodologies proposed in Chapter 2.

Chapter 4

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

In this chapter we describe and analyze empirical hardness models for three SAT solvers — `kcnfs`, `oksolver` and `satz` — and for two different distributions of instances: uniform random 3-SAT with varying ratio of clauses-to-variables, and uniform random 3-SAT with fixed ratio of clauses-to-variables. Furthermore, we present some interesting findings on which features are most useful in predicting whether an instance will be hard to solve.

4.1 Introduction

SAT is among the most studied problems in computer science, representing a generic constraint satisfaction problem with binary variables and arbitrary constraints. It is also the prototypical $\mathcal{NP}$-hard problem, and its worst-case complexity has received much attention. Accordingly, it is not surprising that SAT has become a primary platform for the investigation of average-case and empirical complexity. Particular interest has been paid to randomly generated SAT instances. In this chapter we concentrate on this distribution as it offers both a range of very easy to very hard instances for any given input size and the opportunity to make connections to a wealth of existing work. In fact, this wealth of work on typical-case complexity of SAT inspired our initial development of the empirical hardness methodology (Chapter 2), with this chapter completing the circle.

This chapter has three goals. First, we aim to validate the methodology presented in Chapter 2 in yet another problem domain. We consider three different SAT algorithms (`kcnfs`, `oksolver`, and `satz`– three state-of-the-art solvers for random instances) and two different instance distributions. The first distribution contains random 3-SAT instances with a varying ratio of clauses to variables, allowing us to see whether our techniques automatically select the clauses-to-variables ratio as an important feature, and also what other features are important in this setting. Our second distribution contains random 3-SAT instances with the ratio of clauses-to-variables held constant at the phase transition point. This distribution has received much attention in the past; it gives us the opportunity to explain the orders-of-magnitude runtime variation that persists in this so-called "hard region."

Second, we show that empirical hardness models have other useful applications for SAT. Most importantly, we describe a SAT solver, `SATzilla`, which uses hardness models to choose among existing SAT solvers on a per-instance basis. We explain some details of its construction and summarize its performance in Section 4.5.

Our final, and, perhaps, most important, goal is to offer concrete examples in support of our abstract claim that empirical hardness models are a useful tool for gaining understanding of the behavior of algorithms for solving $\mathcal{NP}$-hard problems.

4.2 The Propositional Satisfiability Problem

Propositional Satisfiability (SAT) [Garey and Johnson 1979] is arguably the most studied problem in computer science. It is the first problem that was proven to be $\mathcal{NP}$-complete [Cook 1971].Besides its considerable theoretical importance, it also has numerous practical applications. SAT is used in areas from program verification to planning and scheduling. SAT being $\mathcal{NP}$-complete also means that efficient algorithms for SAT can translate directly into efficient algorithms for a wide variety of other problems in $\mathcal{NP}$.

Formally, let $X = \{x_1, \ldots, x_v\}$ be a set of v variables. This set induces a set of $2v$ *literals* $L = \{x_1, \neg x_1, \ldots, x_v, \neg x_v\}$. A SAT *instance* consists of a set of c clauses, where each clause $C_i \subseteq L$. Semantically, this defines a propositional formula in conjunctive normal form: $F(X) = \bigwedge_{i=1}^{c} \bigvee_{l \in C_i} l$.

A *truth assignment* τ assigns a truth value (t or f) to each variable in X: $\tau : X \rightarrow \{t, f\}$. We say that τ *satisfies* F ($\tau \models F$) if F evaluates to true according to standard logical rules when each variable is assigned a value according to τ.

Problem 4.1 *The* SAT *problem is the following: given a formula F represented by a set of clauses $C_1, \ldots, C_c$ over a set of variables $x_1, \ldots, x_v$ return "yes" if there exists a truth assignment τ such that $\tau \models F$, and return "no" otherwise.*

By a k-SAT problem we mean a problem where $|C_i| = k$ for all i. It is well-known that the problem remains $\mathcal{NP}$-complete for all $k > 2$. In this chapter we will be mostly concerned with 3-SAT instances.

In order to perform empirical studies, we must define a *distribution* over instances. The most widely studied distribution is called *uniformly random k-SAT*. An instance with c clauses and v variables is generated as follows. For each clause C_i choose exactly k variables (out of v total) uniformly at random, and then assign a positive or negative sign to each variable with probability 1/2. Clauses that are trivial (*i.e.,* contain a literal and its negation) are usually discarded.

The reason why random k-SAT is so prominent in CS research is the discovery of the phase transition in solvability and its apparent correlation with problem hardness. See Section 2.5.1 of Chapter 2 for the overview of relevant literature.

4.2.1 SAT Algorithms

While the exact details of the algorithms are unimportant for the purposes of this chapter, we briefly remind of the methods used to solve SAT instances.

Most of the current state-of-the-art SAT algorithms (with very few notable exceptions) can be divided into two classes. The first class of algorithms is based on the classical Davis-Putnam-Logemann-Loveland (DPLL) procedure [Davis and Putnam 1960; Davis et al. 1962]. This algorithm performs a depth-first search in the space of all possible truth assignments to variables in X. Thus, in the worst case, the search tree is a complete binary tree of depth v. However, this is implemented so that once a variable x_i and it's value $\tau(x_i)$ (t or f) are chosen, the set of clauses is *simplified* so that it contains no mention of x_i. This means that all clauses that are made true by $\tau(x_i)$ are marked as inactive, while x_i literals are removed from clauses where they evaluate to false. If all clauses have been thus eliminated, then a satisfying assignment has been found. Conversely, if a clause becomes empty without being eliminated, then a contradiction has been reached, and the search backtracks to the previous untried value assignment. All DPLL algorithms augment this basic procedure with two heuristics that greatly improve their performance. The first one is the *pure-literal* heuristic: if there is some variable x_i which occurs *only* as a positive literal, or *only* as a negative literal, then it must be instantiated so that it evaluates to true in all occurrences. The second heuristic is called *unit propagation*: if there is a *unit* clause $C_i = \{l\}$ (*i.e.,* a clause of length one), then l must be assigned to be true. This process is repeated until there are no more unit clauses left, or a contradiction is reached.

The above describes the basic DPLL procedure. Modern DPLL-based solvers differ in many additional heuristics that they introduce. For the most part these heuristics decide which variable to assign next, and which value to try first. Some also employ more complex techniques, such as clause learning. These algorithms are *complete* — guaranteed to find a satisfying assignment if one exists, or return "no" otherwise.

In this chapter we will study the behavior of DPLL-based procedures. However, it is useful to review the second major class of SAT algorithms — stochastic local search (SLS) procedures [Hoos and Stützle 2004]. While DPLL procedures look at *partial* truth assignments, SLS methods search in the space of *complete* truth assignments. Each SLS method includes a metric that evaluates the quality of the current assignment. This is often simply the number of unsatisfied clauses, though modern algorithms often use a weighted version with the weight of each clause changing during the run of the algorithm. As a rule, they start with a random assignment and then perform a greedy descent (sometimes including random moves) according to their metric; the move in this space is usually defined as a flip of the truth assignment to a single variable. SLS algorithms are inherently *incomplete* — sometimes they may wander forever without finding a solution, and they can never prove that an instance is unsatisfiable. Nevertheless, in practice, on satisfiable instances, they usually outperform complete algorithms by orders of magnitude both in the running time, and in the size of the instances that can be solved.

4.3 Describing SAT Instances with Features

Figure 4.1 summarizes the 91 features used by our SAT models. Since not every feature is useful in every distribution, we discarded uninformative or highly correlated features after fixing the distribution. For example, while ratio of clauses-to-variables was important for `SATzilla`, it is not at all useful for the fixed-ratio dataset. In order to keep values to sensible ranges, whenever it makes sense we normalize features by either the number of clauses or the number of variables in the formula.

The features can be divided into nine groups. The first group captures problem size, measured by the number of clauses, variables, and the ratio of the two. Because we expect this ratio to be an important feature, we gave it additional expressive power by including squares and cubes of both the ratio and its reciprocal. Also, because we know that features are more powerful in simple regression models when they are directly correlated with the response variable, we include a "linearized" version of the ratio which is defined as the absolute value of the difference between the ratio and the phase transition point, $c/v = 4.26$. It turns out that for variable-ratio data this group of features alone suffices to construct reasonably good models. However, including the rest of our features significantly improves these models. Moreover, in the presence of other features, including higher-order features 4, 5, 7, 8, 10 and 11 does not improve accuracy much and does not qualitatively change the results reported below. Thus, for the rest of this chapter we focus on models that use all of the ratio features.

The next three groups correspond to three different graph representations of a

Problem Size Features:

1. **Number of clauses:** denoted c
2. **Number of variables:** denoted v

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

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

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

Variable-Clause Graph Features:

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

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

Variable Graph Features:

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

Clause Graph Features:

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

33-35. **Weighted clustering coefficient statistics**: mean, variation coefficient, min, max, and entropy.

Balance Features:

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

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

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

Proximity to Horn Formula

49. **Fraction of Horn clauses**

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

LP-Based Features:

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

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

DPLL Search Space:

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

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

Local Search Probes:

68-71. **Minimum fraction of unsat clauses in a run:** mean and variation coefficient for SAPS and GSAT (see [Tompkins and Hoos 2004]).

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

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

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

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

Figure 4.1: SAT Instance Features.

SAT instance. The variable-clause graph (VCG) is a bipartite graph with a node for each variable, a node for each clause, and an edge between them whenever a variable occurs in a clause. The variable graph (VG) has a node for each variable and an edge between variables that occur together in at least one clause. The clause graph (CG) has nodes representing clauses and an edge between two clauses whenever they share a *negated* literal. Each of these graphs corresponds to a constraint graph for the associated CSP; thus, each encodes aspects of the problem's combinatorial structure. For each graph we compute various node degree statistics[1]. For the CG we also compute statistics of weighted clustering coefficients, which measure the extent to which each node belongs to a clique. For each node the *weighted clustering coefficient* is the number of edges among its neighbors (including the node itself) divided by $k(k+1)/2$, where k is the number of neighbors. Including the node when counting edges has an effect of weighting the classical clustering coefficient by the node degree.

The fifth group measures the balance of a formula in several different (syntactical) senses. Here we compute the number of unary, binary, and ternary clauses; statistics of the number of positive vs. negative occurrences of variables within clauses and per variable. The sixth group measures the proximity of the instance to a Horn formula, motivated by the fact that such formulae are an important SAT subclass solvable in polynomial time.

The seventh group of features is obtained by solving a linear programming relaxation of an integer program representing the current SAT instance (in fact, on occasion this relaxation is able to solve the SAT instance!). Let x_j denote both Boolean and LP variables. Define $v(x_j) = x_j$ and $v(\neg x_j) = 1 - x_j$. Then the program is

$$\begin{array}{lll} \textit{maximize:} & \displaystyle\sum_{i=1}^{c} \sum_{l \in C_i} v(l) & \\ \textit{subject to:} & \displaystyle\sum_{l \in C_i} v(l) \geq 1 & \forall C_i \\ & 0 \leq x_j \leq 1 & \forall x_j \end{array}$$

The objective function prevents the trivial solution where all variables are set to 0.5.

[1]The variation coefficient is the ratio of the standard deviation and the mean.

The eighth group involves running DPLL "probes." First, we run a DPLL procedure to an exponentially-increasing sequence of depths, measuring the number of unit propagations done at each depth. We also run depth-first random probes by repeatedly instantiating random variables and performing unit propagation until a contradiction is found. The average depth at which a contradiction occurs is an unbiased estimate of the log size of the search space [Lobjois and Lemaître 1998].

Our final group of features probes the search space with two stochastic local search algorithms, GSAT and SAPS [Hoos and Stützle 2004]. We run both algorithms many times, each time continuing the search trajectory until a plateau cannot be escaped within a given number of steps. We then average statistics collected during each run.

4.4 Empirical Hardness Models for SAT

The first dataset used in this chapter contained 20 000 uniform random 3-SAT instances with 400 variables each. To determine the number of clauses in each instance, we first chose the clauses-to-variables ratio by drawing a uniform sample from $[3.26, 5.26]$ (*i.e.,* the number of clauses varied between 1 304 and 2 104). This range was chosen symmetrically around the phase transition point, 4.26, to ensure that an approximately equal number of satisfiable and unsatisfiable instances would be obtained. The second dataset contained 20 000 uniform random 3-SAT instances with 400 variables and 1 704 clauses each, corresponding to a fixed clauses-to-variables ratio of 4.26. On each dataset we ran three solvers — `kcnfs`, `oksolver` and `satz`— which performed well on random instances in previous years' SAT competitions. Our experiments were executed on 2.4 GHz Xeon processors, under Linux 2.4.20. Our fixed-ratio experiments took about four CPU-months to complete. In contrast, our variable-ratio dataset took only about one CPU-month, since many instances were generated in the easy region away from the phase transition point. Every solver was allowed to run to completion on every instance.

Each dataset was split into 3 parts — training, test, and validation sets — in the ratio 70 : 15 : 15. All parameter tuning was performed with the validation set; the test set was used only to generate the graphs shown in this chapter. Machine learning and statistical analysis performed for this chapter were done with the `R` and `Matlab` software packages.

4.4.1 Variable-Ratio Random Instances

We had three goals with this distribution. First, we wanted to show that our empirical hardness model training and analysis techniques would be able to sift through all the features provided and "discover" that the clauses-to-variables ratio was important to the empirical hardness of instances from this distribution. Second, having included nine features derived from this ratio among our 91 features — the clauses-to-variables ratio itself, the square of the ratio, the cube of the ratio, its reciprocal (*i.e.,* the variables-to-clauses ratio), the square and cube of this reciprocal, the absolute value minus 4.26, and the square and cube of this absolute value — we wanted to find out which particular function of these features would be most predictive of hardness. Third, we wanted to find out what *other* features, if any, were important in this setting.

We begin by examining the clauses-to-variables ratio, c/v, in more detail. Figure 4.2 shows `kcnfs` runtime (log scale) vs. c/v for satisfiable and unsatisfiable instances. First observe that, as expected, there is a clear relationship between runtime and c/v. At the same time, c/v is not a very accurate predictor of hardness by itself: particularly near the phase transition point, there are several orders of magnitude of runtime variance across different instances. This is particularly the case for satisfiable instances around the phase transition; while the variation in runtime between unsatisfiable instances is consistently much smaller. (This is not surprising, and been studied in more detail in, *e.g.,* [Gomes et al. 2004]). It may be noted that overall our dataset is balanced in that it consists of 10 011 satisfiable and 9 989 unsatisfiable instances. Another thing to note is that although most instances left of the phase transition point are satisfiable and vice versa, this relationship is not strict.

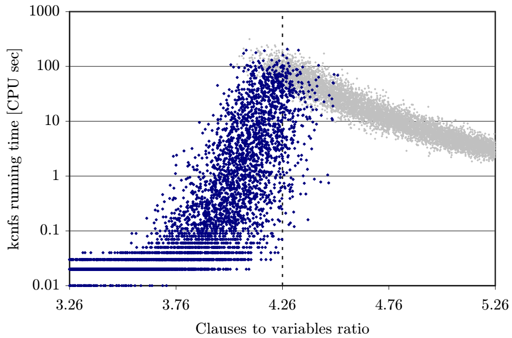
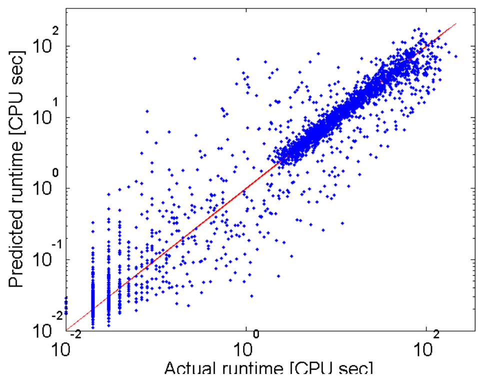


Figure 4.2: Runtime of `kcnfs` on Variable-Ratio Satisfiable (left) and Unsatisfiable Instances (right).

Model Performance

To build models, we first considered linear, logistic and exponential models in our 91 features, evaluating the models on our validation set. Of these, linear were the worst, and logistic and exponential were similar, with logistic being slightly better. Next, we wanted to consider quadratic models under these same three transformations. However, a full quadratic model would have involved 4 277 features, and given that our training data involved 14 000 different problem instances, training the model would have entailed inverting a matrix of nearly sixty million values. In order to concentrate on the most important quadratic features, we first used our variable importance techniques to identify the best 30-feature subset of our 91 features. We computed the full quadratic expansion of these features, then performed forward selection — the only subset selection technique that worked with such a huge number of features — to keep only the most useful features. We ended up with 360 features, some of which were members of our original set of 91 features and the rest of which were products of these original features. Again, we evaluated linear, logistic and exponential models; all three model types were better with the expanded features, and again logistic models were best. Although the actual RMSE values obtained by three different kinds of models were very close to each other, linear models tended to have much higher prediction bias and many more outliers, especially among easy instances.

Figure 4.3 (left) shows the performance of our logistic models in this quadratic



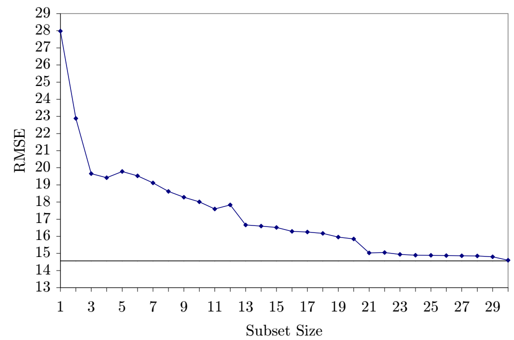


Figure 4.3: Actual vs. Predicted Runtimes for `kcnfs` on Variable-Ratio Instances (left) and RMSE as a Function of Model Size (right).

case for `kcnfs` (evaluated for the first time on our test set). Note that this is a very accurate model: perfect predictions would lie exactly on the line $y = x$, and the vast majority of points lie on or very close to this line, with no significant bias in the residuals.[2] The plots for `satz` and `oksolver` look very similar; the RMSE values for the `kcnfs`, `satz` and `oksolver` models are 13.16, 24.09, and 81.32 seconds, respectively. We note, that the higher error exhibited by `oksolver` models may be at least partially due to considerably higher average runtimes of `oksolver` compared to those of other algorithms.

Analyzing Variable-Ratio Models

We now turn to the question of which variables were most important to our models. For the remainder of this chapter we focus only on our models for `kcnfs`. We choose to focus on this algorithm because it is currently the state-of-the-art random solver; our results with the other two algorithms are comparable.

In order to analyze the models, we once again employed techniques described in Section 2.3 of Chapter 2. Recall, that these techniques are only able to indicate which variables are *sufficient* to approximate the performance of the full model, but we must be very careful in drawing conclusions about the variables that are absent.

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

Variable	Cost of Omission
$\|c/v - 4.26\|$ [9]	100
$\|c/v - 4.26\|^2$ [10]	69
$(v/c)^2\times$ SAPS_BestCoeffVar_Mean [7 $\times$ 90]	53
$\|(c/v) - 4.26\|\times$ SAPS_BestCoeffVar_Mean [9 $\times$ 90]	33

Table 4.1: Variable Importance in Size-4 Model for Variable-Ratio Instances.

Figure 4.3 (right) shows the validation set RMSE of our best subset of each size. Note that our best four-variable model achieves a root-mean-squared error of 19.42 seconds, while our full 360-feature model had an error of about 14.57 seconds. Table 4.1 lists the four variables in this model along with their normalized costs of omission. Note that the most important feature (by far) is the linearized version of c/v, which also occurs (in different forms) in the other three features of this model. Hence, our techniques correctly identified the importance of the clauses-to-variables ratio, which satisfies our first goal. In terms of the second goal, these results indicate that the simple absolute distance of the ratio c/v from the critical value 4.26 appears to be the most informative variant of the nine related features we considered.

The third and fourth features in this model satisfy our third goal: we see that c/v variants are not the only useful features in this model. Interestingly, both of these remaining variables are based on a local search probing feature, the coefficient of variation over the number of clauses unsatisfied in local minima found by SAPS, a high-performance local search algorithm for SAT. It may appear somewhat surprising that such a local search probing feature can convey meaningful information about the runtime behavior of a DPLL algorithm. However, notice that deep local minima in the space searched by a local search algorithm correspond to assignments that leave few clauses unsatisfied. Intuitively, such assignments can cause substantial difficulties for DPLL search, where the respective partial assignments may correspond to large subtrees that do not contain any solutions. Nevertheless, our current understanding of the impact of the features captured by local search probes on DPLL solver performance is rather limited, and further work is needed to fully explain this phenomenon. This is an example of how empirical hardness models can shine light on new and possibly very important research questions.

The Weighted Clustering Coefficient

While analyzing our variable-ratio models, we discovered that the weighted clause graph clustering coefficient (33) was one of the most important features. In fact, it was the most important feature if we excluded higher-order c/v and v/c features from models. It turns out that the WCGCC is almost perfectly correlated with v/c, as illustrated in Figure 4.6 (left). This is particularly interesting as both the clustering coefficient and the connectivity of the constraint graph have been shown to be important statistics in a wide range of combinatorial problems, such as graph coloring and the combinatorial auctions WDP (see Chapter 3). This correlation provides a very nice new structural insight into the clause-to-variables ratio: it shows explicitly how constraint structure changes as the ratio varies. We should note, that one must not get too attached to the connection based on clustering coefficient. We have reasons to believe that at least in case of random SAT, the weight of these coefficients (essentially, node degrees) plays a more prominent role in determining hardness, whereas in other problems the clustering aspect might be more important. Clearly, this is yet another worthwhile research direction. Overall, this discovery demonstrates how our empirical hardness methodology can automatically help to gain new understanding of the nature of $\mathcal{NP}$-hard problems.

