Run the GAMUT:
A Comprehensive Approach to Evaluating Game-Theoretic Algorithms

Eugene Nudelman Jennifer Wortman Yoav Shoham
{eugnud;jwortman;shoham}@cs.stanford.edu Stanford University, Stanford, CA 94305

Kevin Leyton-Brown
kevinlb@cs.ubc.ca University of British Columbia, Vancouver, BC V6T 1Z4

Abstract

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.[2]

1. Introduction

Researchers in multiagent systems have become increasingly interested in game theory as a modeling tool. This has led to growing interest in computational problems associated with game-theoretic domains. Two such problems are computing Nash equilibria and learning to achieve good payoffs in repeated games. It is often difficult to offer theoretical guarantees about such algorithms' performance: the computational complexity of many algorithms for computing Nash remains an interesting open problem [14], and there is rarely anything that can be proven about the sort of performance a learning algorithm will achieve without making reference to the game it will play or the opponents it will face. For these sorts of reasons, researchers needing to evaluate algorithms for game-theoretic problems often choose to perform empirical tests.

One general lesson that has been learned 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 [9]. When we examine 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. To our surprise, we discovered two things.

First, for *every one* 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

1 Available at `http://gamut.stanford.edu`
2 This work was supported by NSF grant IIS-0205633 and DARPA grant F30602-00-2-0598.

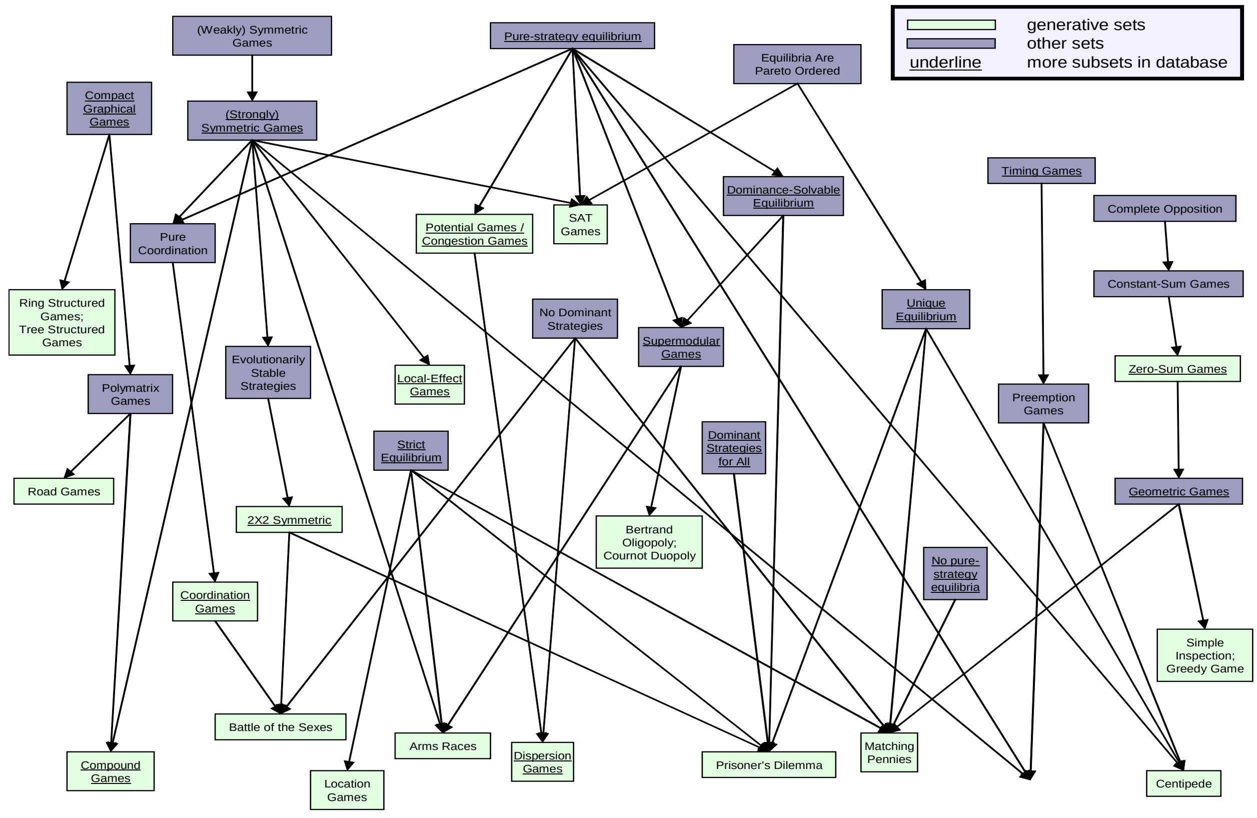

Figure 1. GAMUT Taxonomy (Partial)

different sets of games.

