Empirically Evaluating Multiagent Learning Algorithms

Erik Zawadzki, Asher Lipson and Kevin Leyton-Brown
Department of Computer Science, University of British Columbia, Vancouver, Canada
{*epz, alipson, kevinlb*}*@cs.ubc.ca*

Abstract. There exist many algorithms for learning how to play repeated bimatrix games. Most of these algorithms are justified in terms of some sort of theoretical guarantee. On the other hand, little is known about the empirical performance of these algorithms. Most such claims in the literature are been based on small experiments, which has hampered understanding as well as the development of new multiagent learning (MAL) algorithms. We have developed a new suite of tools for running multiagent experiments: the MultiAgent Learning Testbed (MALT). These tools are designed to facilitate larger and more comprehensive experiments by removing the need to build one-off experimental code. MALT also provides baseline implementations of many MAL algorithms, hopefully eliminating or reducing differences between algorithm implementations and increasing the reproducibility of results. Using this test suite, we ran an experiment unprecedented in size. We analyzed the results according to a variety of performance metrics including reward, maxmin distance, regret, and several notions of equilibrium convergence. We confirmed several pieces of conventional wisdom, but also discovered some surprising results. For example, we found that single-agent Q-learning outperformed many more complicated and more modern MAL algorithms.

Keywords: Game theory, multiagent systems, reinforcement learning, empirical algorithmics

1. Introduction

Urban road networks, hospital systems and commodity markets are all examples of complicated multiagent systems that are essential to everyday life. Indeed, any social interaction can be seen as a multiagent problem. As a result of the prominence of multiagent systems, a lot of attention has been paid to designing and analyzing learning algorithms for multiagent environments. A multitude of different algorithms exist for a variety of different settings. Some prominent examples include algorithms by Littman (1994), Singh et al. (2000), Hu and Wellman (2003), Greenwald and Hall (2003), Bowling (2004a), Powers and Shoham (2005), Banerjee and Peng (2006), and Conitzer and Sandholm (2007).

We take the position that the best multiagent learning (MAL) algorithm is the one that achieves the highest possible average reward.[1] Under this view, the problem faced by the designer of a MAL algorithm is qualitatively the same as the problem faced by the designer of a single-agent reinforcement learning algorithm. However, there is a fundamental difference between the two settings. In the stationary environment faced by classical reinforcement learners, the concept of an optimal policy is well defined, and hence learning algorithms can attempt to identify this policy. In a multiagent environment, the best policy to follow depends on the actions taken by the opponent, and thus on the ways in which the opponent's future behavior will be affected by the learner's present actions. The best policy depends on the opponent's strategy, and so there can be no global "optimum."

[1] For alternatives, see Shoham et al. (2007)—who called the approach that we espouse the "prescriptive, non-cooperative agenda"—or Sandholm (2007).

It is this added conceptual complexity that makes MAL problems interesting; however, it has also made them harder to analyze. Theoretical claims about MAL algorithms generally do not speak directly about average reward. Instead, they tend to describe alternative aspects of the algorithm's performance that are intended to 'stand in' for reward. Some work has insisted that algorithms should converge to stage-game Nash equilibria, or should do so at least in the case of "self play." Others have insisted on other sorts of convergence properties or on regret bounds. Still others have offered different guarantees for performance against different classes of opponents.

Because many MAL algorithms are incomparable on the basis of their theoretical properties, and further because it is unclear the extent to which these various properties correlate with an algorithm's ability to achieve high average reward in practise, it is generally argued that MAL algorithms should be compared empirically. Many such experimental comparisons have been performed in the literature (see, e.g., (Nudelman et al., 2004; Powers and Shoham, 2005)). However, for the most part these experiments have been designed to advocate for a newly-designed algorithm rather than to survey the whole landscape. As a consequence, most of these experiments have been small in terms of the number of game instances and opposing algorithms considered. Furthermore, different experiments have in many cases measured performance in different ways, making it difficult to compare their results and draw an overall conclusion. There is therefore considerable opportunity to expand our understanding of how existing MAL techniques compare in practice.

Part of the reason for the relative paucity of large-scale empirical work is that neither a centralized algorithm repository nor a standardized test setup exists. This is unfortunate, not only because considerable work has to be invested in designing one-off testbeds and reimplementing algorithms, but also because centralized and public repositories increase reproducibility and decrease the danger that different experiments will achieve different results because of differences in implementations. Publicly available and scrutinized implementations offer the promise of experiments that are easier to run, reproduce, and compare.

In this article we make two main contributions. First, we describe the design and implementation of a platform for running MAL experiments (§3). This platform offers several advantages over one-off setups. We hope that it will facilitate new and larger-scale empirical work.

Our second main contribution is the analysis of such an empirical study. This experiment is, to our knowledge, unprecedented in terms of scale. We make suggestions about how empirical MAL performance data should be analyzed (§4), and offer a detailed discussion of different algorithms' average reward in practice (§5). Furthermore, we draw connections between different performance metrics that have been explored in theoretical work (§6), and show that some of the least sophisticated algorithms achieve extremely competitive performance.