Satisfiable vs Unsatisfiable Instances

The previously mentioned similar performance of our predictive models for `kcnfs`, `satz` and `oksolver` raises the question of whether the underlying reason simply lies in a strong correlation between the respective runtimes. Figure 4.4 shows the correlation of `kcnfs` runtime vs. `satz` runtime on satisfiable and unsatisfiable instances. Note that there are two qualitatively different patterns in the performance correlation for the two types of instances: runtimes on UNSAT instances are almost perfectly correlated, while runtimes on SAT instances are almost entirely uncorrelated. We conjecture that this is because proving unsatisfiability of an instance essentially requires exploring the entire search tree, which does not differ substantially between the

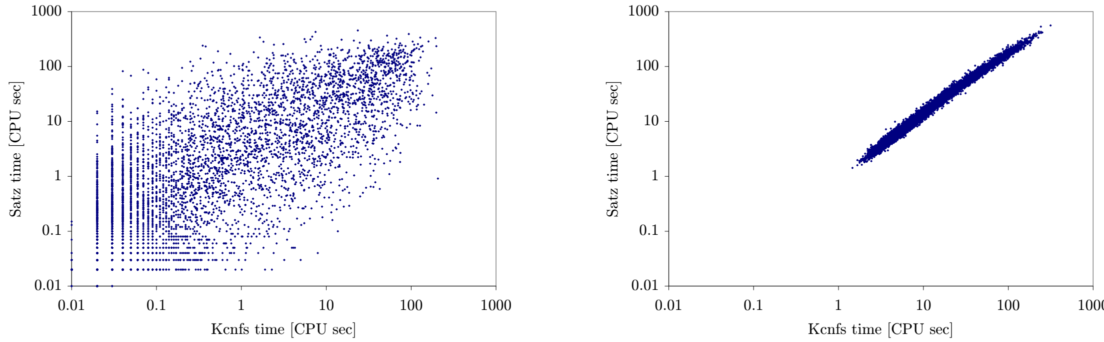


Figure 4.4: Runtime Correlation between `kcnfs` and `satz` for Satisfiable (left) and Unsatisfiable (right) Variable-Ratio Instances.

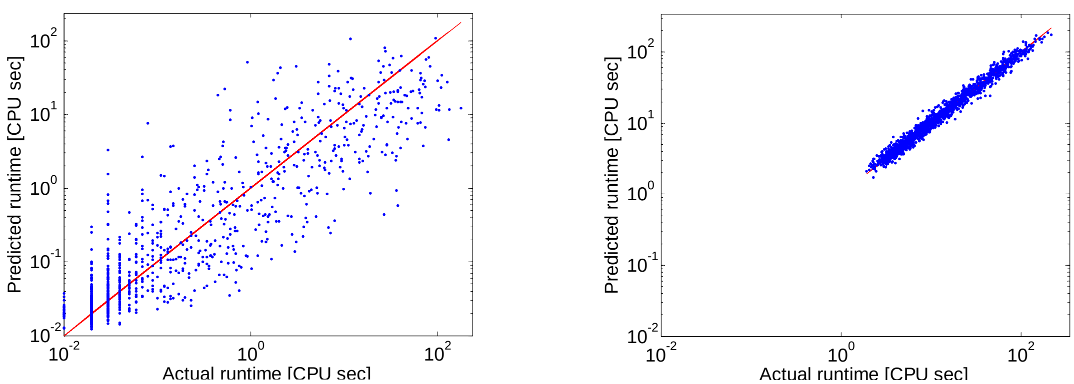


Figure 4.5: Actual vs. Predicted Runtimes for `kcnfs` on Satisfiable (left) and Unsatisfiable (right) Variable-Ratio Instances.

algorithms, while finding a satisfiable assignment depends much more on each algorithm's different heuristics. We can conclude that the similar model accuracy between the algorithms is due jointly to the correlation between their runtimes on UNSAT instances and to the ability of our features to express each algorithm's runtime profile on both SAT and UNSAT instances.

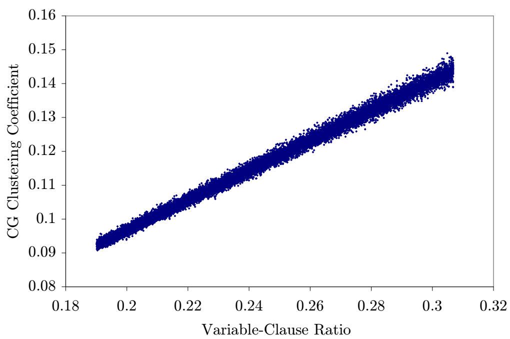
This separation of satisfiable and unsatisfiable regions of space has been independently confirmed by Gomes et al. [2004]. They showed that runtime distributions of DPLL solvers change regimes from extremely high variance (heavy-tailed) to low variance, as one moves across the phase transition.

Motivated by qualitative differences between satisfiable and unsatisfiable instances, we studied the subsets of all satisfiable and all unsatisfiable instances from our dataset separately. Analogously to what we did for the full dataset, we trained a separate predictive model for each of these two subsets. Interestingly, as seen in Figure 4.5, the predictions for unsatisfiable instances are much better than those for satisfiable instances (RMSE 5.3 vs. 13.4). Furthermore, the 'loss curves', which indicate the best RMSE achieved in dependence of model size (*cf.* Figure 4.3), are rather different between the two subsets: For the satisfiable instances, seven features are required to get within 10% of full model accuracy (in terms of RMSE), compared to only three for the unsatisfiable instances. While the seven features in the former model are all local search probe features (namely, in order of decreasing importance, features 68×68, 68×70, 90, 70, 70×70, 90×71, and 71), the three features in the latter are DPLL probe and constraint graph features (namely, features 66×66, 66, and 26×27).

It must be noted that excluding all local search probe features (68-91 in Figure 4.1) in the process of model construction leads to models with only moderately worse performance (RMSE 16.6 instead of 13.4 for satisfiable, 5.5 instead of 5.3 for unsatisfiable, and 17.2 instead of 13.2 for all instances). Interestingly, in such models for satisfiable instances, features based on LP relaxation (features 55–60 in Figure 4.1) become quite important. Even when excluding all probing and LP features (features 55-91), reasonably accurate models can still be obtained (RMSE 14.7, 8.4, and 17.1 for satisfiable, unsatisfiable, and all instances, respectively); this indicates that combinations of the remaining purely structural features still provide a sufficient basis for accurate runtime predictions on the variable-ratio instance distribution.

4.4.2 Fixed-Ratio Random Instances

According to a widely held (yet somewhat simplistic) belief, uniform random 3-SAT is easy when far from the phase-transition point, and hard when close to it. In fact, while the first part of this statement is generally true, the second part is not. Figure 4.6 (right) shows cumulative distributions of the `kcnfs`'s runtime per instance across our second dataset, comprising 20 000 fixed-ratio uniform random 3-SAT instances with 400 variables at $c/v = 4.26$, indicating substantial variation in runtime between



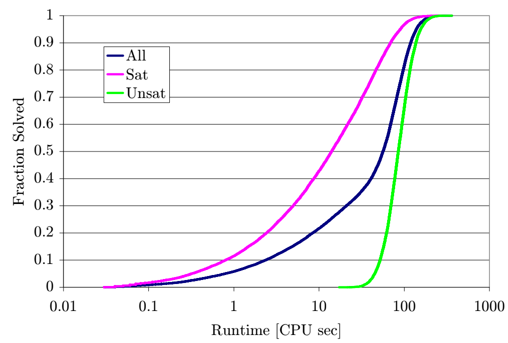


Figure 4.6: Left: Correlation between CG Weighted Clustering Coefficient and v/c. Right: Distribution of `kcnfs` Runtimes Across Fixed-Ratio Instances.

instances in the phase transition region. (Similar observations have been made previously for local search algorithms [Hoos and Stützle 1999].) Random-3-SAT at the phase transition point is one of the most widely used classes of benchmark instances for SAT; in the context of our study of empirical hardness models this instance distribution is particularly interesting since the most important features for predicting instance hardness for the variable-ratio distribution, namely variants of c/v, are kept constant in this case. Hence, it presents the challenge of identifying other features underlying the observed variation in hardness.

We built models in the same way as described in Section 4.4.1, except that all variants of c/v are constant and were hence omitted. Again, we achieved the best (validation set) results with logistic models on a (partial) quadratic expansion of the features; Figure 4.7 (left) shows the performance of our logistic model for `kcnfs` on test data (RMSE = 35.23); similar results were obtained for `oksolver` and `satz` (RMSE = 220.43 and 60.71, respectively; note that particularly for `oksolver` , the higher RMSE values are partly due to overall higher runtimes). The shape of the scatter plots can be visually misleading: although it appears to be not tight, there are many more points that lie along the diagonal than outliers (this becomes evident when plotting the data on a heat map).

Figure 4.7 (right) shows the validation set RMSE of the best model we found at each subset size. Here, a 4-variable model obtains RMSE 39.02 on the validation set, which is within 10% on the RMSE of the full model. The variables in the model,

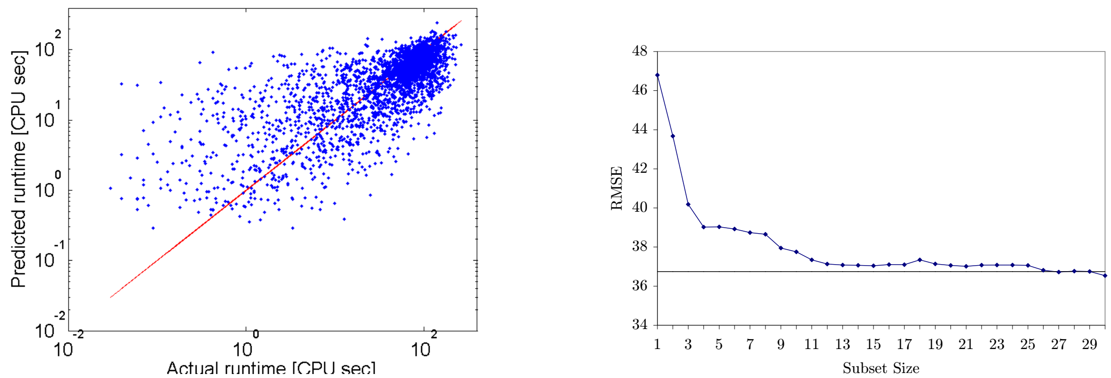


Figure 4.7: Actual vs. Predicted Runtimes for `kcnfs` on Fixed-Ratio Instances (left) and RMSE as a Function of Model Size (right).

`Variable`	`Cost of Omission`
SAPS_BestSolution_Mean^2 [68^2]	100
SAPS_BestSolution_Mean $\times$ Mean_DPLL_Depth [68×66]	74
GSAT_BestSolution_CoeffVar $\times$ Mean_DPLL_Depth [71×66]	21
VCG_CLAUSE_Mean $\times$ GSAT_FirstLMRatio_Mean [17×88]	9

Table 4.2: Variable Importance in Size-4 Model for Fixed-Ratio Instances.

along with their costs of omission, are given in Table 4.2. Note that this model is dominated by local search and DPLL probing features, and the most important feature is the deepest local minimum reached on a SAPS trajectory (BestSolution), which intuitively captures the degree to which a given instance has "almost" satisfying assignments.

As for the variable-ratio set, we studied the subsets of all satisfiable and all unsatisfiable instances from our fixed-ratio data set separately and trained separate models for each of these subsets. Analogous to our results for the variable-ratio sets, we found that our model for the former subset gave significantly better predictions than that for the latter (RMSE 15.6 vs. 30.2). Surprisingly, in both cases, only a single feature is required to get within 10% of full model accuracy (in terms of RMSE on the training set): the product of the two SAPS probing features 69 and 82 in the case of satisfiable instances, and the square of DPLL probing feature 66 in the case of unsatisfiable instances.

We also constructed models that do not use local search features and/or probing features and obtained results that are qualitatively the same as those for the variable-ratio data set. Furthermore, we have observed results on the correlation of runtimes between solvers that are analogous to those reported in Section 4.4.1.

4.5 SATzilla: An Algorithm Portfolio for SAT

It is well known that for SAT (as for many other hard problems) different algorithms often perform very differently on the same instances (*cf.* left side of Figure 4.4 which shows that there is very weak correlation between the runtimes of `kcnfs` and `satz` on satisfiable random 3-SAT instances). Thus, SAT seems well suited for our portfolio methodology of Section 2.4.2.

`SATzilla` was our attempt in constructing such a portfolio for SAT. It was entered in the 2003 and 2004 SAT competitions [Le Berre and Simon 2003]. In order to participate in these competitions, `SATzilla` had to employ runtime models across a much less structured distribution than uniformly random instances. Moreover, during the competition it would face completely new instances, often of a previously unseen kind. As a result, the models that we obtained had, in some parts of space, predictive power that is far worse than that of the other models described in this chapter. Nevertheless, `SATzilla` performed reasonably well. The reason for that is that fairly inaccurate models often suffice to build good portfolios: if algorithms' performances are close to each other, picking the wrong one is not very costly, while if algorithms' behaviors differ significantly, the discrimination task is relatively easy.

The version of `SATzilla` built for the 2003 competition consisted of `2clseq`, `eqSatz`, `HeerHugo`, `JeruSat`, `Limmat`, `oksolver`, `Relsat`, `Sato`, `Satz-rand` and `zChaff`. The 2004 version dropped `HeerHugo`, but added `Satzoo`, `kcnfs`, and `BerkMin`, new solvers that appeared in 2003 and performed well in the 2003 competition.

To construct `SATzilla` we gathered from various public websites a library of about 5 000 SAT instances, for which we computed runtimes and the features described in Section 4.3. We built models using ridge regression, a machine learning technique that finds a linear model (a hyperplane in feature space) that minimizes a combination of root mean squared error and a penalty term for large coefficients. To yield better models, we dropped from our dataset all instances that were solved by all or none of the algorithms, or as a side-effect of feature computation.

Upon execution, `SATzilla` begins by running a UBCSAT [Tompkins and Hoos 2004] implementation of `WalkSat` for 30 seconds. In our experience, this step helps to filter out easy satisfiable instances. Next, `SATzilla` runs the `Hypre` preprocessor [Bacchus and Winter 2003], which uses hyper-resolution to reason about binary clauses. This step is often able to dramatically shorten the formula, sometimes resulting in search problems that are easier for DPLL-style solvers. Perhaps more importantly, the simplification "cleans up" instances, allowing the subsequent analysis of their structure to better reflect the problem's combinatorial "core"[3]. Third, `SATzilla` computes its features. Sometimes, a feature can actually solve the problem; if this occurs, execution stops. We found it worthwhile to train models on instances that cannot be solved by features. Some features can also take an inordinate amount of time, particularly with very large inputs. To prevent feature computation from consuming all of our allotted time, certain features run only until a timeout is reached, at which point `SATzilla` gives up on computing the given feature. Fourth, `SATzilla` evaluates a regression model for each algorithm in order to compute a prediction of that algorithm's running time. If some of the features have timed out, a different model is used, which does not involve the missing feature and which was trained only on instances where the same feature timed out. Finally, `SATzilla` runs the algorithm with the best predicted runtime until the instance is solved or the allotted time is used up.

As described in the official report written by the 2003 SAT competition organizers [Le Berre and Simon 2003], `SATzilla`'s performance in this competition demonstrated the viability of our portfolio approach. `SATzilla` qualified to enter the final round in two out of three benchmark categories – Random and Handmade. Unfortunately, a bug caused `SATzilla` to crash often on Industrial instances (due to their extremely large sizes) and so `SATzilla` did not qualify for the final round in this category.

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

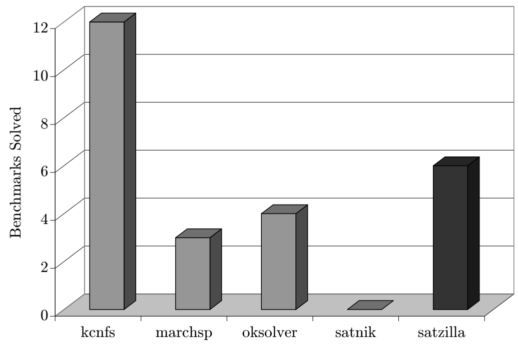


Figure 4.8: SAT-2003 Competition, Random Category.

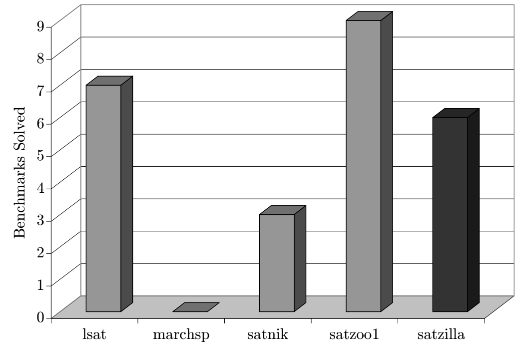


Figure 4.9: SAT-2003 Competition, Handmade Category.

During the competition, instances were partitioned into different *series* based on their similarity. Solvers were then ranked by the number of series in which they managed to solve at least one benchmark. `SATzilla` placed second in the Random category (the first solver was `kcnfs`, which wasn't in the portfolio as it hadn't yet been publicly released). In the Handmade instances category `SATzilla` was third (2^{nd} on satisfiable instances), again losing only to new solvers.

Figures 4.8 and 4.9 show the raw number of instances (in tens) solved by the top five solvers in each of the Random and Handmade categories. In general the solvers that did well in one category did very poorly (or didn't qualify for the final) in the other. `SATzilla` was the only solver that achieved strong performance in both categories.

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

The 2004 competition involved a number of changes to the rules (some of which were announced during the competition!) that make drawing meaningful results for that year difficult. The first problem was that the timeout used to terminate solvers was eventually chosen to be shorter than expected, in order to cope with limited computational resources and an increased number of submissions. `SATzilla`'s performance was affected severely by this, as it consumed a very non-trivial portion of its time evaluating features. With the new timeout `SATzilla` often had very little time left to actually solve the problem — an effect that was particularly pronounced on unsatisfiable instances. This effect was compounded by the new rule that all solvers were now run together, and not grouped into incomplete and complete types. As a result, a solver could advance to the second round either if it solved a large number of instances quickly (which in practice only advanced incomplete solvers), or if it did quite well on unsatisfiable instances (which essentially excluded `SATzilla` for the reason above).

Although `SATzilla` didn't advance to the second round, and hence we have limited data on its performance in 2004, we did observe a very interesting trend. In the first round the solvers were clearly clustered in their performance according to the number of instances solved. The cluster with the better performance consisted exclusively of *incomplete* solvers and `SATzilla` (albeit, last among those). *All* of the other complete solvers formed the second cluster. Therefore, we still strongly believe that `SATzilla` is a very successful approach to SAT.

4.6 Conclusion and Research Directions

In this chapter we have once again validated the empirical hardness methodology of Chapter 2. We have shown that empirical hardness models are a valuable tool for the study of the empirical behavior of complex algorithms such as SAT solvers by building accurate models of runtime on test distributions of fixed- and variable-ratio uniform random-3-SAT instances. On the variable-ratio dataset, our techniques were able to automatically "discover" the importance of the c/v ratio. Analysis in this case provided insight into the structural variations in uniform random 3-SAT formulae at the phase transition point that correlate with the observed dramatic variation in empirical hardness. Finally, we argued that our empirical hardness models offer practical benefit in less well-controlled domains by presenting `SATzilla`, the algorithm portfolio for SAT.

More importantly, the results presented highlight a number of avenues for future research in both SAT and typical-case complexity communities. These are:

- Investigate the surprisingly close relationship between DPLL-type solvers and stochastic local search methods on satisfiable instances.

- Further understand the apparently fundamental distinction between satisfiable and unsatisfiable instances (which is already known in the local search community, and partly explored by Gomes et al. [2004]).

- Understand the structural effect of the clustering coefficient and node degrees of the clause graph on problem hardness, and especially investigate this as a venue for relating SAT to other hard problems.

We strongly believe that by studying in more detail *how* some of the features identified through the use of predictive statistical models cause instances to be easy or hard for certain types of algorithms, our understanding of how to solve SAT most efficiently will be further advanced.

Part II

Algorithms as Tools

Chapter 5

Computational Game Theory

In this chapter we present some background on computational game theory that is necessary for understanding chapters 6 and 7.

5.1 Game Theory Meets Computer Science

In recent years, researchers in artificial intelligence and, in fact, all of computer science have become increasingly interested in game theory as a modeling tool.

A lot of research in AI focuses on the design of intelligent autonomous agents that act (*i.e.,* continuously make *decisions*) in simulated or real-world environments. The common view that is taken (*e.g.,* [Russel and Norvig 2003]) is that the agent should act so as to maximize its *expected utility*. This is transplanted directly from the classical decision theory. Famous representation theorems, such as the one due to Savage [1954], indeed demonstrate that many decision-making situations can be reduced to utility maximization.

However, one might argue that one mark of intelligence is its ability to be successful in achieving its goals while co-existing with other similarly intelligent beings. After all, short of an intelligent rover on a desert planet, a rational agent will likely have to deal at least with humans in addition to other autonomous agents. Such *multiagent* settings appear to be fundamentally different from the classical decision-theoretic settings — the acts of different agents become intricately intertwined. As with general reactive environments, what an agent does now will effect whatever happens in the future. However, the actions of other agents may suddenly become much less predictable. What an agent does may depend on what it *believes* others will do in the future, which, in turn, may depend on what they believe the agent believes about them, and so on, ad infinitum. As a simple example, consider driving agents that must pick a side of the road to drive on. There is no intrinsic advantage to either choice, *i.e.,* there is no optimal choice. However, each agent wants to pick the same side as everybody else. Similarly, the choice of the side for everybody else depends on what they believe about our agent, and so on. As is common in dynamical systems, in such multiagent systems the notion of optimality (*e.g.,* utility maximization) has to give way to the notion of an equilibrium. Game theory (see, *e.g.,* [Osborne and Rubinstein 1994] for an introduction) was developed by economists precisely to model such interactions among small numbers of self-interested agents. Nash equilibrium [Nash 1950] is the classical solution concept that, as a first approximation, generalizes utility maximization.

Besides appearing extremely relevant to the AI researchers, game theory, and, more generally, microeconomic models, came into sharp focus of general computer science research with the rise of the Internet [Papadimitriou 2001]. Internet provides an extremely complex environment in which lots of self-interested entities, such as humans, corporations, web-crawler agents, network routers, etc. interact on a continuous basis. Such massive interaction poses a large number of important questions. From the perspective of both an agent designer, or a human dealing with a website, it is imperative to understand what is the best way to act in such an environment. A protocol designer might ask what global behavior might emerge if everybody acts in their own interest, rather than blindly executes prescribed actions (leading, *e.g.,* to the study of the price of anarchy [Roughgarden 2005]). Even more interestingly, one might try to design protocols with just the right incentives for agents to follow them and achieve desired global behavior (this sort of "inverse" game theory is the thriving field called Mechanism Design). In fact, a lot of research in auctions, and combinatorial auctions in particular, got started precisely with these motivations. Our study of empirical complexity for the combinatorial auctions WDP (see Chapter 3) was fueled by this interest in auctions.

These largely philosophical considerations explain why economic models are interesting and relevant to CS researchers in many fields. However, these notions, with game theory in particular, have been mostly developed by economists as modeling tools, with almost complete disregard for the associated computational questions, such as how one might automatically reason about these models and compute various solution concepts, or what is the complexity of such computation. It turned out that, regardless of practical applicability of game theory, these questions are very interesting to computer scientists in their own right, as they often seem to be quite different from our traditional areas of expertise.