Second, we were surprised to find very little work that aimed to understand, taxonomize or even enumerate non-generic games in a holistic or integrative way. We came across work on understanding *generic* games [7], and found a complete taxonomy of two-player two-action games [15]. 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.

In this paper we aim to fill this gap: to identify interesting sets of non-generic games comprehensively and with as little bias as possible. 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 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 the appendix, we briefly describe GAMUT's architecture and implementation, including discussion of how new games may easily be added.

2. GAMUT

For the initial 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 the appendix). 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).

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 1. Game Generators in GAMUT

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 1. To illustrate the sort of information that can be conveyed by this figure, we can see that all Dispersion Games [6] are Congestion Games [17] 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 [13]; 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 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 than we can fit into a conference-length paper, this database also allows users to navigate according to subset/superset relationships and to perform searches.

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 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 generators 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, we can generate any game

$$\begin{pmatrix} R,R & S,T \\ T,S & P,P \end{pmatrix}$$

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 [5]. 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 [18] 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 [15] 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 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 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 the Appendix.

3. Running the GAMUT

At the beginning of this paper we claimed 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).

3.1. Computation of Nash Equilibria

One of the most interesting computational problems in game theory is computing Nash equilibria. All evidence suggests that this is a hard problem (e.g., [4, 3]), yet the precise complexity class into which the problem falls is unknown [14]. In this section we use GAMUT to evaluate three algorithms' empirical properties on this problem.

3.1.1. Experimental Setup The best-known game theory software package is Gambit [12], a collection of state-of-the-art algorithms. For two-player games the Lemke-Howson algorithm [8] is best and is used by default in Gambit. For n-player games Gambit uses an algorithm based on Simplicial Subdivision [19]. In both cases, Gambit performs iterative removal of dominated strategies as a preprocessing step. Govindan and Wilson [5] introduced an alternative algorithm based on a continuation method. We use a recent optimized implementation, the GameTracer package [1]. This work also included 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 [1], 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 paper 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.

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.

Suffixes “-CG”, “-RG”, “-SG”, “-SW” and “-Road” indicate, respectively, complete, random, star-shaped, small-world, and road-shaped (see [20]) graphs. Another distribution that we decided to split was the Covariant Game distribution, which implements the random game model of [16]. 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 $[\frac{-1}{n-1}, 1]$.

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 [11, 21] were obtained for generic games with payoffs drawn independently uniformly at random (in GAMUT, this would be the `RandomGame` generator). Our work may therefore represent the first systematic attempt to understand the empirical behavior of these algorithms on non-generic games.

3.1.2. Experimental Results Figure 2 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, 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. Moreover, for all three algorithms, we observe that this variation is not an artifact of one particular problem size.

Figure 3 illustrates runtime differences both across and among distributions for 6-player 5-action games. (Though we do not have space to show them here, we observed qualitatively similar results for different input sizes and for the Lemke-Howson algorithm.) 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

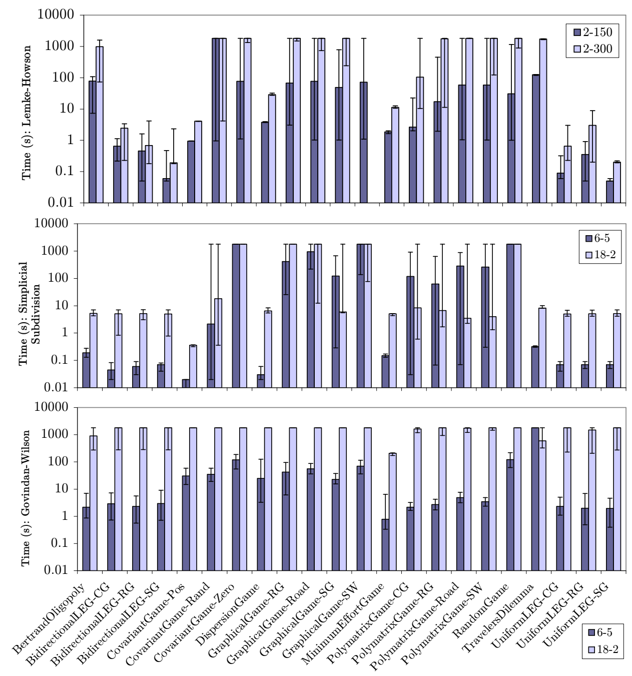

Figure 2. Effect of Problem Size on Solver Performance

be inferred from 25th/50th/75th quartile figures such as Figure 2. 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.

Figure 3 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 4. 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. 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.

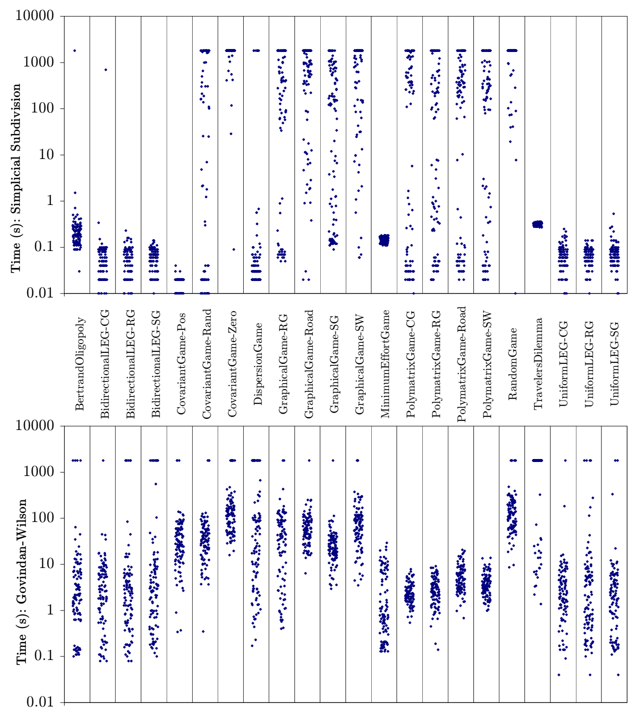

Figure 3. Runtime Distribution for 6-player, 5-action Games

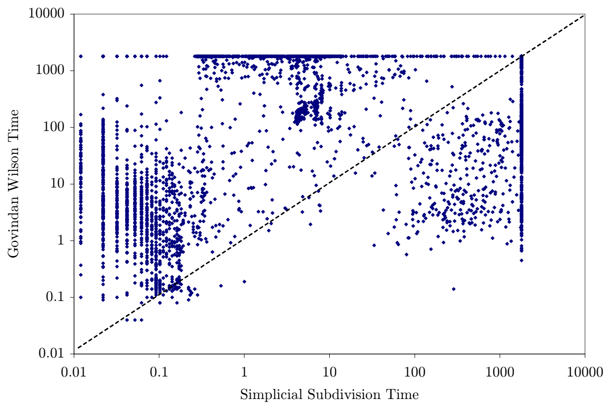

Figure 4. Correlation, 6-player, 5-action.

3.2. Multiagent Learning in Repeated Games

The last few years have seen a surge of research into multiagent learning, resulting in the recent proposal of several new algorithms. 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 believe it is clear, however, that our results show that these algorithms' performance depends crucially on the distributions of games on which they are run, and thus that GAMUT will be a useful tool for researchers in the multiagent learning community.

3.2.1. Experimental Setup We used three learning algorithms: Minimax-Q [10], WoLF [2], and a version of the original Q-learning algorithm for single agent games [22] 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 assumes away the multiagent component, and thus is not guaranteed to converge at all against an adaptive opponent. Minimax-Q plays a safety-level strategy, and so does not necessarily converge to a best response. WoLF is a variable-learning-rate policy-hill-climbing algorithm that *is* designed to converge to a best response. 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 [10, 2, 22] and then fixed these parameters for all experiments.

In our experiments we chose to focus on a set of 13 distributions. As before, we keep 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. 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 the outcomes of coin flips. On each run, we repeated the game 100,000 times. The first 90,000 rounds allow the algorithms to settle into their long-run behavior; we then compute 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.

3.2.2. 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.