5.2 Notation and Background

In this section we formally define basic game-theoretic notions and notation that will be used throughout subsequent chapters.
First, we define the basic object of study in non-cooperative game-theory — a normal-form game.

Definition 5.1 *A finite, n-player,* **normal-form game** *is a tuple* $G = \langle N, (A_i), (u_i) \rangle$, *where*

- $N = \{1, \ldots, n\}$ *is the set of players.*
- $A_i = \{a_{i1}, \ldots, a_{im_i}\}$ *is a finite set of actions (also called pure strategies) available to player i, where m_i is the number of available actions for that player.*
- $u_i : A_1 \times \ldots \times A_n \to \mathbb{R}$ *is the utility function for each player i. It maps a profile of actions to a value.*

The idea behind this model is that all players simultaneously choose one of the possible actions. Their utility then depends on the actions of all players together, rather than only on their own. Although it might appear simple, such a model can in principle capture a wide variety of strategic interactions, as long as the setting is common knowledge among players. For example, it is possible to represent a complicated

		Agent 2: L	Agent 2: R
Agent 1	L	$10, 10$	$-100, -100$
	R	$-100, -100$	$10, 10$

Figure 5.1: A Coordination Game.

sequence of interactions that unfolds over time by having strategies that represent all possible plans of action, contingent on all situations that might hypothetically arise. The idea then is that all agents choose such a plan simultaneously in the beginning, and the actual interaction is then executed by robots following pre-specified plans.

For a concrete example, consider the game in Figure 5.1. This game captures the scenario of 2 agents choosing which side of the road to drive on, as described in the previous section. Here their interests are aligned in principle, but they face a problem of coordination. Rows represent possible strategies of the first player (drive on the left or on the right), while columns represent those for the second player. Numbers in each cell represent utility achieved by first and second player respectively if the corresponding outcome is realized. This game encodes the fact that both players get disastrous payoffs (-100) if they choose different driving convention, as that is very likely to lead to an accident. On the other hand, choosing the same convention allows them to get positive utility.

We will use a_i as a variable that takes on the value of a particular action a_{ij} of player i, and $a = (a_1, \ldots, a_n)$ to denote a profile of actions, one for each player. Also, let $a_{-i} = (a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)$ denote this same profile excluding the action of player i, so that (a_i, a_{-i}) forms a complete profile of actions. We will use similar notation for any profile that contains an element for each player.

A key notion that is used in game theory is that of a mixed strategy. A mixed strategy for a player specifies the probability distribution used to select an action that a player will play in a game.

Definition 5.2 *A* **mixed strategy** p_i *for a player* i *is a probability distribution over* A_i.

We will denote by $\mathcal{P}_i$ the set of available mixed strategies for player i:

$$\mathcal{P}_i = \{p_i : A_i \to [0,1] | \sum_{a_i \in A_i} p_i(a_i) = 1\}$$

We will sometimes use a_i to denote the mixed strategy in which $p_i(a_i) = 1$.

Definition 5.3 *The* **support** *of a mixed strategy p_i is the set of all actions $a_i \in A_i$ such that $p_i(a_i) > 0$.*

We will use $x = (x_1, \ldots, x_n)$ to denote a profile of values that specifies the *size* of the support of each player.

Notice that each mixed strategy profile $p = (p_1, \ldots, p_n)$ induces a probability distribution $p(a) = \Pi_{i \in N} p_i(a_i)$ over pure-strategy outcomes of the game. We can thus extend u_i to the mixed strategy profiles to denote *expected* utility for player i: $u_i(p) = \sum_{a \in A} p(a) u_i(a)$.

Now that we have defined the model of strategic interactions, it is natural to ask how the agents should behave in such interactions. As was mentioned above, agents now cannot simply optimize their expected utility, since that utility depends on actions of other agents. Instead, we can talk about *conditional* optimality.

Definition 5.4 *An action a_i is a* **best-response** *of player i to the strategy profile p_{-i} of other players, if $u_i(a_i, p_{-i}) \geq u_i(a'_i, p_{-i})$ for all $a'_i \in A_i$.*

In other words, a_i is the best action player i given that all other players play according to p_{-i}. We say that a mixed-strategy profile p_i is a best response to p_{-i} if all actions in the *support* of p_i are best responses.

Now we are ready to define the primary solution concept for a normal-form game: the Nash equilibrium. The idea behind a Nash equilibrium is that if all players are playing best-responses to the rest, then no agent has incentive to unilaterally deviate, and thus the profile is stable.

Definition 5.5 *A (mixed) strategy profile $p^* \in \mathcal{P}$ is a* **Nash equilibrium (NE)** *if: $\forall i \in N, a_i \in A_i : \quad u_i(a_i, p^*_{-i}) \leq u_i(p^*_i, p^*_{-i})$.*

The famous theorem due to Nash [1950] states that a mixed-strategy Nash equilibrium exists for any normal-form game.

Theorem 5.6 (Nash 1950) *Any finite n-player game G in normal form has a mixed-strategy Nash equilibrium.*

The proof of this theorem is based on classical fixed-point theorems from mathematical analysis (such as Kakutani's or Brouwer's) applied to best-response correspondences. Unfortunately, such proofs are non-constructive.

5.3 Computational Problems

We now turn our attention to some interesting computational problems in game theory. There are numerous questions one might ask about normal-form games: we might try to determine which games are equivalent, look for solutions, eliminate redundant and/or dominated strategies, etc. In this section we will briefly describe two of them that will get mentioned subsequently in this thesis. These are the problems of finding Nash equilibria and of adapting to unknown opponents.

5.3.1 Finding Nash Equilibria

The main problem to which the second part of this thesis is devoted is that of finding a sample Nash equilibrium of a normal-form game.

Problem 5.7 *Given a game G in normal form, compute some mixed-strategy profile p which is a Nash equilibrium of G.*

Despite the unquestionable importance of the Nash equilibrium concept in game theory, remarkably little is known about this problem. Part of the difficulty lies with the fact that existence proofs are non-constructive and don't shed light onto the question of computing a NE.

One technical hurdle is that Problem 5.7 does not fall into a standard complexity class [Papadimitriou 2001], because it cannot be cast as a decision problem. We know that the answer to the existence question is "yes". This makes this somewhat similar to the problem of factoring, complexity of which is also unknown. Megiddo and Papadimitriou [1991] defined a new (but rather limited) complexity class, $\mathcal{TFNP}$ (total functions $\mathcal{NP}$), to encompass such "finding" problems. Nevertheless, we do have a lot of evidence pointing to this being a hard problem [Gilboa and Zemel 1989; Conitzer and Sandholm 2003]. It appears that any attempt to turn this into a decision problem, either by imposing restrictions on the kind of NE that is desired, or by asking questions about properties of potential NEs, immediately renders the problem to be $\mathcal{NP}$- or $co\mathcal{NP}$-hard. However, these results don't have immediate implications for the complexity of finding *any* NE. Altogether, this state of affairs prompted Papadimitriou to call the complexity of the problem of finding a sample Nash equilibrium, together with factoring, "the most important concrete open question on the boundary of $\mathcal{P}$ today" [Papadimitriou 2001].

From the mathematical programming point of view, Problem 5.7 can be formulated in a number of ways [McKelvey and McLennan 1996]: optimization of a non-linear function, finding a fixed-point of a correspondence or of a function, minimizing a function on a polytope, semi-algebraic programming, or a non-linear complementarity problem (NCP) (the problem of finding two orthogonal vectors satisfying certain constraints). The non-linearity of the NCP formulation is due to computing product probabilities over all but one players; it goes away in the case of only two players. The complementarity constraint essentially arises because actions in the support of each equilibrium strategy must be different from those outside of the support: each agent must be indifferent among the actions in his support, and prefer them to any action that is not included.

Existing Algorithms

Although the complexity of Problem 5.7 remains open, there exist a number of algorithms for computing Nash equilibria. In this section, we provide a brief overview of them. In addition to specific references given for each algorithm, further explanation can be found in two thorough surveys on NE computation – [von Stengel 2002] and [McKelvey and McLennan 1996]. One thing that is relatively common to all of these algorithms is that they are based on fairly deep mathematical insights into the nature of Nash equilibria. For that reason, they tend to be hard to analyze. All of them are known to have worst-case running times that are at least exponential; however, precisely characterizing their running times remains elusive. More importantly, from the point of view of this thesis, it is very hard to elicit connections between problem structure and successes or failures of these algorithms.

The most commonly-used algorithm for finding a NE in two-player games is the Lemke-Howson algorithm [Lemke and Howson 1964], which is a special case of Lemke's method [Lemke 1965] for solving linear complementarity problems. The Lemke-Howson algorithm is a complementary pivoting algorithm, where an arbitrary selection of an action for the first player determines the first pivot, after which every successive pivot is determined uniquely by the current state of the algorithm, until an equilibrium is found. Thus, each action for the first player can be thought of as defining a path from the starting point (the extraneous solution of all players assigning probability zero to all actions) to a NE. In the implementation of Lemke-Howson in Gambit [McKelvey et al. 1992], used in experimental results in the subsequent chapters, the first action of the first player is selected. Lemke-Howson is known to take exponentially many pivoting steps in the worst case [Savani and von Stengel 2004].

For n-player games, until recently, Simplicial Subdivision [van der Laan et al. 1987], and its variants, were the state of the art. This approach approximates a fixed point of a function (*e.g.,* the best-response correspondence) which is defined on a simplotope (a product of simplices). The approximation is achieved by triangulating the simplotope with a mesh of a given granularity, and traversing the triangulation along a fixed path. The worst-case running time of this procedure is exponential in dimension and accuracy [McKelvey and McLennan 1996].

More recently, Govindan and Wilson [2003] introduced a continuation method for finding a NE in an n-player game. Govindan-Wilson works by first perturbing a game to one that has a known equilibrium, and then by tracing the solution back to the original game as the magnitude of the perturbation approaches zero. The structure theorem of Kohlberg and Mertens [1986] guarantees that it is possible to trace both the game and the solution simultaneously. This method has been efficiently implemented by Blum et al. [2003], who also extended it to solve graphical games and Multi-Agent Influence Diagrams [Koller and Milch 2001].

The algorithm that is described in Chapter 7 in spirit is closest to the procedure described by Dickhaut and Kaplan [1991] for finding all NEs. Their program enumerates all possible pairs of supports for a two-player game. For each pair of supports, it solves a feasibility program (similar to the one we will describe below) to check whether there exists a NE consistent with this pair. A similar enumeration method was suggested earlier by Mangasarian [1964], based on enumerating vertices of a polytope. Clearly, either of these two enumeration methods could be converted into an algorithm for finding a sample NE by simply stopping after finding the first NE. However, because the purpose of these algorithms is to instead find all NEs, no heuristics are employed to speed the computation of the first NE.

The most recent algorithm, besides ours, for finding a NE was proposed in [Sandholm et al. 2005]. Their method is applicable only to the two-player games. It is based on formulating the problem of finding a NE as a mixed-integer linear program (in fact, a feasibility program). While the fact that it is easy in principle to use MIP to solve this problem has been folk knowledge in the OR community for many years, this work appears to be the first to provide an experimental evaluation of such methods. In principle, their algorithm is quite similar to the algorithm described in Chapter 7. Binary variables essentially enumerate all possible pairs of supports (*cf.* [Dickhaut and Kaplan 1991]), while the rest of the constraints are the same as the ones utilized explicitly in Chapter 7. The major difference is that [Sandholm et al. 2005] rely on general-purpose MIP software (namely, CPLEX) to provide enumeration order, rather than use domain specific heuristics. Unfortunately, their experimental results show that this approach in general is not competitive with either Lemke-Howson, or the algorithm of Chapter 7. It does have an advantage of being easy to modify in order to find equilibria with desired properties.

5.3.2 Multiagent Learning

Although the second part of this thesis is mostly devoted to the problem of finding a sample Nash equilibrium, in Chapter 6 we will refer to another problem associated with normal-form games — multiagent learning in repeated normal-form games. This research area is still very young. One of the biggest challenges that this field is currently facing is a precise definition of the problem, as different researchers often strive to achieve slightly different objectives [Shoham et al. 2004].

For the purposes of this work we can informally state the problem as follows:

Problem 5.8 *Devise an algorithm that, given a normal-form game G, own player's identity i, and a set of "black-box" non-static opponents, quickly starts achieving a good payoff when playing G repeatedly against the given opponents.*

This description is intentionally vague, and requires clarification on several points. For example, there are several interpretations of what a "good payoff" might be. One reasonable requirement is for the algorithm to play best-response to its opponents in each stage of the repeated game. This can be problematic, however. If the game is repeated, opponent's actions may depend on the whole past history of play, which means that such a goal might be unachievable. More importantly, because of the repeated nature, much better cumulative payoffs might be attainable by making occasional concessions to the opponents. Therefore, a common requirement is to achieve a good *average* payoff in the limit. Often, such an average is also computed in a *discounted* fashion, in order to emphasize immediate rewards.

Problem 5.8 is unsolvable as stated. Another very reasonable restriction (as proposed by [Powers and Shoham 2004; Powers and Shoham 2005]) is to require good performance only against a specific *class* of opponents, and provide some minimum payoff guarantees against the rest. A lot of work (*e.g.,* [Bowling and Veloso 2001]) has also proposed an additional requirement of some form of *convergence* on the algorithms. While this appears to be natural against opponents that themselves eventually converge, we do not view this requirement as intrinsic to the problem.

In stating Problem 5.8 we have assumed that the game's payoffs are known, and the only thing that is not known to the agent is the behavior of its opponents. Thus, it might be more proper to call this the problem of *online adaptation*. In fact, a lot of work in the field (with few notable exceptions), does make the same assumption. Therefore, for historical reasons, we will continue to use the term multiagent learning.

The last few years have seen a surge of research into multiagent learning, resulting in the recent proposal of many new algorithms. Here we briefly describe the few that are mentioned in Chapter 6: Minimax-Q [Littman 1994], WoLF (Win or Learn Fast) [Bowling and Veloso 2001], and Single-Agent-Q — a version of the original Q-learning algorithm for single agent games [Watkins and Dayan 1992] modified for use by an individual player in a repeated game setting. These algorithms have received much study in recent years; they each have very different performance guarantees, strengths and weaknesses. Single-agent Q-learning is simply the standard reinforcement learning algorithm forced to play in a repeated game. It is thus completely oblivious to the existence of opponents and their payoffs, *i.e.,* it assumes away the multiagent component. Therefore, single-Agent-Q is not guaranteed to converge at all against an adaptive opponent. Minimax-Q, one of the earliest proposed multiagent learning algorithms, is a slightly smarter adaptation of the classic Q-learning idea. The major difference is the way the value of the current state is computed. Minimax-Q assumes a safety-level (minimax) strategy. Thus, it does not necessarily converge to a best response in general-sum games, though it converges in self-play in zero-sum games. Finally, WoLF is a variable-learning-rate policy-hill-climbing algorithm that *is* designed to converge to a best response, assuming its opponent stops adapting at some point.

Chapter 6

Evaluating Game-Theoretic Algorithms

Unlike many theoretical studies into the nature of computational problems, empirical studies cannot be performed on a problem "as a whole"; they always require a precise underlying *distribution* of problem instances. In this chapter we present GAMUT[1], a suite of game generators that became a definitive testbed for game-theoretic algorithms. We explain why such a generator is necessary, offer a way of visualizing relationships between the sets of games supported by GAMUT, and give an overview of GAMUT's architecture. Experimentally, we highlight the importance of using comprehensive test data by showing surprisingly large variation in algorithm performance across different sets of games for the problem of finding a sample Nash equilibrium and for multiagent learning.

6.1 The Need for a Testbed

One general lesson that has been learned in the past decade by researchers working in a wide variety of different domains is that an algorithm's performance can vary substantially across different "reasonable" distributions of problem instances, even when problem size is held constant (*cf.* results in chapters 3 and 4). When we examine

[1] Available at `http://gamut.stanford.edu`

the empirical tests that have been performed on algorithms that take games as their inputs, we find that they have typically been small-scale and involved very particular choices of games. Such tests can be appropriate for limited proofs-of-concept, but cannot say much about an algorithm's expected performance in new domains. For this, a comprehensive body of test data is required.

It is not obvious that a library of games should be difficult to construct. After all, games (if we think for the moment about normal-form representations) are simply matrices with one dimension indexed by action for each player, and one further dimension indexed by player. We can thus generate games by taking the number of players and of actions for each player as parameters, and populate the corresponding matrix with real numbers generated uniformly at random. Is anything further required?

We set out to answer this question by studying sets of games that have been identified as interesting by computer scientists, game theorists, economists, political scientists and others over the past 50 years. Our attempt to get a sense of this huge literature led us to look at several hundred books and papers, and to extract one or more sets of games from more than a hundred sources[2]. To our surprise, we discovered two things.

First, for *all but a few* of the sets of games that we encountered, the technique described above would generate a game from that set with probability zero. More formally, all of these sets are *non-generic* with respect to the uniform sampling procedure. It is very significant to find that an unbiased method of generating games has only an infinitesimal chance of generating any of these games that have been considered realistic or interesting. Since we know that algorithm performance can depend heavily on the choice of test data, it would be unreasonable to extrapolate from an algorithm's performance on random test data to its expected performance on real-world problems. It seems that test data for games must take the form of a patchwork of generators of different sets of games.

Second, we were surprised to find very little work that aimed to understand,

[2]We compiled an extensive bibliography referencing this literature; it is available online in the form of an interactive database. To learn more, please see Section 6.2.1.

taxonomize or even enumerate non-generic games in a holistic or integrative way. We came across work on understanding *generic* games [Kohlberg and Mertens 1986], and found a complete taxonomy of two-player two-action games [Rapoport et al. 1976]. Otherwise, work that we encountered tended to fall into one or both of two camps. Some work aimed to describe and characterize particular sets of games that were proposed as reasonable models of real-world strategic situations or that presented interesting theoretical problems. Second, researchers proposed novel representations of games, explicitly or implicitly identifying sets of games that could be specified compactly in these representations.

The work on GAMUT aimed to fill this gap: to identify interesting sets of non-generic games comprehensively and with as little bias as possible. Indeed, immediately after its initial release in 2004 GAMUT was used by several researchers in the field, demonstrating the real need for it.

In the next section we describe this effort, highlighting relationships between different sets of games we encountered in our literature search and describing issues that arose in the identification of game generation algorithms. In Section 6.3 we give experimental proof that a comprehensive test suite is required for the evaluation of game-theoretic algorithms. For our two example problems, computing Nash equilibria and learning in repeated games, we show that performance for different algorithms varies dramatically across different sets of games even when the size of the game is held constant, and that performance on random games can be a bad predictor of performance on other games. Finally, in Section 6.4, we briefly describe GAMUT's architecture and implementation, including discussion of how new games may easily be added.

6.2 GAMUT

For the current version of GAMUT we considered only games whose normal-form representations can be comfortably stored in a computer. Note that this restriction does not rule out games that are presented in a more compact representation such as extensive form or graphical games; it only rules out *large* examples of such games. It also rules out games with infinite numbers of agents and/or of actions and Bayesian games. We make no requirement that games must *actually* be stored in normal form; in fact, GAMUT supports a wide array of representations (see Section 6.4). Some are *complete* (able to represent any game) while other *incomplete* representations support only certain sets of games. We will say that a given representation describes a set of games *compactly* if its descriptions of games in the set are exponentially shorter than the games' descriptions in normal form.

In total we identified 122 interesting sets of games in our literature search, and we were able to find finite time generative procedures for 71. These generative sets ranged from specific two-by-two matrix games with little variation (*e.g.,* Chicken) to broad classes extensible in both number of players and number of actions (*e.g.,* games that can be encoded compactly in the Graphical Game representation).

6.2.1 The Games

To try to understand the relationships between these different sets of non-generic games, we set out to relate them taxonomically. We settled on identifying subset relationships between the different sets of games. Our taxonomy is too large to show in full, but a fragment of it is shown in Figure 6.1. To illustrate the sort of information that can be conveyed by this figure, we can see that all Dispersion Games [Grenager et al. 2002] are Congestion Games [Rosenthal 1973] and that all Congestion Games have pure-strategy equilibria.

Besides providing some insight into the breadth of generators included in GAMUT and the relationships between them, our taxonomy also serves a more practical purpose: allowing the quick and intuitive selection of a set of generators. If GAMUT is directed to generate a game from a set that does not have a generator (*e.g.,* supermodular games [Milgrom and Roberts 1990] or games having unique equilibria) it chooses uniformly at random among the generative descendants of the set and then generates a game from the chosen set. GAMUT also supports generating games that belong to multiple intersecting sets (*e.g.,* symmetric games having pure-strategy equilibria); in this case GAMUT chooses uniformly at random among the generative sets that are

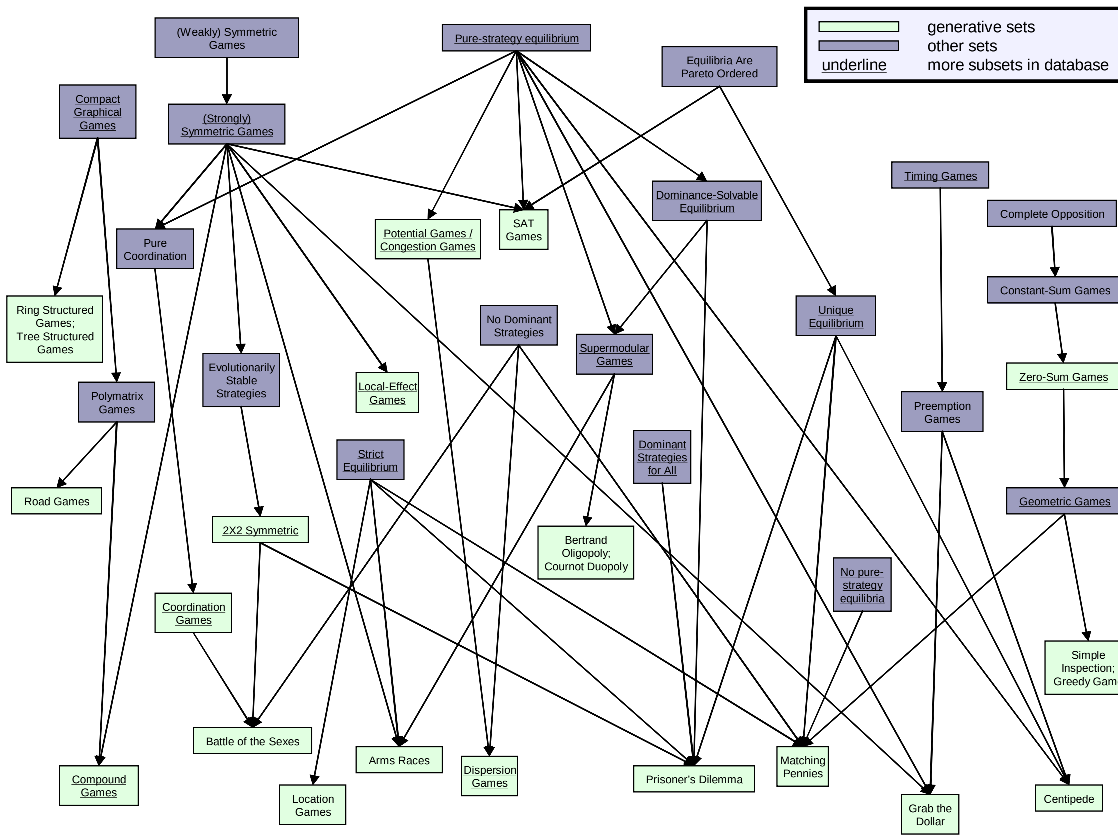


Figure 6.1: GAMUT Taxonomy (Partial).

descendants of all the named sets.

The data we collected in our literature search — including bibliographic references, pseudo-code for generating games and taxonomic relationships between games — will be useful to some researchers in its own right. We have gathered this information into a database which is publicly available from `http://gamut.stanford.edu` as part of the GAMUT release. Besides providing more information about references, this database also allows users to navigate according to subset/superset relationships and to perform searches.

6.2.2 The Generators

Roughly speaking, the sets of games that we enumerated in the taxonomy can be partitioned into two classes, reflected by different colored nodes in Figure 6.1. For some sets we were able to come up with an efficient algorithmic procedure that can, in finite time, produce a sample game from that set, and that has the ability to produce any game from that set. We call such sets *generative*. For others, we could find no reasonable procedure. One might consider a rejection sampling approach that would generate games at random and then test whether they belong to a given set S. However, if S is non-generic — which is true for most of our sets, as discussed above — such a procedure would fail to produce a sample game in any finite amount of time. Thus, we do not consider such procedures as generators.

Cataloging the relationships among sets of games and identifying generative sets prepared us for our next task, creating game generators. The wrinkle was that generative algorithms were rarely described explicitly in the literature. While in most cases coming up with an algorithm was straightforward, we did encounter several interesting issues.

Sometimes an author defined a game too narrowly for our purposes. Many traditional games (*e.g.,* Prisoner's Dilemma) are often defined in terms of precise payoffs. Since our goal was to construct a generator capable of producing an infinite number of games belonging to the same set, we had to generalize these games. In the case of Prisoner's Dilemma (Figure 6.2), we can generate any game

R, R	S, T
T, S	P, P

Figure 6.2: Generic Prisoner's Dilemma.

which satisfies $T > R > P > S$ and $R > (S+T)/2$. (The latter condition ensures that all three of the non-equilibrium outcomes are Pareto optimal.) Thus, an algorithm for generating an instance of Prisoner's Dilemma reduces to generating four numbers that satisfy the given constraints. There is one subtlety involved with this approach to generalizing games. It is a well-known fact that a positive affine transformation of payoffs does not change strategic situation modeled by the game. It is also a common practice to normalize payoffs to some standard range before reasoning about games. We ensure that no generator ever generates instances that differ only by a positive affine transformation of payoffs.

In other cases the definition of a set was too broad, and thus had to be restricted. In many cases, this could be achieved via an appropriate parametrization. An interesting example of this is the set of Polymatrix Games [Govindan and Wilson 2003]. These are n-player games with a very special payoff structure: every pair of agents plays a (potentially different) 2-player game between them, and each agent's utility is the sum of all of his payoffs. The caveat, however, is that the agent must play the *same* action in all of his two-player games. We realized that these games, though originally studied for their computational properties, could be generalized and used essentially as a compact representation for games in which each agent only plays two-player games against some *subset* of the other agents. This led to a natural parametrization of polymatrix games with graphs. Nodes of the graph now represent agents, and edges are labeled with different 2-player games.[3] Thus, though we still can sample from the set of all polymatrix games using a complete graph, we are now able also to focus on more specific and, thus, even more structured subsets.

Sometimes we encountered purely algorithmic difficulties. For example, in order to implement geometric games [Ruckle 1983] we needed data structures capable of representing and performing operations on abstract sets (such as finding intersection, or enumerating subsets).

In some cases one parameterized generator was able to generate games from many different sets. For example, we implemented a single generator based on work by Rapoport [Rapoport et al. 1976] which demonstrated that there are only 85 strategically different 2x2 games, and so did not need to implement generators for individual 2x2 games mentioned in the literature. We did elect to create separate generators for several very common games (*e.g.,* Matching Pennies; Hawk and Dove). We also used our taxonomy to identify similar sets of games, and either implemented them with the same generator or allowed their separate generators to benefit from sharing common

[3]Note that this is a strict subset of graphical games, where payoffs for each player also depend only on the actions of its neighbors, but it is not assumed that payoffs have the additive decomposition.

Arms Race	Grab the Dollar	Polymatrix Game
Battle of the Sexes	Graphical Game	Prisoner's Dilemma
Bertrand Oligopoly	Greedy Game	Random Games
Bidirectional LEG	Guess 2/3 Average	Rapoport's Distribution
Chicken	Hawk and Dove	Rock, Paper, Scissors
Collaboration Game	Local-Effect Game	Shapley's Game
Compound Game	Location Game	Simple Inspection Game
Congestion Game	Majority Voting	Traveler's Dilemma
Coordination Game	Matching Pennies	Uniform LEG
Cournot Duopoly	Minimum Effort Game	War of Attrition
Covariant Game	N-Player Chicken	Zero Sum Game
Dispersion Game	N-Player Pris Dilemma	

Table 6.1: Game Generators in GAMUT.

algorithms and data structures. In the end we built 35 parameterized generators to support all of the generative sets in our taxonomy; these are listed in Table 6.1.

The process of writing generators presented us with a nontrivial software engineering task in creating a coherent and easily-extensible software framework. Once the framework was in place, incrementally adding new generators became easy. Some of these implementation details are described in Section 6.4.

6.3 Running the GAMUT

In Section 6.1 we stated that it is necessary to evaluate game-theoretic algorithms on a wide range of distributions before empirical claims can be made about the algorithms' strengths and weaknesses. Of course, such a claim can only be substantiated after a test suite has been constructed. In this section we show that top algorithms for two computational problems in game theory do indeed exhibit dramatic variation across distributions, implying that small performance tests would be unreliable.

All our experiments were performed using a cluster of 12 dual-CPU 2.4GHz Xeon machines running Linux 2.4.20, and took about 120 CPU-days to run. We capped runs for all algorithms at 30 minutes (1800 seconds).

6.3.1 Computation of Nash Equilibria

The first problem that we used for validation is that of computing Nash equilibria (see Section 5.3.1 of Chapter 5). In this section we use GAMUT to evaluate three algorithms' empirical properties on this problem.

Experimental Setup

The best-known game theory software package is Gambit [McKelvey et al. 1992], a collection of state-of-the-art algorithms. We used Gambit's implementation of the Lemke-Howson algorithm (LH) [Lemke and Howson 1964] or two-player games, and its implementation of Simplicial Subdivision (SD) [van der Laan et al. 1987] for n-player games. In both cases, Gambit performs iterative removal of dominated strategies as a preprocessing step. For n-player games we have also considered the continuation method introduced by Govindan and Wilson (GW) [Govindan and Wilson 2003]. We used a recent optimized implementation, the GameTracer package [Blum et al. 2003]. GameTracer also includes speedups for the Govindan-Wilson algorithm on the special cases of compact graphical games and MAIDs, but because we expanded all games to their full normal forms Govindan-Wilson did not benefit from these extensions in our experiments.

One factor that can have a significant effect on an algorithm's runtime is the size of its input. Since our goal was to investigate the extent to which runtimes vary as the result of differences between distributions, we studied fixed-size games. To make sure that our findings were not artifacts of any particular problem size we compared results across several fixed problem sizes. We ran the Lemke-Howson algorithm on games with 2 players, 150 actions and 2 players, 300 actions. Because Govindan-Wilson is very similar to Lemke-Howson on two-player games and is not optimized for this case [Blum et al. 2003], we did not run it on these games. We ran Govindan-Wilson and Simplicial Subdivision on games with 6 players, 5 actions and 18 players, 2 actions. For each problem size and distribution, we generated 100 games.

Both to keep our machine-time demands manageable and to keep the graphs in this chapter from getting too cluttered, we chose not to use *all* of the GAMUT generators. Instead, we chose a representative slate of 22 distributions from GAMUT. Some of our generators (*e.g.,* Graphical Games, Polymatrix games, and Local Effect Games – LEGs) are parameterized by graph structure; we split these into several sub-distributions based on the kind of graph used. Suffixes "-CG", "-RG", "-SG",

"-SW" and "-Road" indicate, respectively, complete, random, star-shaped, small-world, and road-shaped (see [Vickrey and Koller 2002]) graphs. Another distribution that we decided to split was the Covariant Game distribution, which implements the random game model of [Rinott and Scarsini 2000]. In this distribution, payoffs for each outcome are generated from a multivariate normal distribution, with correlation between all pairs of players held at some constant ρ. With $\rho = 1$ these games are common-payoff, while $\rho = \frac{-1}{n-1}$ yields minimum correlation and leads to zero-sum games in the two-player case. Rinott and Scarsini show that the probability of the existence of a pure strategy Nash equilibrium in these games varies as a monotonic function of ρ, which makes the games computationally interesting. For these games, suffixes "-Pos", "-Zero", and "-Rand" indicate whether ρ was held at 0.9, 0, or drawn uniformly at random from $[\frac{-1}{n-1}, 1]$. Chapter 7 has more intuitions as to why these games are particularly interesting.

As mentioned in Chapter 5, Lemke-Howson, Simplicial Subdivision and Govindan-Wilson are all very complicated path-following numerical algorithms that offer virtually no theoretical guarantees. They all have worst-case running times that are at least exponential, but it is not known whether this bound is tight. On the empirical side, very little previous work has attempted to evaluate these algorithms. The best-known empirical results [McKelvey and McLennan 1996; von Stengel 2002] were obtained for generic games with payoffs drawn independently uniformly at random (in GAMUT, this would be the `RandomGame` generator). The work on GAMUT may therefore have been the first systematic attempt to understand the empirical behavior of these algorithms on non-generic games.

Experimental Results

Figure 6.3 shows each algorithm's performance across distributions for two different input sizes. The Y-axis shows CPU time measured in seconds and plotted on a log scale. Column height indicates median runtime over 100 instances, with the error bars showing the 25th and 75th percentiles. The most important thing to note about this graph is that each algorithm exhibits highly variable behavior across our distributions. This is less visible for the Govindan-Wilson algorithm on 18-player games,

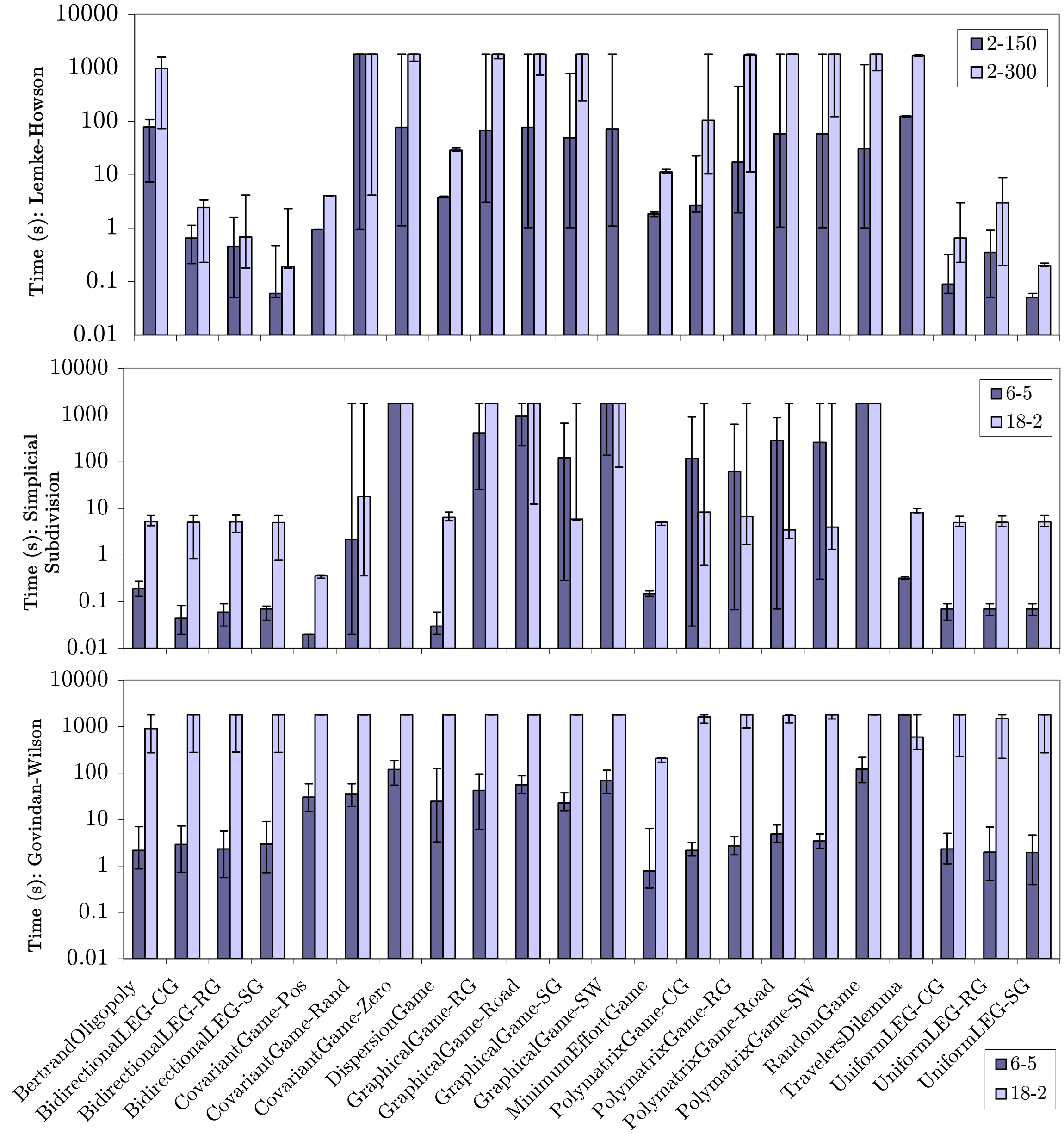


Figure 6.3: Effect of Problem Size on Solver Performance.

only because this algorithm's runtime exceeds our cap for a majority of the problems. However, even on this dataset the error bars demonstrate that the distribution of runtimes varies substantially with the distribution of instances. Moreover, for all three algorithms, we observe that this variation is not an artifact of one particular problem size. Thus, GAMUT distributions are indeed structurally different. This result is even more surprising since, on the face of it, none of the three algorithms are directly affected by the structure of the game.

Figure 6.4 illustrates runtime differences both across and among distributions for 6-player 5-action games. (We observed very similar results for different input sizes and for the Lemke-Howson algorithm; we do not include those here as they don't yield qualitatively new information.) Each dot on the graph corresponds to a single run of an algorithm on a game. This graph shows that the distribution of algorithm runtimes varies substantially from one distribution to another, and cannot easily be inferred from 25th/50th/75th quartile figures such as Figure 6.3. The highly similar Simplicial Subdivision runtimes for Traveler's Dilemma and Minimum Effort Games are explained by the fact that these games can be solved by iterated elimination of dominated strategies — a step not performed by the GameTracer implementation of Govindan-Wilson. We note that distributions that are related to each other in our taxonomy (*e.g.,* all kinds of Graphical Games, LEGs, or Polymatrix Games) usually give rise to similar — but not identical — algorithmic behavior. The fact that these related distributions are not identical implies that the underlying graph structure does carry influence on the algorithms, even though the games are represented explicitly in normal form. We conjecture that for games of larger sizes, as the differences among different graph structures become more pronounced, the differences in runtimes among these distributions will also increase.

Figure 6.4 makes it clear that algorithms' runtimes exhibit substantial variation and that algorithms often perform very differently on the same distributions. However, this figure makes it difficult for us to reach conclusions about the extent to which the algorithms are correlated. For an answer to this question, we turn to Figure 6.5. Each data point represents a single 6-player, 5-action game instance, with the X-axis representing runtime for Simplicial Subdivision and the Y-axis for Govindan-Wilson.

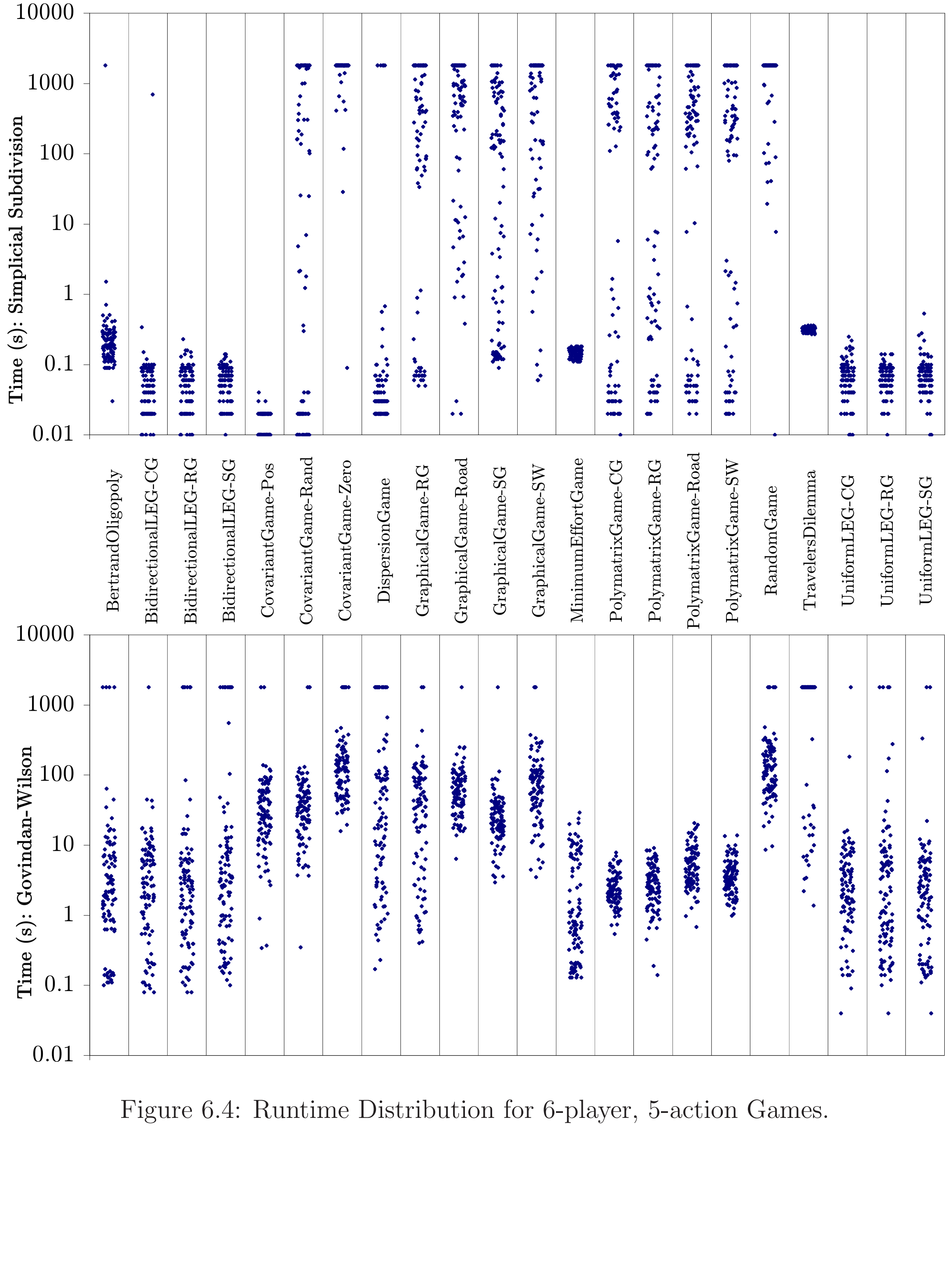


Figure 6.4: Runtime Distribution for 6-player, 5-action Games.

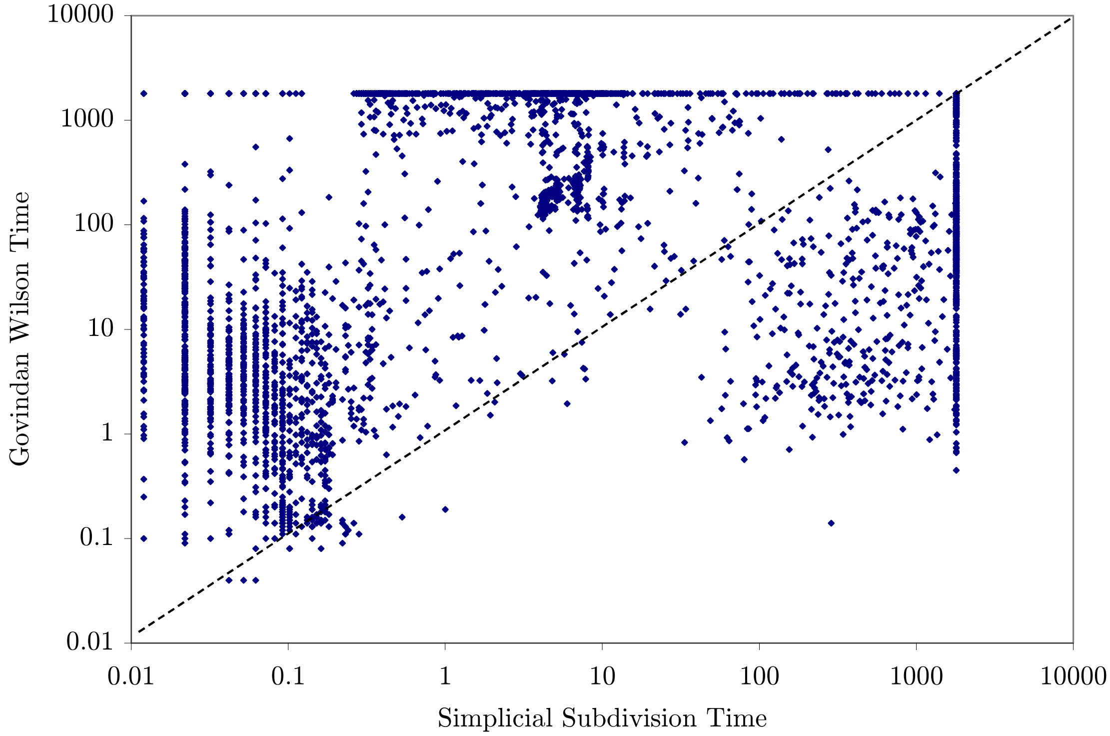


Figure 6.5: Govindan-Wilson Runtimes vs. Simplicial Subdivision Runtimes, 6-player, 5-action.