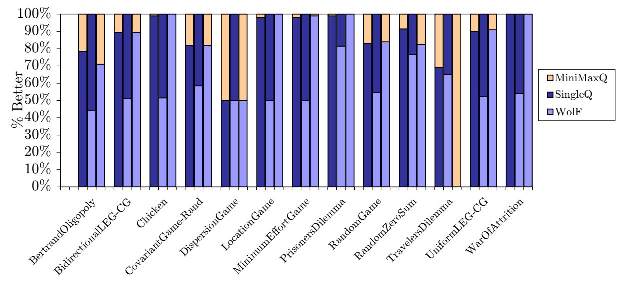

Figure 5. Pairwise Comparison of Algorithms

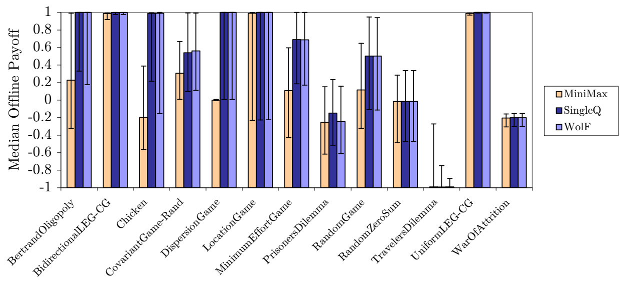

Figure 6. Median Payoffs as Player 1

Figure 5 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 across games. The overall conclusion that we can draw from this graph 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 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. 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, 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 5.

4. Conclusion

In this paper we presented GAMUT, a game theory test suite. We surveyed hundreds of books and papers to compile a comprehensive database of structured non-generic games and the relationships between them. We built a highly modular and extensible software framework, and used it to implement generators for these sets of games. Finally, we 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. We hope that GAMUT will become a useful tool for researchers working at the intersection of game theory and computer science.

Acknowledgments

We would like to thank: Bob Wilson for identifying many useful references and for sharing his insights into the space of games; Rob Powers for providing us with implementations of multiagent learning algorithms; members of Stanford's Multiagent group for many useful comments.

Appendix: GAMUT Implementation

The GAMUT software was built using 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 the 35 procedures we identified in 2.2. They are listed in Table 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 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 2, have been implemented to meet these needs in a

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 2. GAMUT Support Classes

modular way. As with games, additional classes of functions and graphs can be easily added.

`Outputter` classes encapsulate the notion of 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 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 2.2, games' strategic properties are preserved under positive affine transformations. Normalization allows payoff magnitudes to be compared and can avoid machine precision problems.

References

[1] B. Blum, C. Shelton, and D. Koller. A continuation method for Nash equilibria in structured games. In *IJCAI*, 2003.

[2] M. Bowling and M. Veloso. Rational and convergent learning in stochastic games. In *IJCAI*, 2001.

[3] V. Conitzer and T. Sandholm. Complexity results about Nash equilibria. In *IJCAI*, 2003.

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

[5] S. Govindan and R. Wilson. A global Newton method to compute Nash equilibria. In *Journal of Economic Theory*, 2003.

[6] T. Grenager, R. Powers, and Y. Shoham. Dispersion games. In *AAAI*, 2002.

[7] E. Kohlberg and J.F. Mertens. On the strategic stability of equilibria. *Econometrica*, 54, 1986.

[8] C. Lemke and J. Howson. Equilibrium points of bimatrix games. *Journal of the Society for Industrial and Applied Mathematics*, 12, 1964.

[9] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *CP*, 2002.

[10] M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *ICML*, 1994.

[11] R. McKelvey and A. McLennan. Computation of equilibria in finite games. In J. Rust H. Amman, D. Kendrick, editor, *Handbook of Computational Economics*, volume I. Elsevier, 1996.

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

[13] P. Milgrom and J. Roberts. Rationalizability, learning and equilibrium in games with strategic complementarities. *Econometrica*, 58, 1990.

[14] C. Papadimitriou. Algorithms, games, and the internet. In *STOC*, 2001.

[15] A. Rapoport, M. Guyer, and D. Gordon. *The 2x2 Game*. Univeristy of Michigan Press, 1976.

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

[17] R.W. Rosenthal. A class of games possessing pure-strategy Nash equilibria. *International Journal of Game Theory*, 2, 1973.

[18] W. H. Ruckle. *Geometric Games and Their Applications*. Pitman, 1983.

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

[20] D. Vickrey and D. Koller. Multi-agent algorithms for solving graphical games. In *AAAI*, 2002.

[21] B. von Stengel. Computing equilibria for two-person games. In R. Aumann and S. Hart, editors, *Handbook of Game Theory*, volume 3, chapter 45. North-Holland, 2002.

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