Both axes use a log scale. This figure shows that when we focus on instances rather than on distributions, these algorithms are very highly uncorrelated. Simplicial Subdivision does strictly better on 67.2% of the instances, while timing out on 24.7%. Govindan-Wilson wins on 24.7% and times out on 36.5%. It is interesting to note that if a game is easy for Simplicial Subdivision, then it will often be harder for Govindan-Wilson, but in general neither algorithm dominates.

The fact that algorithms' runtimes appear to be largely uncorrelated implies that some benefit could be derived in combining these algorithms into a portfolio, in the spirit of the results in Section 3.7 in Chapter 3. Figure 6.6 provides a closer look at the optimal portfolio, with the view similar to that of Figure 6.3. We see that indeed, a portfolio of algorithms could improve average runtimes on some, though not all, distributions. However, the effect is not as dramatic as in Chapter 3.

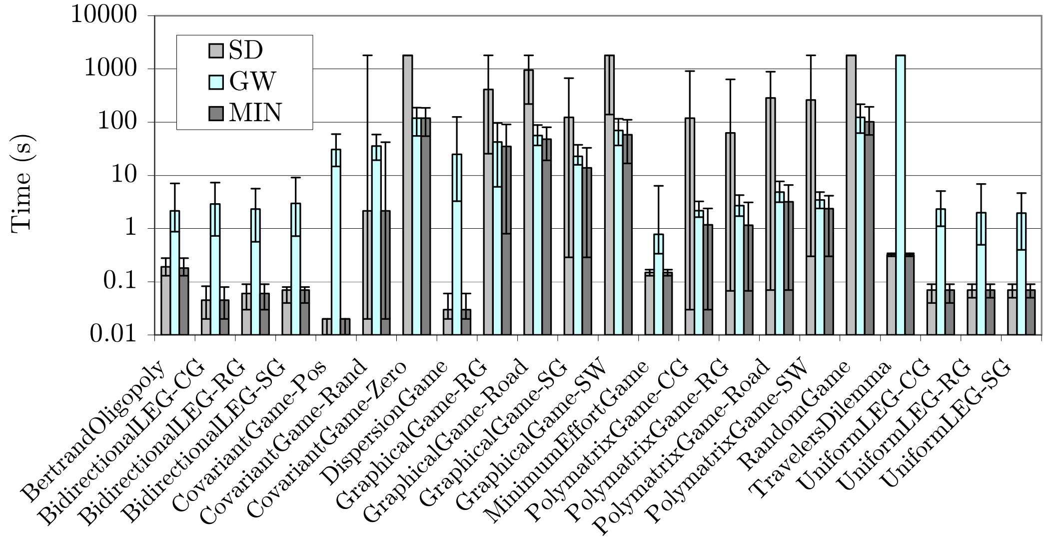


Figure 6.6: Optimal Portfolio, 6-player, 5-action.

6.3.2 Multiagent Learning in Repeated Games

Results in the previous section convincingly demonstrate the need for GAMUT; we now turn to another problem of computational game theory, namely, multiagent learning (see Chapter 5), in order to demonstrate that this variation across distributions is not an artefact of any particular domain. This research area is still at a very early stage, particularly with respect to the identification of the best metrics and standards of performance to use for evaluating algorithms. As a result, we do not claim that our results demonstrate anything about the relative merit of the algorithms we study. We simply demonstrate that these algorithms' performance depends crucially on the distributions of games on which they are run.

Experimental Setup

We used three learning algorithms: Minimax-Q [Littman 1994], WoLF [Bowling and Veloso 2001], and single-agent Q-learning [Watkins and Dayan 1992] described in Section 5.3.2 of Chapter 5. Previous work in the literature has established that each of these algorithms is very sensitive to its parameter settings (*e.g.,* learning rate) and that the best parameter settings usually vary from one game to the next. Since it is infeasible to perform per-game parameter tuning in an experiment involving tens of thousands of games, we determined parameter values that reproduced previously-published results from [Littman 1994; Bowling and Veloso 2001; Watkins and Dayan 1992] and then fixed these parameters for all experiments.

In our experiments we chose to focus on a set of 13 distributions. As before, we kept game sizes constant, this time at 2 actions and 2 players for each game. Although it would also be interesting to study performance in larger games, we decided to focus on a simpler setting in which it would be easier to understand the results of our experiments. Indeed, even with this small game size, we observed that different distributions gave rise to qualitatively different results. For each distribution we generated 100 game instances. For each instance we performed nine different pairings (each possible pairing of the three algorithms, including self-pairings, and in the case of non-self-pairings also allowing each algorithm to play once as player 1 and once as player 2). We ran the algorithms on each pairing ten times, since we found that algorithm performance varied based on their random number generators. On each run, we repeated the game 100,000 times. The first 90,000 rounds allowed the algorithms to settle into their long-run behavior; we then computed each algorithm's payoff for each game as its average payoff over the following 10,000 rounds. We did this to approximate the offline performance of the learned policy and to minimize the effect of relative differences in the algorithms' learning rates.

Experimental Results

There are numerous ways in which learning algorithms can be evaluated. In this section we focus on just two of them. A more comprehensive set of experiments would be required to judge the relative merits of algorithms, but this smaller set of experiments is sufficient to substantiate our claim that algorithm performance varies significantly from one distribution to another.

Figure 6.7 compares the pairwise performance of three algorithms. The height of a bar along the Y-axis indicates the (normalized) fraction of games in which the corresponding algorithm received a weakly greater payoff than its opponent. In this metric we ignore the magnitude of payoffs, since in general they are incomparable

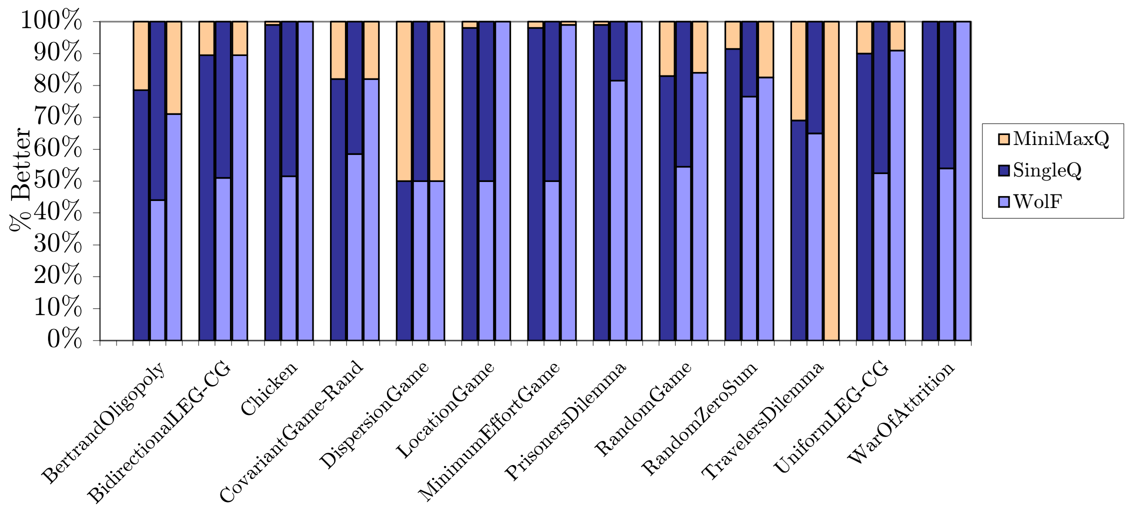


Figure 6.7: Pairwise Comparison of Algorithms.

across games: even though we normalized payoffs to lie in the $[-1, 1]$ interval, things like payoff in the best equilibrium, or the ratio of payoffs between best and second best outcome are not constant across distributions (and, to a lesser extent, within a distribution).

The overall conclusion that we can draw from Figure 6.7 is that there is great variation in the relative performance of algorithms across distributions. There is no clear "winner"; even Minimax-Q, which is usually outperformed by WoLF, manages to win a significant fraction of games across many distributions, and *dominates* it on Traveler's Dilemma. WoLF and Single-Agent-Q come within 10% of a tie most of the time — suggesting that these algorithms often converge to the same equilibria — but their performance is still far from consistent across different distributions.

Figure 6.8 compares algorithms using a different metric. Here the Y-axis indicates the average payoff for an algorithm when playing as player 1, with column heights indicating the median and error bars indicating 25th and 75th percentiles. Since all algorithms we ran against everybody else, including self-play, payoffs can be thought as averaging across the fixed population of three opponents. Payoffs are normalized to fall on the range $[-1, 1]$. Despite this normalization, it is difficult to make meaningful comparisons of payoff values across distributions. This graph is interesting because,

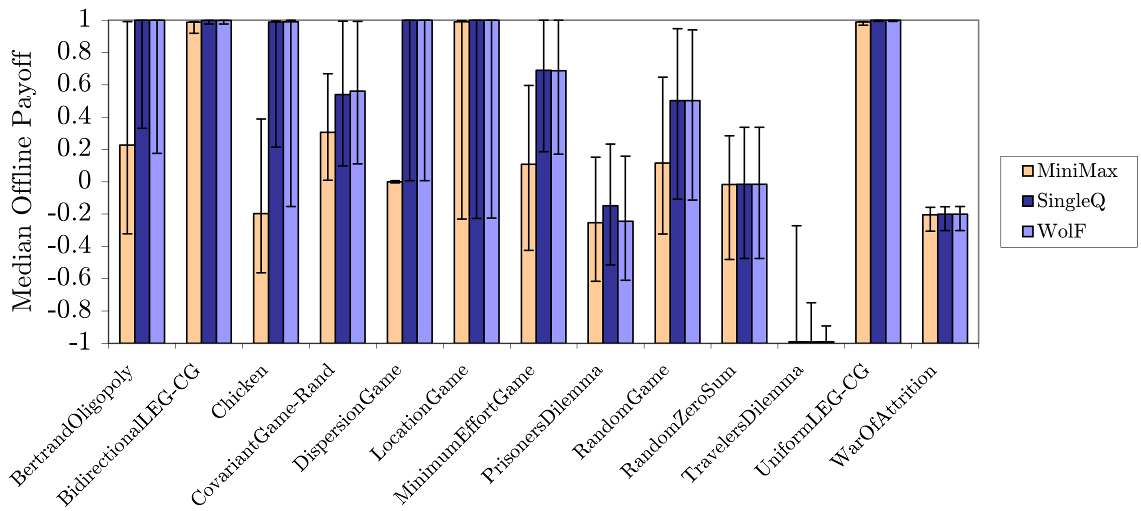


Figure 6.8: Median Payoffs as Player 1.

while focusing on relative performance rather than trying to identify a "winning" algorithm, it demonstrates again that the algorithms' performance varies substantially along the GAMUT. Moreover, this metric shows Minimax-Q to be much more competitive than was suggested by Figure 6.7.

6.4 GAMUT Implementation Notes

The GAMUT software was built as an object-oriented framework and implemented in Java[4]. Our framework consists of objects in four basic categories: game generators, graphs, functions, and representations. Our main design objective was to make it as easy as possible for end users to write new objects of any of the four kinds, in order to allow GAMUT to be extended to support new sets of games and representations.

Currently, GAMUT contains 35 implementations of `Game` objects, which correspond to the 35 procedures we identified in Section 6.2.2. They are listed in Table 6.1. While the internal representations and algorithms used vary depending on the set of games being generated, all of them must be able to return the number of players, the number of actions for each player, and the payoff for a each player for

[4]See `http://gamut.stanford.edu` for detailed software documentation.

GAMUT Graph Classes:

Barabasi-Albert PLOD	Power-Law Out-Degree
Complete Graph	Random Graph
N-Ary Tree	Ring Graph
N-Dimensional Grid	Small World Graph
N-Dimensional Wrapped Grid	Star Graph

GAMUT Function Classes:

Exponential Function	Polynomial Function
Log Function	Table Function
Decreasing Wrapper	Increasing Polynomial

GAMUT Outputter Classes:

Complete Representations	Incomplete Representations
Default GAMUT Payoff List	Local-Effect Form
Extensive Form	Two-Player Readable Matrix Form
Gambit Normal Form	
Game Tracer Normal Form	
Graphical Form	

Table 6.2: GAMUT Support Classes.

any action profile. `Outputter` classes then encode generated games into appropriate representations.

Many of our generators depend on random graphs (*e.g.,* Graphical Games, Local Effect Games, Polymatrix Games) and functions (*e.g.,* Arms Race, Bertrand Oligopoly, Congestion Games). `Graph` and `Function` classes, listed in Table 6.2, have been implemented to meet these needs in a modular way. As with games, additional classes of functions and graphs can be easily added.

`Outputter` classes encapsulate the notion of a representation. GAMUT allows for representations to be incomplete and to work only with compatible generators; however, most output representations work with all game generators. Table 6.2 lists the complete and incomplete representations that are currently supported by GAMUT.

In keeping with our main goal of easy extensibility, GAMUT also implements a wide range of support classes that encapsulate common tasks. For example, GAMUT uses a powerful parameter handling mechanism. Users who want to create a new generator can specify types, ranges, default values and help strings for parameters. Given this information, user help, parsing, and even randomization will all be handled automatically. Since a large (and mundane) part of the user's job now becomes declarative, it is easy to focus on the more interesting and conceptual task of implementing the actual generative algorithm.

Other support utilities offer the ability to convert games into fixed-point arithmetic and to normalize payoffs. The former, besides often being more efficient, sometimes makes more sense game-theoretically: the notion of a Nash equilibrium can become muddy with floating point, since imprecision can lead to equilibrium instability. As mentioned in Section 6.2.2, games' strategic properties are preserved under positive affine transformations. Normalization allows payoff magnitudes to be compared and can help to avoid machine precision problems.

6.5 Conclusion

GAMUT, a comprehensive game-theoretic test-suit, is interesting in its own right. It was built based on a thorough survey of hundreds of books and papers in game-theoretic literature. GAMUT includes a comprehensive database of structured non-generic games and the relationships between them. It is built as a highly modular and extensible software framework, which is used to implement generators for the sets of games in its database. Experimental results in this chapter demonstrated the importance of comprehensive test data to game-theoretic algorithms by showing how performance depends crucially on the distribution of instances on which an algorithm is run. GAMUT is rapidly becoming an indispensable tool for researchers working at the intersection of game theory and computer science.

More importantly for the purposes of this thesis, GAMUT now provides us with a definitive *distribution* of normal-form games that we will study in more detail in the next chapter.

Chapter 7

Simple Search Methods For Finding A Nash Equilibrium

Now that we have an interesting instance distribution (GAMUT), we need a tool that allows us to examine games in GAMUT in more detail. As described in Chapter 5, despite several decades of research into the problem of finding a sample Nash equilibrium of a normal-form game, it remains thorny; its precise computational complexity is unknown, and new algorithms have been relatively few and far between. In this chapter we present two very simple search methods for computing a sample Nash equilibrium in a normal-form game: one for 2-player games and one for n-player games. The work presented in this chapter makes two related contributions to the problem, both bringing to bear fundamental lessons from computer science. In a nutshell, they are:

- The use of heuristic search techniques in algorithms.
- The use of an extensive test suite to evaluate algorithms.

The surprising result of applying these insights is that, for games in GAMUT, even very simple heuristic methods are quite effective, and significantly outperform the state of the art Lemke-Howson, Simplicial Subdivision and Govindan-Wilson algorithms. More importantly, their simplicity allows us to pinpoint exactly why they can be so fast, allowing us to draw an interesting conclusion about games in GAMUT.

7.1 Algorithm Development

In the developments of novel algorithms, one can identify two extremes. The first extreme is to gain deep insight into the structure of the problem, and craft highly specialized algorithms based on this insight. The other extreme is to identify relatively shallow heuristics, and hope that these, coupled with the ever increasing computing power, are sufficient. While the first extreme is certainly more appealing, the latter often holds the upper hand in practice.[1] For example, a deep understanding of linear programming yielded polynomial-time interior point methods [Karmarkar 1984], though in practice one tends to use methods based on the exponential Simplex algorithm [Wood and Dantzig 1949]. In reality, neither extreme is common. It is more usual to find a baseline algorithm that contains many choice points based on some amount of insight into the problem, and then apply heuristics to making these choices. For example, for the problem of SAT (Chapter 4), the current state of the art solvers apply heuristics to search the space spanned by the underlying Davis-Putnam procedure [Cook and Mitchell 1997].

The use of heuristics, while essential, raises the question of algorithm evaluation. The gold standard for algorithms is trifold: soundness, completeness, and low computational complexity. An algorithm is sound if any proposed solution that it returns is in fact a solution, and it is complete if, whenever at least one solution exists, the algorithm finds one. Low computational complexity is generally taken to mean that the worst-case running time is polynomial in the size of the input.[2]

Of course, in many cases, it is not possible (or has turned out to be extremely difficult) to achieve all three simultaneously. This is particularly true when one uses

[1] Of course, there is no difference in kind between the two extremes. To be effective, heuristics too must embody some insight into the problem. However, this insight tends to be limited and local, yielding a rule of thumb that aids in guiding the search through the space of possible solutions or algorithms, but does not directly yield a solution.

[2] In rare cases this is not sufficient. For example, in the case of linear programming, Khachiyan's ellipsoid method [Khachiyan 1979] was the first polynomial-time algorithm, but, in practice, it could not compete with either the existing, exponential simplex method [Wood and Dantzig 1949], or polynomial interior point methods [Karmarkar 1984] that would later be developed. However, this will not concern us here, since there is no known worst-case polynomial-time complexity algorithm for the problem in question — computing a sample Nash equilibrium — and there are strong suspicions that one does not exist.

heuristic methods. In this chapter we focus on approaches that sacrifice the third goal, that of low worst-case complexity. Without a worst-case guarantee (and even worse, when one knows that the running time is exponential in the worst case), an algorithm must be evaluated using empirical tests, in which case the choice of problem distributions on which it is tested becomes critical. This is another important lesson from computer science—one should spend considerable effort devising an appropriate test suite, one that faithfully mirrors the domain in which the algorithms will be applied. The complexity sacrifice of our heuristic methods is not in vain, however. Their simplicity allows for an easy and intuitive characterization of the kinds of games on which they can perform well. In that sense, these algorithms will become magnifying lenses, through which we'll see a glimpse of structure in games from GAMUT.

With these computer science lessons in mind, let us recap (see Chapter 5) the extant game theoretic literature on computing a sample Nash equilibrium. Algorithm development has clearly been of the first kind, namely exploiting insights into the structure of the problem. For 2-player games, the problem can be formulated as a linear complementarity problem (LCP). The Lemke-Howson algorithm [Lemke and Howson 1964] is based on a general method for solving LCPs. Despite its age, this algorithm has remained the state of the art for 2-player games. For n-player games, the best existing algorithms are Simplicial Subdivision [van der Laan et al. 1987] and Govindan-Wilson[Govindan and Wilson 2003]. Each of these algorithms is based on non-trivial insights into the mathematical structure of the problem.

From the evaluation point of view, these algorithms are all sound and complete, but not of low complexity. Specifically, they all have a provably exponential worst-case running time. The existing literature does not provide a useful measure of their empirical performance, because most tests are only on so-called "random" games, in which every payoff is drawn independently from a uniform distribution. This is despite the fact that this distribution is widely recognized to have rather specific properties that are not representative of problem domains of interest.

Work described in this chapter diverges from the traditional game-theoretic approach in two fundamental ways: (1) we propose new algorithms that are based on (relatively) shallow heuristics, and (2) we test our algorithms, along with existing ones, extensively through the use of GAMUT, a comprehensive testbed. The result is a pair of algorithms (one for 2-player games, and another for n-player games, for $n > 2$) that we show to perform very well in practice. Specifically, they outperform previous algorithms, Lemke-Howson on 2-player games, and Simplicial Subdivision and Govindan-Wilson on n-player games, sometimes dramatically.

The basic idea behind our two search algorithms is simple. Recall that the general problem of computing a Nash equilibrium is a complementarity problem. It turns out that the source of the complementarity (orthogonality) constraint lies in fixing *supports* for each player; basically, an action that is played with non-zero probability must yield a certain payoff. As a consequence, computing whether there exists a NE with a *particular support* for each player is a relatively easy feasibility program. Our algorithms explore the space of support profiles using a backtracking procedure to instantiate the support for each player separately. After each instantiation, they prune the search space by checking for actions in a support that are strictly dominated, given that the other agents will only play actions in their own supports.

Both of the algorithms are biased towards simple solutions through their preference for small supports. Our surprising discovery was that the games drawn from classes that researchers have focused on in the past (*i.e.,* those in GAMUT) tend to have (at least one) "simple" Nash equilibrium; hence, our algorithms are often able to find one quickly. Thus, utilizing these very simple heuristic approaches allows to get a better feel for the properties of equilibria in games of interest.

The rest of this chapter is organized as follows. First, we formulate the basis for searching over supports. We then define our two algorithms. The n-player algorithm is essentially a generalization of the 2-player algorithm, but we describe them separately, both because they differ slightly in the ordering of the search, and because the 2-player case admits a simpler description of the algorithm. Then, we describe our experimental setup, and separately present our results for 2-player and n-player games. We then describe the nature of the equilibria that are found by different algorithms on our data.

7.2 Searching Over Supports

The basis of our two algorithms is to search over the space of possible instantiations of the support $S_i \subseteq A_i$ for each player i. Given a support profile $S = (S_1, \ldots, S_n)$ as input, Feasibility Program 1, below, gives the formal description of a program for finding a Nash equilibrium p consistent with S (if such an strategy profile exists). In this program, v_i corresponds to the expected utility of player i in an equilibrium. The first two classes of constraints require that each player must be indifferent among all actions within his support, and must not strictly prefer an action outside of his support. These imply that no player can deviate to a pure strategy that improves his expected utility, which is exactly the condition for the strategy profile to be a Nash equilibrium.

Because $p(a_{-i}) = \prod_{j \neq i} p_j(a_j)$, this program is linear for $n = 2$ and nonlinear (in fact, multilinear) for all $n > 2$. Note that, strictly speaking, we do not require that each action $a_i \in S_i$ be in the support, because it is allowed to be played with zero probability. However, player i must still be indifferent between action a_i and each other action $a'_i \in S_i$; thus, simply plugging in $S_i = A_i$ would not necessarily yield a Nash equilibrium as a solution.

Feasibility Program 1

Input: $S = (S_1, \ldots, S_n)$, a support profile
Output: NE p, if there exists both a strategy profile $p = (p_1, \ldots, p_n)$ and a value profile $v = (v_1, \ldots, v_n)$ such that:

$$
\begin{aligned}
\forall i \in N, a_i \in S_i : &\quad \sum_{a_{-i} \in S_{-i}} p(a_{-i}) u_i(a_i, a_{-i}) = v_i \\
\forall i \in N, a_i \notin S_i : &\quad \sum_{a_{-i} \in S_{-i}} p(a_{-i}) u_i(a_i, a_{-i}) \leq v_i \\
\forall i \in N : &\quad \sum_{a_i \in S_i} p_i(a_i) = 1 \\
\forall i \in N, a_i \in S_i : &\quad p_i(a_i) \geq 0 \\
\forall i \in N, a_i \notin S_i : &\quad p_i(a_i) = 0
\end{aligned}
$$

7.3 Algorithm for Two-Player Games

In this section we describe Algorithm 1, our 2-player algorithm for searching the space of supports. There are three keys to the efficiency of this algorithm. The first two are the factors used to order the search space. Specifically, Algorithm 1 considers every possible support size profile separately, favoring support sizes that are balanced and small. The motivation behind these choices comes from work such as [McLennan and Berg 2002], which analyzes the theoretical properties of the Nash equilibria of games drawn from a particular distribution. Specifically, for n-player games, the payoffs for an action profile are determined by drawing a point uniformly at random in a unit sphere. Under this distribution, for $n = 2$, the probability that there exists a NE consistent with a particular support profile varies inversely with the size of the supports, and is zero for unbalanced support profiles.

The third key to Algorithm 1 is that it separately instantiates each players' support, making use of what we will call "conditional (strict) dominance" to prune the search space.

Definition 7.1 *An action $a_i \in A_i$ is* **conditionally dominated***, given a profile of sets of available actions $R_{-i} \subseteq A_{-i}$ for the remaining agents, if the following condition holds:* $\exists a'_i \in A_i \ \forall a_{-i} \in R_{-i} : \quad u_i(a_i, a_{-i}) < u_i(a'_i, a_{-i})$

Observe, that this definition is strict, because, in a Nash Equilibrium, no action that is played with positive probability can be conditionally dominated given the actions in the support of the opponents' strategies.

The preference for small support sizes amplifies the advantages of checking for conditional dominance. For example, after instantiating a support of size two for the first player, it will often be the case that many of the second player's actions are pruned, because only two inequalities must hold for one action to conditionally dominate another.

Pseudo-code for Algorithm 1 is given below. Note that this algorithm is complete, because it considers all support size profiles, and because it only prunes actions that are *strictly* dominated.

Algorithm 1

```
for all support size profiles x = (x_1, x_2), sorted in increasing order of, first, |x_1 − x_2|
and, second, (x_1 + x_2) do
  for all S_1 ⊆ A_1 s.t. |S_1| = x_1 do
    A'_2 ← {a_2 ∈ A_2 not conditionally dominated, given S_1 }
    if ∄a_1 ∈ S_1 conditionally dominated, given A'_2 then
      for all S_2 ⊆ A'_2 s.t. |S_2| = x_2 do
        if ∄a_1 ∈ S_1 conditionally dominated, given S_2 then
          if Feasibility Program 1 is satisfiable for S = (S_1, S_2) then
            Return the found NE p
```

7.4 Algorithm for N-Player Games

Algorithm 1 can be interpreted as using the general backtracking algorithm to solve a constraint satisfaction problem (CSP) for each support size profile (for an introduction to CSPs, see, for example, Dechter [2003]). The variables in each CSP are the supports S_i, and the domain of each S_i is the set of supports of size x_i. While the single constraint is that there must exist a solution to Feasibility Program 1, an extraneous, but easier to check, set of constraints is that no agent plays a conditionally dominated action. The removal of conditionally dominated strategies by Algorithm 1 is similar to using the AC-1 algorithm to enforce arc-consistency with respect to these constraints. We use this interpretation to generalize Algorithm 1 for the n-player case. Pseudocode for Algorithm 2 and its two procedures, Recursive-Backtracking and Iterated Removal of Strictly Dominated Strategies (IRSDS) are given below[3].

IRSDS takes as input a domain for each player's support. For each agent whose support has been instantiated, the domain contains only that instantiated support, while for each other agent i it contains all supports of size x_i that were not eliminated in a previous call to this procedure. On each pass of the *repeat-until* loop, every action found in at least one support in a player's domain is checked for conditional domination. If a domain becomes empty after the removal of a conditionally dominated

[3] Even though our implementation of the backtracking procedure is iterative, for simplicity we present it here in its equivalent, recursive form. Also, the reader familiar with CSPs will recognize that we have employed very basic algorithms for backtracking and for enforcing arc consistency; we return to this point in the conclusion.

action, then the current instantiations of the Recursive-Backtracking are inconsistent, and IRSDS returns *failure*. Because the removal of an action can lead to further domain reductions for other agents, IRSDS repeats until it either returns *failure* or iterates through all actions of all players without finding a dominated action.

Algorithm 2

---

for all $x = (x_1, \ldots, x_n)$, sorted in increasing order of, first, $\sum_i x_i$ and, second, $max_{i,j}(x_i - x_j)$ **do**
  $\forall i : S_i \leftarrow NULL$ *//uninstantiated supports*
  $\forall i : D_i \leftarrow \{S_i \subseteq A_i : |S_i| = x_i\}$ *//domain of supports*
  if Recursive-Backtracking$(S, D, 1)$ returns a NE p **then**
    Return p

Procedure 1 Recursive-Backtracking.

Input: $S = (S_1, \ldots, S_n)$: a profile of supports
    $D = (D_1, \ldots, D_n)$: a profile of domains
    i: index of next support to instantiate
Output: A Nash equilibrium p, or $failure$
if $i = n + 1$ **then**
  if Feasibility Program 1 is satisfiable for S **then**
    Return the found NE p
  else
    Return $failure$
else
  for all $d_i \in D_i$ **do**
    $S_i \leftarrow d_i$
    $D_i \leftarrow D_i - \{d_i\}$
    if IRSDS$((\{S_1\}, \ldots, \{S_i\}, D_{i+1}, \ldots, D_n))$ succeeds **then**
      if Recursive-Backtracking$(S, D, i + 1)$ returns NE p **then**
        Return p
  Return $failure$

Finally, we note that Algorithm 2 is not a strict generalization of Algorithm 1, because it orders the support size profiles first by size, and then by a measure of balance. The reason for the change is that balance (while still significant) is less important for $n > 2$ than it is for $n = 2$. For example, under the model of McLennan

Procedure 2 Iterated Removal of Strictly Dominated Strategies (IRSDS).

Input: $D = (D_1, \ldots, D_n)$: profile of domains
Output: Updated domains, or $failure$
repeat
  $changed \leftarrow false$
  for all $i \in N$ **do**
    for all $a_i \in \bigcup_{d_i \in D_i} d_i$ **do**
      for all $a'_i \in A_i$ **do**
        if a_i is conditionally dominated by a'_i, given $\bigcup_{d_{-i} \in D_{-i}} d_{-i}$ **then**
          $D_i \leftarrow D_i - \{d_i \in D_i : a_i \in d_i\}$
          $changed \leftarrow true$
          if $D_i = \emptyset$ **then**
            Return $failure$
until $changed = false$
Return D

and Berg [2002], for $n > 2$, the probability of the existence of a NE consistent with a particular support profile is no longer zero when the support profile is unbalanced.[4]

7.5 Empirical Evaluation

It is very clear from the description of our algorithms that they can be quite inefficient: in the worst case they have to enumerate all possible supports before finding a NE, of which there are exponentially many. Therefore, a natural question to ask is whether this can every work well in practice. From the design of these algorithms, it is clear that there are only two possible ways in which this can happen. First, it might be possible that removing of conditionally dominant strategies does a lot of pruning, allowing the algorithms to zoom fast through lots of supports. This is unlikely, however, as even in such a case exponentially many supports have to be considered.

[4]While this change of ordering does provide substantial improvements, the algorithm could certainly still be improved by adding more complex heuristics. For example, McKelvey and McLennan [1997] show that, in a generic game, there cannot be a totally mixed NE if the size of one player's support exceeds the sum of the sizes of all other players' supports. However, we believe that handling cases such as this one would provide a relatively minor performance improvement, since our algorithm often finds a NE with small support.

The only alternative lies with the termination condition. Since we terminate as soon as the first equilibrium is found, these algorithms may be fast if most games actually have small balanced supports.

It is worth noting that, while the different distributions in GAMUT vary in their structure, some randomness must be injected into the generator in order to create a distribution over games of that structure. Our results then show that games drawn from GAMUT are similar to those generated uniformly at random in that they are likely to have a pure strategy equilibria, or equilibria with small balanced supports despite the imposed structure. Thus, the success of our algorithms is a reflection on the structure of games of interest to researchers as much as it is a demonstration of the techniques we employ.

7.5.1 Experimental Setup

To evaluate the performance of our algorithms we ran several sets of experiments on a representative subset of the distributions from GAMUT. To make figure labels readable, we use non-descriptive tags for each distribution. Table 7.1 provides a legend for these labels. Chapter 6 has more information about GAMUT.

A distribution of particular importance is the one most commonly tested on in previous work: D18, the "Uniformly Random Game", in which every payoff in the game is drawn independently from an identical uniform distribution. Also important are distributions D5, D6, and D7, which fall under a "Covariance Game" model studied by Rinott and Scarsini [2000], in which the payoffs for the n agents for each action profile are drawn from a multivariate normal distribution in which the covariance ρ between the payoffs of each pair of agents is identical. When $\rho = 1$, the game is common-payoff, while $\rho = \frac{-1}{N-1}$ yields minimal correlation, which occurs in zero-sum games. Thus, by altering ρ, we can smoothly transition between these two extreme classes of games.

Our experiments were executed on a cluster of 12 dual-processor, 2.4GHz Pentium machines, running Linux 2.4.20. We capped runs for all algorithms at 1800 seconds. When describing the statistics used to evaluate the algorithms, we will use

D1	Bertrand Oligopoly	D2	Bidirectional LEG, Complete Graph
D3	Bidirectional LEG, Random Graph	D4	Bidirectional LEG, Star Graph
D5	Covariance Game: $\rho = 0.9$	D6	Cov. Game: $\rho \in [-1/(N-1), 1]$
D7	Covariance Game: $\rho = 0$	D8	Dispersion Game
D9	Graphical Game, Random Graph	D10	Graphical Game, Road Graph
D11	Graphical Game, Star Graph	D12	Graphical Game, Small-World
D13	Minimum Effort Game	D14	Polymatrix Game, Complete Graph
D15	Polymatrix Game, Random Graph	D16	Polymatrix Game, Road Graph
D17	PolymatrixGame, Small-World	D18	Uniformly Random Game
D19	Travelers Dilemma	D20	Uniform LEG, Complete Graph
D21	Uniform LEG, Random Graph	D22	Uniform LEG, Star Graph
D23	Location Game	D24	War Of Attrition

Table 7.1: GAMUT Distribution Labels.

"unconditional" (on having solved the instance) to refer to the value of the statistic when timeouts are counted as 1800 seconds, and "conditional" to refer to its value excluding timeouts.

When $n = 2$, we solved Feasibility Program 1 using CPLEX 8.0's callable library [ILOG 2004][5]. For $n > 2$, because the program is nonlinear, we instead solved each instance of the program by executing AMPL, using MINOS [Murtagh and Saunders 2004] as the underlying optimization package. Obviously, we could substitute in any nonlinear solver; and, since a large fraction of our running time is spent on AMPL and MINOS, doing so would greatly affect the overall running time.

Before presenting the empirical results, we note that a comparison of the worst-case running times of our two algorithms and the three we tested against does not distinguish between them, since they all have exponential worst-case complexity.

[5]We note that the implementation of Lemke-Howson that we used employs a mathematical software that is less efficient than CPLEX, which undoubtedly has an effect on the running time comparisons between the two algorithms. However, the difference is not nearly great enough to explain the observed gap between the algorithms.

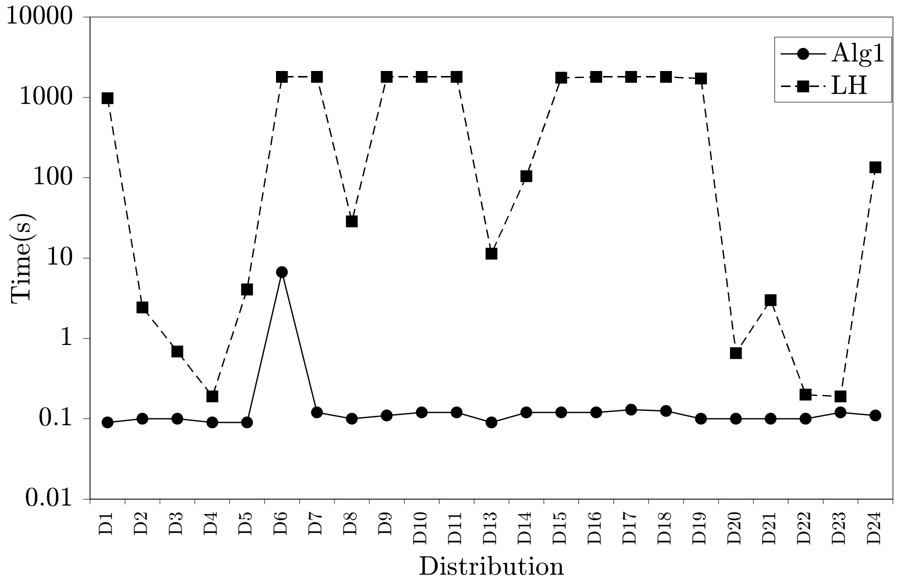


Figure 7.1: Unconditional Median Running Times for Algorithm 1 and Lemke-Howson on 2-player, 300-action Games.

7.5.2 Results for Two-Player Games

In the first set of experiments, we compared the performance of Algorithm 1 to that of Lemke-Howson (implemented in Gambit, which added the preprocessing step of iterated removal of weakly dominated strategies) on 2-player, 300-action games drawn from 24 of GAMUT's 2-player distributions. All algorithms were executed on 100 games drawn from each distribution. The time is measured in seconds and plotted on a logarithmic scale.

Figure 7.1 compares the unconditional median running times of the algorithms, and shows that Algorithm 1 outperforms Lemke-Howson on all distributions.[6] However, this does not tell the whole story. For many distributions, it simply reflects the fact that there is a greater than 50% chance that the distribution will generate a game with a pure strategy NE, which Algorithm 1 will then find quickly. Two other important statistics are the percentage of instances solved (Figure 7.2), and

[6]Obviously, the lines connecting data points across distributions for a particular algorithm are meaningless — they were only added to make the graph easier to read.

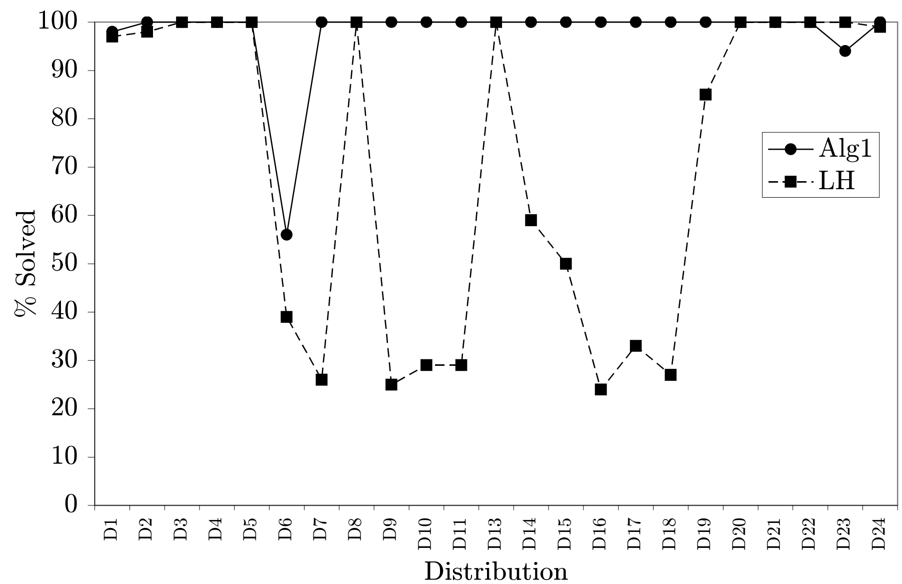


Figure 7.2: Percentage Solved by Algorithm 1 and Lemke-Howson on 2-player, 300-action Games.

the average running time conditional on solving the instance (Figure 7.3). Here, we see that Algorithm 1 completes far more instances than Lemke-Howson on several distributions, and solves fewer on just a single distribution (6 fewer, on D23).

Figure 7.3 further highlights the differences between the two algorithms. It demonstrates that even on distributions for which Algorithm 1 solves far more games, its conditional average running time is 1 to 2 orders of magnitude smaller than that of Lemke-Howson.

Clearly, the hardest distribution for both of the algorithms is D6, which consists of “Covariance Games” in which the covariance ρ is drawn uniformly at random from the range $[-1, 1]$. In fact, neither of the algorithms solved any of the games in another “Covariance Game” distribution in which $\rho = -0.9$, and these results were omitted from the graphs, because the conditional average is undefined for these results. On the other hand, for the distribution “CovarianceGame-Pos” (D5), in which $\rho = 0.9$, all algorithms perform well.

To further investigate this continuum, we sampled 300 values for ρ in the range

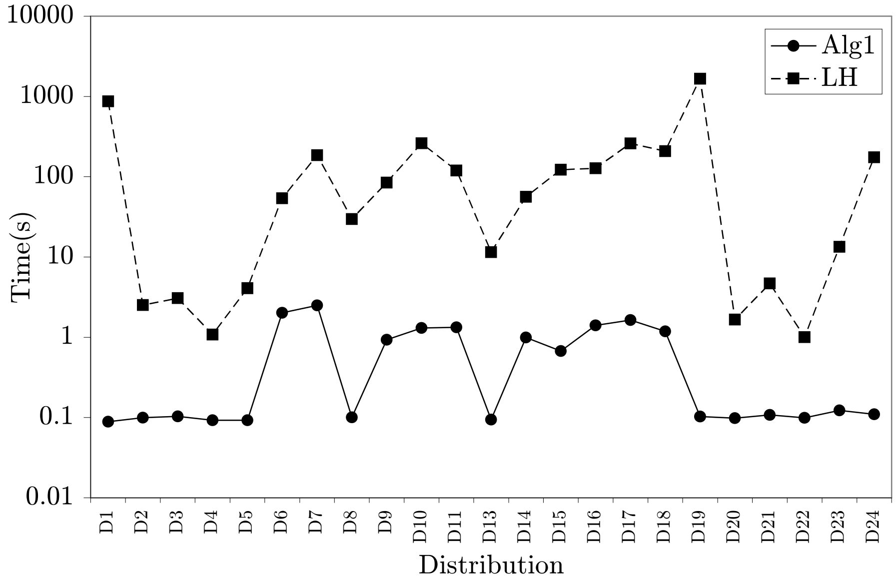


Figure 7.3: Average Running Time on Solved Instances for Algorithm 1 and Lemke-Howson on 2-player, 300-action Games.

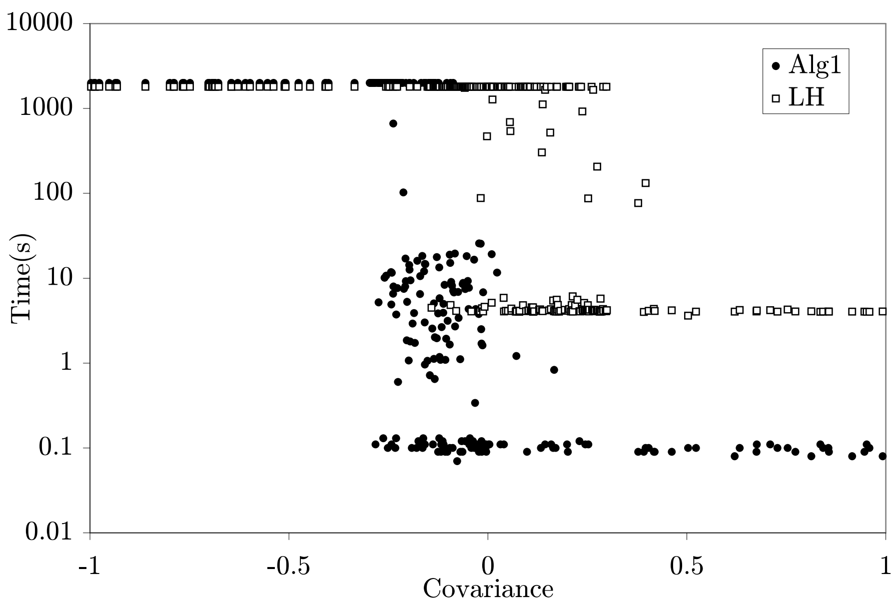


Figure 7.4: Running Time for Algorithm 1 and Lemke-Howson on 2-player, 300-action "Covariance Games".

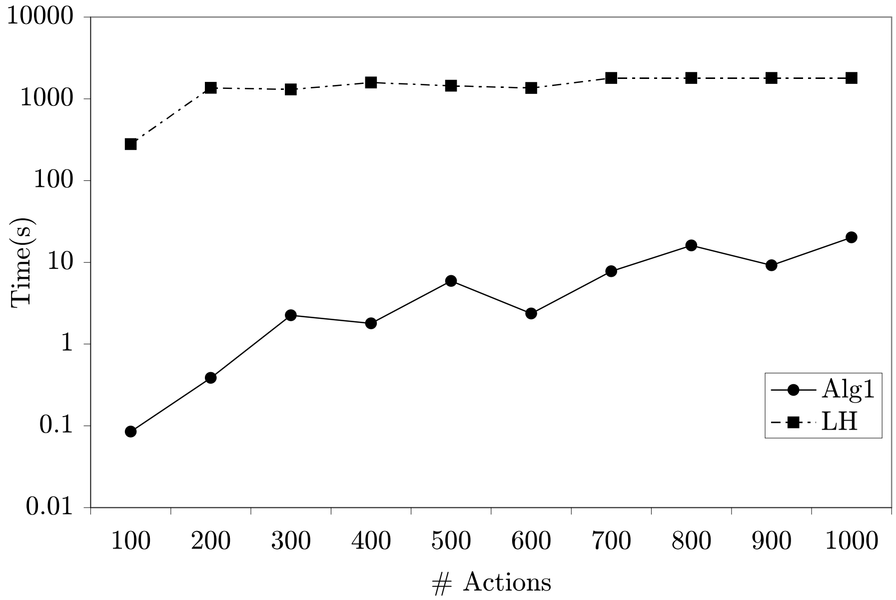


Figure 7.5: Unconditional Average Running Time for Algorithm 1 and Lemke-Howson on 2-player "Uniformly Random Games" vs. the Number of Actions.

$[-1, 1]$, with heavier sampling in the transition region and at zero. For each such game, we plotted a point for the running time of Algorithm 1, Lemke-Howson in Figure 7.4.[7]

The theoretical results of Rinott and Scarsini [2000] suggest that the games with lower covariance should be more difficult for Algorithm 1, because they are less likely to have a pure strategy NE. Nevertheless, it is interesting to note the sharpness of the transition that occurs in the $[-0.3, 0]$ interval. More surprisingly, a similarly sharp transition also occurs for Lemke-Howson, despite the fact that it operates in an unrelated way to Algorithm 1. It is also important to note that the transition region for Lemke-Howson is shifted to the right by approximately 0.3 relative to that of Algorithm 1, and that on instances in the easy region for both algorithms, Algorithm 1 is still an order of magnitude faster.

Finally, we explored the scaling behavior of all algorithms by generating 20 games from the "Uniformly Random Game" distribution (D18) for each multiple of 100 from

[7] The capped instances for Algorithm 1 were perturbed slightly upward on the graph for clarity.

100 to 1000 actions. Figure 7.5 presents the *unconditional* average running time, with a timeout counted as 1800s. While Lemke-Howson failed to solve any game with more than 600 actions and timed out on some 100-action games, Algorithm 1 solved all instances, and, without the help of cutoff times, still had an advantage of 2 orders of magnitude at 1000 actions.

7.5.3 Results for N-Player Games

In the next set of experiments we compared Algorithm 2 to Govindan-Wilson and Simplicial Subdivision (which was implemented in Gambit, and thus combined with iterated removal of weakly dominated strategies). First, to compare performance on a fixed problem size we tested on 6-player, 5-action games drawn from 22 of GAMUT's n-player distributions.[8] While the numbers of players and actions appear small, note that these games have 15625 outcomes and 93750 payoffs. Once again, Figures 7.6, 7.7, and 7.8 show unconditional median running time, percentage of instances solved, and conditional average running time, respectively. Algorithm 2 has a very low unconditional median running time, for the same reason that Algorithm 1 did for two-player games, and outperforms both other algorithms on all distributions. While this dominance does not extend to the other two metrics, the comparison still favors Algorithm 2.

We again investigated the relationship between ρ and the hardness of games under the "Covariance Game" model. For general n-player games, minimal correlation under this model occurs when $\rho = -\frac{1}{n-1}$. Thus, we can only study the range $[-0.2, 1]$ for 6-player games. Figure 7.9 shows the results for 6-player 5-action games. Algorithm 2, over the range $[-0.1, 0]$, experiences a transition in hardness that is even sharper than that of Algorithm 1. Simplicial Subdivision also undergoes a transition, which is not as sharp, that begins at a much larger value of ρ (around 0.4). However, the running time of Govindan-Wilson is only slightly affected by the covariance, as it neither suffers as much for small values of ρ nor benefits as much from large values.

[8] Two distributions from the tests of 2-player games are missing here, due to the fact that they do not naturally generalize to more than 2 players.

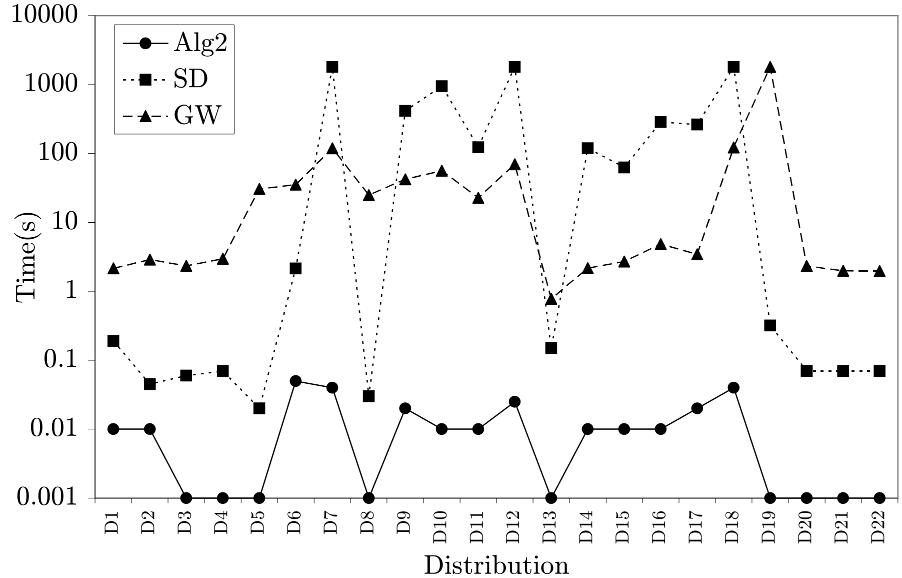


Figure 7.6: Unconditional Median Running Times for Algorithm 2, Simplicial Subdivision, and Govindan-Wilson on 6-player, 5-action Games.

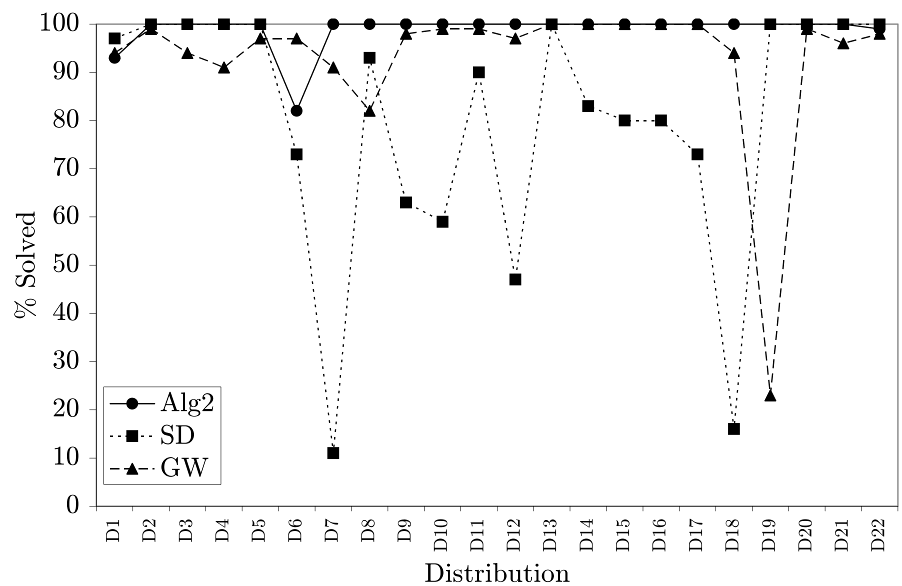


Figure 7.7: Percentage Solved by Algorithm 2, Simplicial Subdivision, and Govindan-Wilson on 6-player, 5-action Games.

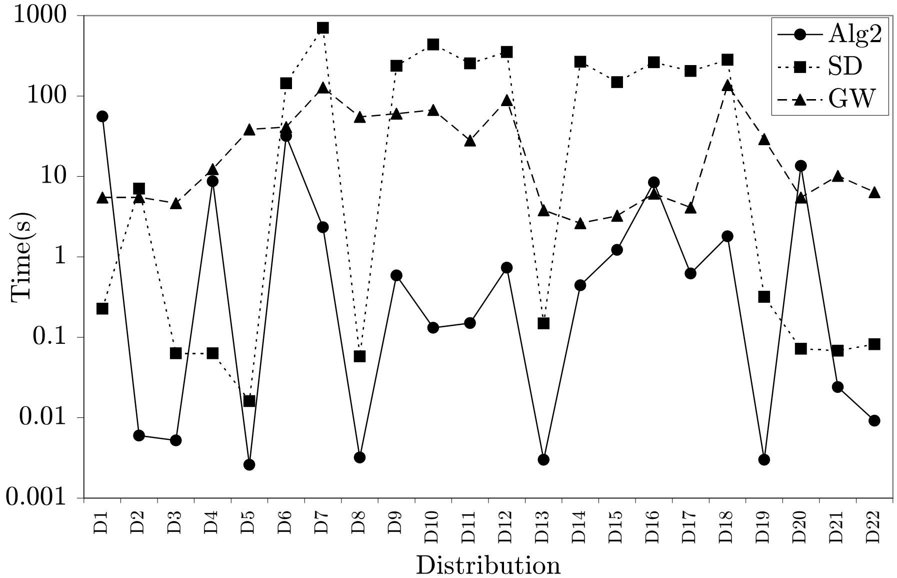


Figure 7.8: Average Running Time on Solved Instances for Algorithm 2, Simplicial Subdivision, and Govindan-Wilson on 6-player, 5-action Games.

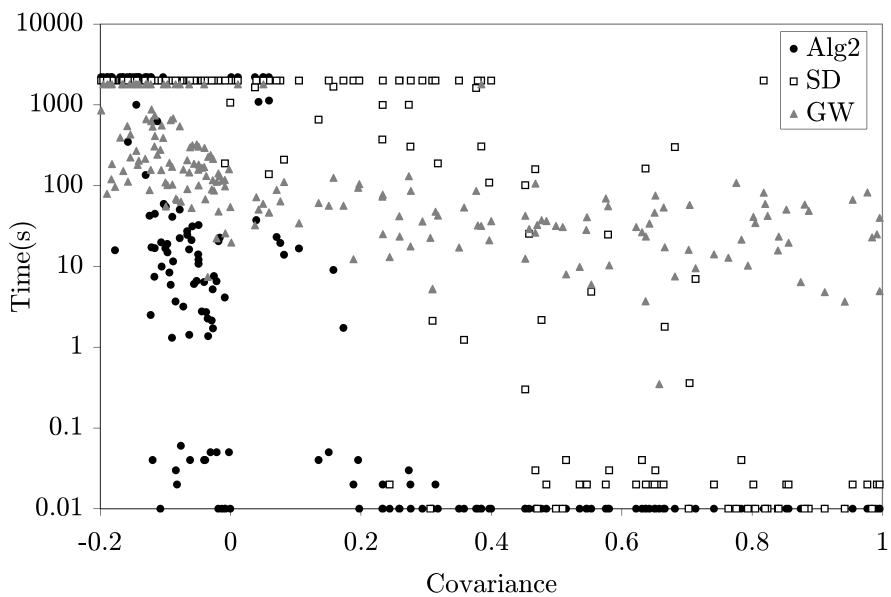


Figure 7.9: Running Time for Algorithm 2, Simplicial Subdivision, and Govindan-Wilson on 6-player, 5-action "Covariance Games".

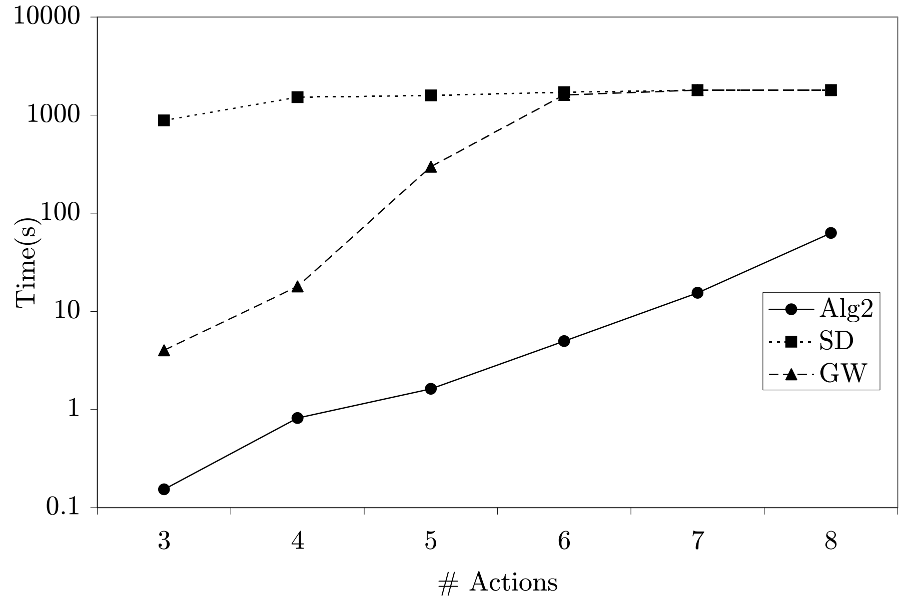


Figure 7.10: Unconditional Average Time for Algorithm 2, Simplicial Subdivision, and Govindan-Wilson on 6-player "Uniformly Random Games" vs. the Number of Actions.

Finally, Figures 7.10 and 7.11 compare the scaling behavior (in terms of unconditional average running times) of the three algorithms: the former holds the number of players constant at 6 and varies the number of actions from 3 to 8, while the latter holds the number of actions constant at 5, and varies the number of players from 3 to 8. In both experiments, both Simplicial Subdivision and Govindan-Wilson solved no instances for the largest two sizes, while Algorithm 2 still found a solution for most games.

7.5.4 On the Distribution of Support Sizes

It is clear from the performance of Algorithm 1 and Algorithm 2 that many games in GAMUT must have small balanced Nash equilibria. We now take a closer look at the nature of these equilibria by examining the kinds of equilibria discovered by different algorithms.

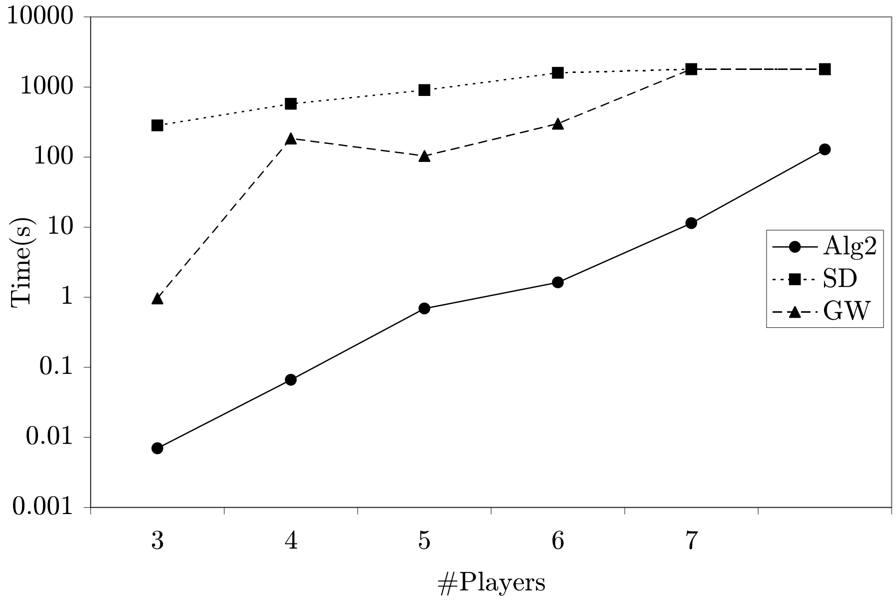


Figure 7.11: Unconditional Average time for Algorithm 2, Simplicial Subdivision, and Govindan-Wilson on 5-action "Uniformly Random Games" vs. the Number of Players.

Pure-Strategy Equilibria

By definition, the first step of both Algorithm 1 and Algorithm 2 is to check whether the input game has a pure-strategy Nash equilibrium (PSNE). This step can be performed very fast even on large games. It is interesting to see just how much of performance of these algorithms is due to the existence of pure-strategy equilibria. Figures 7.12 and 7.13 show the fraction of games in each distribution that posses a PSNE, for 2-player, 300-action and 6-player, 5-action games, respectively. These figures demonstrate that a pure-strategy equilibrium is present in many, though not all, games generated by GAMUT. We note, however, that our algorithms often perform well even on distributions that don't all have a PSNE, as they sometimes find equilibria of larger sizes.

These graphs thus demonstrate that looking for pure strategy Nash equilibria could be a useful preprocessing step for Lemke-Howson, Simplicial Subdivision , and Govindan-Wilson, but that at the same time, even with such a preprocessing step, these algorithms would not catch up with Algorithm 1 and Algorithm 2. This step is essentially the first step of our algorithms, but in the cases that are not caught by this step, our algorithms degrade gracefully while the others do not.

Support Sizes Found

Many researchers feel that equilibria with small support sizes are easier to justify. It is, therefore, interesting to see what kinds of equilibria that different algorithms find in games of interest. Specifically, we would like to see whether other algorithms find equilibria that correspond to our own heuristics of small and balanced supports.

Figures 7.14 and 7.15 show the average total size of the support ($i.e.$, $\sum_i x_i$) of the first equilibrium found by each algorithm on 2-player, 300-action and 6-player, 5-action games, respectively. Total size 2 in Figure 7.14, and total size 6 in Figure 7.15, correspond to a pure-strategy equilibrium. As expected, our algorithms tend to find equilibria with much smaller supports than those of other algorithms, even on distributions where pure-strategy Nash equilibria often exist. Notice, however, that all of the algorithms have a bias towards somewhat small support sizes. On average,

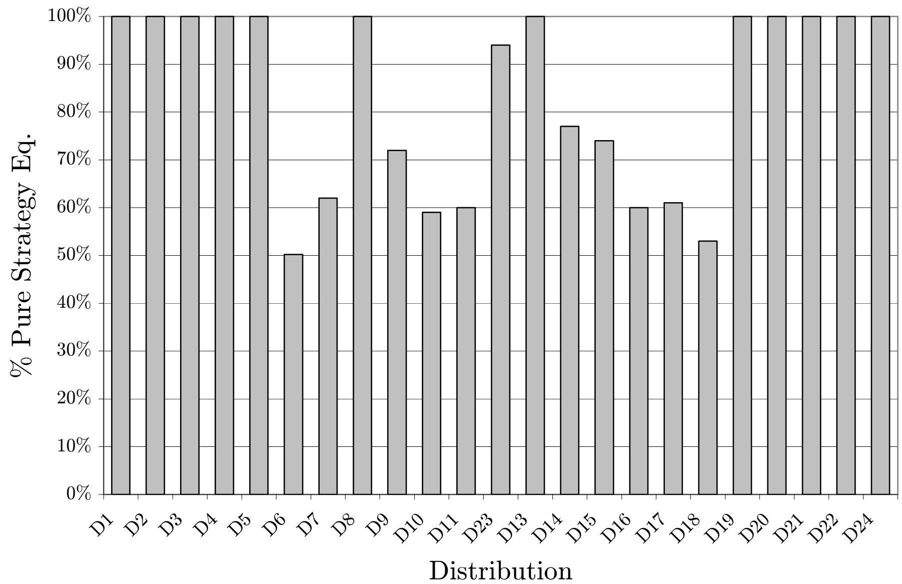


Figure 7.12: Percentage of Instances Possessing a Pure-Strategy NE, for 2-player, 300-action Games.

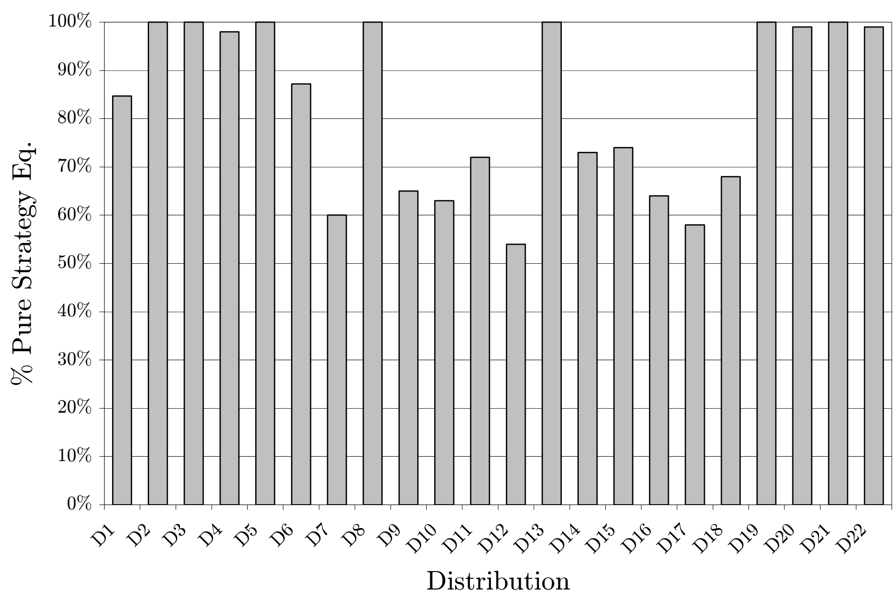


Figure 7.13: Percentage of Instances Possessing a Pure-Strategy NE, for 6-player, 5-action Games.

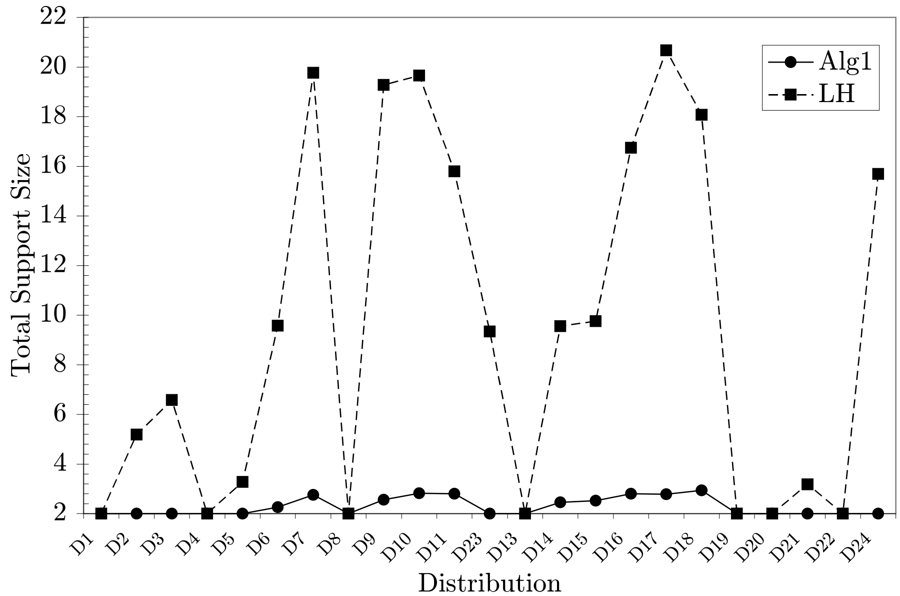


Figure 7.14: Average of Total Support Size for Found Equilibria, on 2-player, 300-action Games.

the equilibria found by Lemke-Howson in 2-player games have at most 10 actions per player (out of 300), while the equilibria found by Govindan-Wilson on 6-player games have on average 2.3 actions per player out of 5. For comparison, the absolute largest equilibrium found by Govindan-Wilson across all instances had between 3 and 5 actions for each player (4.17 on average). The games with these large equilibria all came from Covariance Game distributions. They all also possessed either a PSNE, or a NE with at most 2 actions per player, and were all quickly solved by Algorithm 2. The largest equilibrium among 2-player games that was found by Lemke-Howson had 122 actions for both players. That game came from distribution D23.

The second search bias that our algorithms employ is to look at balanced supports first. Figure 7.16 shows the average measure of support balance (*i.e.*, $max_{i,j}(x_i - x_j)$) of the first equilibrium found by each algorithm on 6-player, 5-action games. We omit a similar graph for 2-player games, since almost always perfectly balanced supports were found by both Algorithm 1 and Lemke-Howson. The only exception was distribution D24, on which Lemke-Howson found equilibria with average (dis)balance

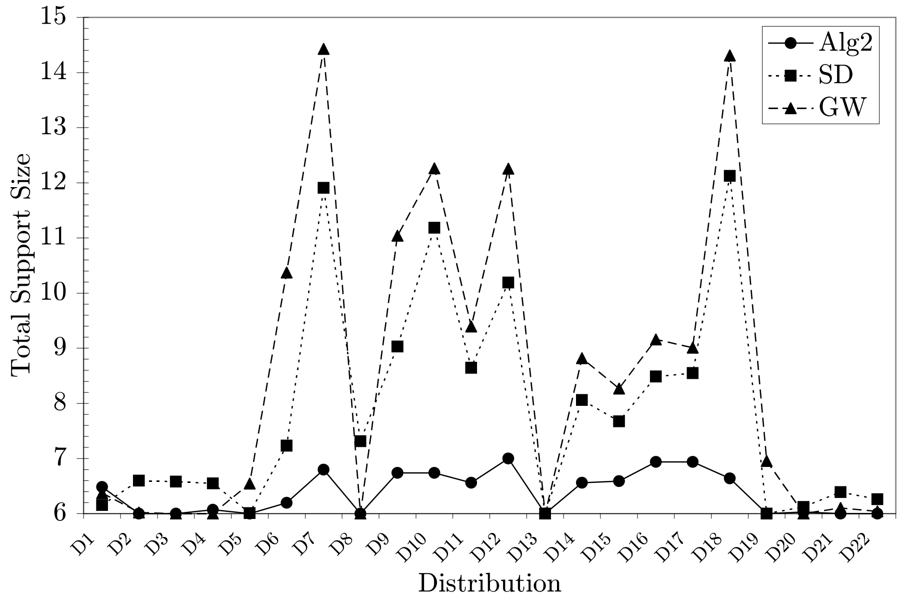


Figure 7.15: Average of Total Support Size for Found Equilibria, on 6-player, 5-action Games.

of 13.48. On 6-player games, as again expected, Algorithm 2 tends to find equilibria that are almost balanced, while the other two algorithms find much less balanced supports. Nevertheless, it is important to note that the balance of the supports found by Algorithm 2 is not uniformly zero, suggesting that it still finds many equilibria that are not pure-strategy. Thus, once again, we see that using the preprocessing step of finding PSNEs first, while greatly improving previous algorithms, would not be enough to close the gap between them on the one hand and Algorithm 2 on the other.

7.6 Conclusion

In this chapter, we presented a pair of algorithms for finding a sample Nash equilibrium. Both employ backtracking approaches (augmented with pruning) to search the space of support profiles, favoring supports that are small and balanced. We showed that they outperform the current state of the art on games drawn from GAMUT.

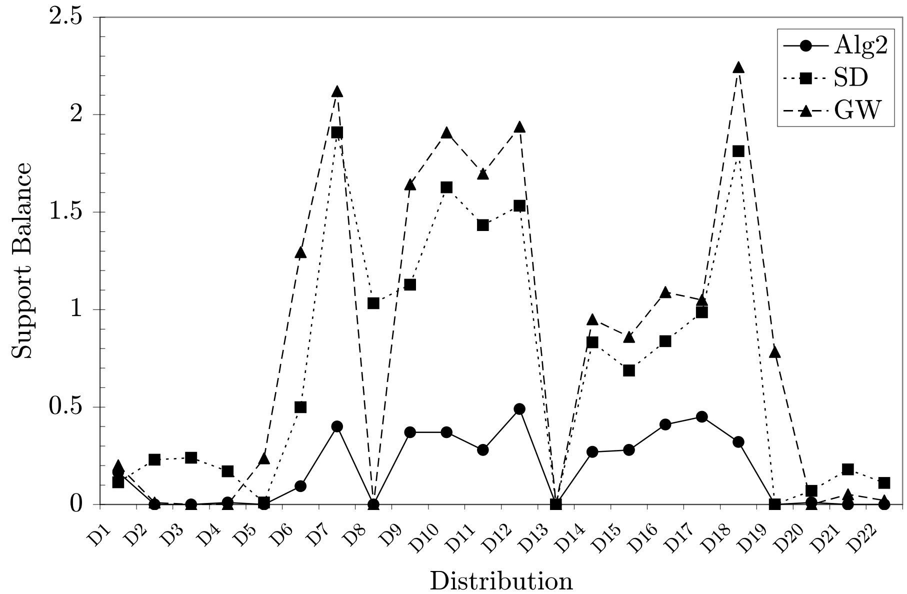


Figure 7.16: Average of Support Size Balance for Found Equilibria, on 6-player, 5-action Games.

Another approach that we have tried, and found to be successful, is to overlay a particular heuristic onto Lemke-Howson. Recall (Chapter 5) that, in the execution of Lemke-Howson, the first pivot is determined by an arbitrary choice of an action of the first player. This initial pivot then determines a path to a NE. Thus, we can construct an algorithm that, like our two algorithms, is biased towards "simple" solutions through the use of breadth-first search — initially, it branches on each possible starting pivot; then, on each iteration, it performs a single pivot for each possible Lemke-Howson execution, after which it checks whether a NE has been found. This modification significantly outperforms the standard Lemke-Howson algorithm on most classes of games. However, it still performs worse than our Algorithm 1.

The most difficult games we encountered came from the "Covariance Game" model, as the covariance approaches its minimal value, and this is a natural target for future algorithm development. We expect these games to be hard in general, because, empirically, we found that as the covariance decreases, the number of equilibria decreases as well, and the equilibria that do exist are more likely to have support sizes near one half of the number of actions, which is the support size with the largest number of supports.

It is certainly possible to improve on the methods presented here. For instance, one might employ more sophisticated CSP techniques. Another promising direction to explore is local search, in which the state space is the set of all possible supports, and the available moves are to add or delete an action from the support of a player. While the fact that no equilibrium exists for a particular support does not give any guidance as to which neighboring support to explore next, one could use a relaxation of Feasibility Program 1 that penalizes infeasibility through an objective function. Our results show that AI techniques can be successfully applied to this problem, and we have only scratched the surface of possibilities along this direction.

However, the most important lesson that can be drawn from this work, is that games in GAMUT, most of which were proposed to model interesting strategic situations, tend to have Nash equilibria with small and balanced supports. This may be very reassuring to game-theory researchers: pure-strategy equilibria, or “simple” equilibria, seem to be more justifiable; philosophically, using mixed strategies often appears to be problematic. We have demonstrated that it pays off for the algorithms to have strong bias towards such solutions, and that researchers of this problem should think more about distributions and structure of Nash equilibria. On a higher level, this work demonstrated how empirical evaluation of simple algorithms can bring forth new understanding of the underlying problem that would have been impossible to obtain theoretically.

Bibliography

Achlioptas, D., C. P. Gomes, H. A. Kautz and B. Selman (2000). Generating satisfiable problem instances. *Proc. AAAI-2000*.

Achlioptas, D., H. Jia and C. Moore (2004). Hiding satisfying assignments: Two are better than one. *Proc. AAAI-2004*.

Anderson, A., M. Tenhunen and F. Ygge (2000). Integer programming for combinatorial auction winner determination. *Proc. ICMAS-2000* (pp. 39–46).

Bacchus, F. and J. Winter (2003). Effective preprocessing with hyper-resolution and equality reduction. *Proc. SAT-2003* (pp. 341–355).

Blum, B., C. R. Shelton and D. Koller (2003). A continuation method for Nash equilibria in structured games. *Proc. IJCAI-2003*.

Bowling, M. and M. Veloso (2001). Rational and convergent learning in stochastic games. *Proc. IJCAI-2003*.

Chaloner, K. and I. Verdinelli (1995). Bayesian experimental design: A review. *Statistical Science*, *10*, 273–304.

Cheeseman, P., B. Kanefsky and W. M. Taylor (1991). Where the Really Hard Problems Are. *Proc. IJCAI-1991*.

Coarfa, C., D. Demopoulos, A. San Miguel Aguirre, D. Subramanian and M. Vardi (2000). Random 3-SAT: The plot thickens. *Proc. CP-2000*.

Conitzer, V. and T. Sandholm (2003). Complexity results about Nash equilibria. *Proc. IJCAI-2003*.

Cook, S. (1971). The complexity of theorem proving procedures. *Proc. STOC-1971*.

Cook, S. A. and D. G Mitchell (1997). Finding hard instances of the satisfiability problem: A survey. In Du, Gu and Pardalos (Eds.), *Satisfiability problem: Theory and applications*, vol. 35, 1–17. American Mathematical Society.

Cramton, P., Y. Shoham and R. Steinberg (editors) (2006). *Combinatorial auctions*. MIT Press.

Davis, M., G. Logemann and D. Loveland (1962). A machine program for theorem-proving. *Communications of the ACM*, *5*(7), 394 – 397.

Davis, M. and H. Putnam (1960). A computing procedure for quantification theory. *Journal of the ACM*, *7*, 201 – 215.

de Farias, D. Pucci and N. Megiddo (2004). How to combine expert (or novice) advice when actions impact the environment. *Proc. NIPS-2004*.

de Vries, S. and R. Vohra (2003). Combinatorial auctions: A survey. *INFORMS Journal on Computing*, *15*(3).

Dechter, R. (2003). *Constraint processing*. Morgan Kaufmann.

Dickhaut, J. and T. Kaplan (1991). A program for finding Nash equilibria. *The Mathematica Journal*, 87–93.

Doucet, A., N. de Freitas and N. Gordon (ed.) (2001). *Sequential Monte Carlo Methods in Practice*. Springer-Verlag.

Efron, B., T. Hastie, I. Johnstone and R. Tibshirani (2002). Regression shrinkage and selection via the lasso.

Friedman, J. (1991). Multivariate adaptive regression splines. *Annals of Statistics*, *19*.

Fujishima, Y., K. Leyton-Brown and Y. Shoham (1999). Taming the computational complexity of combinatorial auctions: Optimal and approximate approaches. *Proc. IJCAI-1999*.

Garey, M.R. and D.S. Johnson (1979). *Computers and intractability, a guide to the theorey of NP-completeness*. W.H. Freeman and Co.

Gilboa, I. and E. Zemel (1989). Nash and correlated equilibria: Some complexity considerations. *Games and Economic Behavior*, *1*.

Gomes, C., C. Fernández, B. Selman and C. Bessière (2004). Statistical regimes across constrainedness regions. *Proc. CP-2004*.

Gomes, C. and B. Selman (2001). Algorithm portfolios. *Artificial Intelligence*, *126*(1-2), 43–62.

Gomes, C., B. Selman, N. Crato and H. Kautz (2000). Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, *24*(1).

Gomes, C. P. and B. Selman (1997). Problem structure in the presence of perturbations. *Proc. AAAI/IAAI-1997*.

Gonen, R. and D. Lehmann (2000). Optimal solutions for multi-unit combinatorial auctions: Branch and bound heuristics. *Proc. EC-2000*.

Gonen, R. and D. Lehmann (2001). *Linear programming helps solving large multi-unit combinatorial auctions* (Technical Report TR-2001-8). Leibniz Center for Research in Computer Science.

Govindan, S. and R. Wilson (2003). A global Newton method to compute Nash equilibria. *Journal of Economic Theory*.

Grenager, T., R. Powers and Y. Shoham (2002). Dispersion games. *Proc. AAAI-2002*.

Hastie, T., R. Tibshirani and J. Friedman (2001). *Elements of statistical learning*. Springer.

Hoos, H.H. and C. Boutilier (2000). Solving combinatorial auctions using stochastic local search. *Proc. AAAI-2000* (pp. 22–29).

Hoos, H.H. and T. Stützle (1999). Towards a characterisation of the behaviour of stochastic local search algorithms for SAT. *Artificial Intelligence*, *112*, 213–232.

Hoos, H. H. and T. Stützle (2004). *Stochastic local search—foundations and applications*. Morgan Kaufmann.

Horvitz, E., Y. Ruan, C. Gomes, H. Kautz, B. Selman and M. Chickering (2001). A Bayesian approach to tackling hard computational problems. *Proc. UAI-2001*.

Ieong, S., R. McGrew, E. Nudelman, Y. Shoham and Q. Sun (2005). Fast and compact: A simple class of congestion games. *Proc. AAAI-2005*.

ILOG (2004). CPLEX. http://www.ilog.com/products/cplex.

Jia, H., C. Moore and B. Selman (2004). From a glass spin model to a 3-SAT hard formula. *Proc. SAT-2004*.

Jia, H., C. Moore and D. Strain (2005). Generating hard satisfiable formulas by hiding solutions deceptively. *Proc. AAAI-2005*.

Karmarkar, N. (1984). A new polynomial-time algorithm for linear programming. *Combinatorica*, *4*, 373–395.

Khachiyan, L. (1979). A polynomial time algorithm for linear programming. *Doklady Akademii Nauk SSSR*, *244*, 1093–1096.

Kohavi, R. and G. John (1997). Wrappers for feature subset selection. *Artificial Intelligence Journal, Special Issue on Relevance*, *97*(1–2), 273–324.

Kohlberg, E. and J.F. Mertens (1986). On the strategic stability of equilibria. *Econometrica*, *54*.

Kolaitis, P. (2003). Constraint satisfaction, databases and logic. *Proc. IJCAI-2003* (pp. 1587–1595).

Koller, D. and B. Milch (2001). Multi-agent influence diagrams for representing and solving games. *Proc. IJCAI-2001*.

Korf, R. and M. Reid (1998). Complexity analysis of admissible heuristic search. *Proc. AAAI-98*.

Lagoudakis, M. and M. Littman (2000). Algorithm selection using reinforcement learning. *Proc. ICML-2000*.

Lagoudakis, M. and M. Littman (2001). Learning to select branching rules in the DPLL procedure for satisfiability. *Proc. LICS/SAT-2001*.

Le Berre, D. and L. Simon (2003). The essentials of the SAT 2003 competition. *Proc. SAT-2003* (pp. 452–467).

Lemke, C. (1965). Bimatrix equilibrium points and mathematical programming. *Management Science*, *11*, 681–689.

Lemke, C. and J. Howson (1964). Equilibrium points of bimatrix games. *Journal of the Society for Industrial and Applied Mathematics*, *12*, 413–423.

Levin, L. (1973). Universal search problems. *Problemy Peredachi Informatsii (Problems in Information Transmission)*, *9*(3), 265 – 266. In Russian.

Leyton-Brown, K., E. Nudelman, G. Andrew, J. McFadden and Y. Shoham (2003a). Boosting as a metaphor for algorithm design. *Proc. CP-2003*.

Leyton-Brown, K., E. Nudelman, G. Andrew, J. McFadden and Y. Shoham (2003b). A portfolio approach to algorithm selection. *Proc. IJCAI-2003*.

Leyton-Brown, K., E. Nudelman and Y. Shoham (2002). Learning the empirical hardness of optimization problems: The case of combinatorial auctions. *Proc. CP-2002*.

Leyton-Brown, K., M. Pearson and Y. Shoham (2000a). Towards a universal test suite for combinatorial auction algorithms. *Proc. EC-2000*.

Leyton-Brown, K., Y. Shoham and M. Tennenholtz (2000b). An algorithm for multi-unit combinatorial auctions. *Proc. of AAAI-2000*.

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. *Proc. ICML-1994*.

Lobjois, L. and M. Lemaître (1998). Branch and bound algorithm selection by performance prediction. *Proc. AAAI-1998*.

Mangasarian, O. (1964). Equilibrium points of bimatrix games. *Journal of the Society for Industrial and Applied Mathematics*, *12*, 780–780.

Mason, R. L., R. F. Gunst and J. L. Hess (2003). *Statistical design and analysis of experiments*. Wiley-Interscience.

McKelvey, R. and A. McLennan (1996). Computation of equilibria in finite games. In H. Amman and an D. Kendrick J. Rust (Eds.), *Handbook of computational economics*, vol. I. Elsevier.

McKelvey, R. and A. McLennan (1997). The maximal number of regular totally mixed Nash equilibria. *Journal of Economic Theory*, *72*, 411–425.

McKelvey, R. D., A. McLennan and T. Turocy (1992). Gambit: game theory analysis software and tools. http://econweb.tamu.edu/gambit.

McLennan, A. and J. Berg (2002). The asymptotic expected number of Nash equilibria of two player normal form games. Mimeo, University of Minnesota.

Megiddo, N. and C. Papadimitriou (1991). A note on total functions, existence theorems and complexity. *Theoretical Computer Science*, *81*, 317–324.

Milgrom, P. and J. Roberts (1990). Rationalizability, learning and equilibrium in games with strategic complementarities. *Econometrica*, *58*.

Monasson, R., R. Zecchina, S. Kirkpatrick, B. Selman and L. Troyansky (1998). Determining computational complexity for characteristic 'phase transitions'. *Nature*, *400*.

Murtagh, B. and M. Saunders (2004). MINOS. http://www.sbsi-sol-optimize.com.

Nash, J.F. (1950). Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences of the United States of America*, *36*, 48–49.

Nisan, N. (2000). Bidding and allocation in combinatorial auctions. *Proc. EC-2000*.

Nudelman, E., K. Leyton-Brown, A. Devkar, Y. Shoham and H. Hoos (2004a). Understanding random SAT: Beyond the clauses-to-variables ratio. *Proc. CP-2004*.

Nudelman, E., J. Wortman, Y. Shoham and K. Leyton-Brown (2004b). Run the GAMUT: A comprehensive approach to evaluating game-theoretic algorithms. *Proc. AAMAS-2004*.

Osborne, M.J. and A. Rubinstein (1994). *A course in game theory*. MIT Press.

Papadimitriou, C. (2001). Algorithms, games, and the internet. *Proc. STOC-2001*.

Porter, R., E. Nudelman and Y. Shoham (2004). Simple search methods for finding a Nash equilibrium. *Proc. AAAI-2004*.

Porter, R., E. Nudelman and Y. Shoham (to appear). Simple search methods for finding a Nash equilibrium. *Games and Economic Behavior*.

Powers, R. and Y. Shoham (2004). New criteria and a new algorithm for learning in multi-agent systems. *Proc. NIPS-2004*.

Powers, R. and Y. Shoham (2005). Learning against opponents with bounded memory. *Proc. IJCAI-2005*.

Rapoport, A., M. Guyer and D. Gordon (1976). *The 2x2 game*. Univeristy of Michigan Press.

Rice, J. R. (1976). The algorithm selection problem. *Advances in Computers*, *15*, 65–118.

Rinott, Y. and M. Scarsini (2000). On the number of pure strategy Nash equilibria in random games. *Games and Economic Behavior*, *33*.

Rosenthal, R.W. (1973). A class of games possessing pure-strategy Nash equilibria. *International Journal of Game Theory*, *2*, 65–67.

Rothkopf, M.H., A. Pekeč and R.M. Harstad (1998). Computationally manageable combinatorial auctions. *Management Science*, *44*(8), 1131–1147.

Roughgarden, T. (2005). *Selfish routing and the price of anarchy*. Cambridge, MA: MIT Press.

Ruan, Y., E. Horvitz and H. Kautz (2002). Restart policies with dependence among runs: A dynamic programming approach. *Proc. CP-2002*.

Ruckle, W. H. (1983). *Geometric games and their applications*. Pitman.

Russel, S. J. and P. Norvig (2003). *Artificial intelligence: A modern approach*. Prentice Hall.

Sandholm, T. (1999). An algorithm for optimal winner determination in combinatorial auctions. *Proc. IJCAI-1999*.

Sandholm, T. (2002). Algorithm for optimal winner determination in combinatorial auctions. *Artificial Intelligence*, *135*(1), 1–54.

Sandholm, T., A. Gilpin and V. Conitzer (2005). Mixed-integer programming methods for finding Nash equilibria. *Proc. AAAI-2005*.

Sandholm, T., S. Suri, A. Gilpin and D. Levine (2001). CABOB: A fast optimal algorithm for combinatorial auctions. *Proc. IJCAI-2001*.

Savage, L. J. (1954). *The foundations of statistics*. New York: Wiley.

Savani, R. and B. von Stengel (2004). Exponentially many steps for finding a Nash equilibrium in a bimatrix game. *Proc. FOCS-2004* (pp. 258–267).

Schapire, R. (1990). The strength of weak learnability. *Machine Learning*, *5*, 197–227.

Selman, B., D. G. Mitchell and H. J. Levesque (1996). Generating hard satisfiability problems. *Artificial Intelligence*, *81*(1-2), 17–29.

Shoham, Y., R. Powers and T. Grenager (2004). On the agenda(s) of research on multi-agent learning. *AAAI Fall Symposium on Artificial Multi-Agent Learning*.

Slaney, J. and T. Walsh (2001). Backbones in optimization and approximation. *Proc. IJCAI-2001*.

Tompkins, D. and H. Hoos (2004). UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. *Proc. SAT-2004* (pp. 37–46).

van der Laan, G., A. Talman and L. van der Heyden (1987). Simplicial variable dimension algorithms for solving the nonlinear complementarity problem on a product of unit simplices using a general labelling. *Mathematics of Operations Research*.

Vickrey, D. and D. Koller (2002). Multi-agent algorithms for solving graphical games. *Proc. AAAI-2002*.

von Stengel, B. (2002). Computing equilibria for two-person games. In R. Aumann and S. Hart (Eds.), *Handbook of game theory*, vol. 3, chapter 45. North-Holland.

Watkins, C. J. C. H. and P. Dayan (1992). Technical note: Q-learning. *Machine Learning*, *8(3/4)*.

Williams, R., C. Gomes and B. Selman (2003). Backdoors to typical case complexity. *Proc. IJCAI-2003* (pp. 1173–1178).

Wood, M. and G. Dantzig (1949). Programming of interdependent activities. i. general discussion. *Econometrica*, *17*, 193–199.

Zhang, W. (1999). *State-space search: Algorithms, complexity, extensions, and applications*. Springer.