2. Algorithms and Past Experimental Work

MAL algorithms have been studied for over half a century. This rich investigation has produced not only a profusion of competing algorithms but also various distinct problem formulations. Does an algorithm know the game's reward functions before the game starts, or do reward functions need to be learned? How many opponents can an algorithm face? What signals about the opponent's actions can an algorithm observe? Can an algorithm rely on being able to determine stage-game

Nash equilibria or other computationally-expensive game properties? Each of these assumptions changes the learning problem.

In this section we describe the algorithms we study in this paper, and also survey past experimental evaluations of MAL algorithms. The creators of the algorithms that we describe answered the above questions in different ways, reflecting the community's broader disagreement about precisely what problem MAL algorithms should aim to solve. In order to permit the study of a broad range of algorithms, we have answered the above questions permissively: we allow algorithms access to the reward functions, to signals about the opponent's actions, and to computationally-costly game properties. Thus, we are able to compare algorithms that require this information to others that are capable of learning it. (Where possible, we have implemented such learning-capable algorithms in such a way that they make use of *a priori* available information directly instead of learning it, to ensure that these algorithms are not disadvantaged.)

The other important experimental choice we faced was the class of games upon which to evaluate algorithms. We chose to restrict ourselves to 2-player repeated games. (Note, however, that we do not restrict the number of actions in the repeated game.) We chose this setting instead of n-player repeated games or either 2- or n-player stochastic games for two reasons. First, the case of two-player repeated games has received the most past study (though see e.g., (Vu et al., 2005)). Second, considerably more work has been done to identify experimentally-interesting test data for this case. We restricted our attention to algorithms that can play two-player games of any size and with any payoff structure. We thus did not make use of work that insists (e.g.) on two-action games (Singh et al., 2000) or constant-sum games (Littman, 1994). We also mention as an aside that MAL experiments have been conducted in settings that are neither generalizations nor restrictions of our setting, such as the population-based work by Axelrod (1987) and Airiau et al. (2007).

2.1. Fictitious Play

`Fictitious play` (Brown, 1951) is probably the earliest example of a learning algorithm for two-player games repeated games. Essentially, `fictitious play` assumes that the opponent is playing an unknown and potentially mixed stationary strategy, and tries to estimate this strategy from the opponent's empirical distribution of actions—the frequency counts for each of its actions normalized to be probabilities. Clearly, in order to collect the frequency counts `fictitious play` must be able to observe the opponent's actions. The algorithm then, at each iteration, best responds to this estimated strategy. Because `fictitious play` needs to calculate a best response, it also assumes complete knowledge of its own payoffs.

Fictitious play is guaranteed to converge to a Nash equilibrium in self play for a restricted set of games. These games are said to have the *fictitious play property* (see, for instance Monderer and Shapley (1996); for an example of a simple 2×2 game without this property see Monderer and Sela (1996)). `Fictitious play` will also eventually best respond to any stationary strategy. This algorithm's general structure has been extended in a number of ways, including *smooth fictitious play* (Fudenberg and Kreps, 1993), and we will see later that `fictitious play` provides the foundation for several more modern algorithms.

`Fictitious play` is known to be subject to miscoordination problems, particularly in self play, and particularly in games that reward asymmetric coordination (e.g., dispersion games). There are some clever measures that can be taken to avoid some of these kinds of problems (e.g., best response tie-breaking rules and randomization), but miscoordination remains a general problem for the `fictitious play` approach.

2.2. Determined

`Determined` or ‘bully’ (see, for example, Powers and Shoham (2005)) is an algorithm that solves the multiagent learning problem by ignoring it. MAL algorithms typically change their behavior by adapting to signals about the game. However, `determined`, as its name suggests, simply relies on other algorithms to adapt their strategies to it.

`Determined` enumerates the stage-game Nash equilibria and selects the one that maximizes its personal reward in equilibrium; then, it plays its corresponding action forever.[2] Certainly, `determined` can lead to some obvious problems. For instance, in self play two `determined` agents can stubbornly play actions from different equilibria, leading to sub-equilibrium average reward. Additionally, enumerating all the Nash equilibria not only requires complete knowledge of every agents’ reward functions, but is also computationally costly, limiting the use of this strategy to relatively small stage games. All the same, `determined` serves as a useful baseline for comparison. Also, slight variations of this algorithm are, like `fictitious play`, at the heart of some more modern algorithms.

2.3. Targeted Algorithms

We next focus on two so-called *targeted* algorithms, which focus on playing against particular classes of opponents. Both these algorithms are based around identifying what the opponent is doing (with particular attention paid to stationarity and Nash equilibrium), and then updating their behavior based on this assessment.

Meta (Powers and Shoham, 2005) switches between three simpler strategies: a strategy similar to `fictitious play`, a `determined`-style algorithm that stubbornly plays a Nash equilibrium, and the maxmin strategy. Strategy selection depends on recorded histories of average reward and empirical distributions of the opponents’ actions across different periods of play. `Meta` was shown both theoretically and empirically to be nearly optimal against itself, close to the best response against stationary agents, and to approach (or exceed) the security level of the game in all cases.

`AWESOME` also tracks the opponent’s behavior in different periods of play and tries to maintain hypotheses about its play. For example, it attempts to determine whether the other algorithm is playing a particular stage-game Nash equilibrium. If it is, `AWESOME` responds with its own component of that special equilibrium. This special equilibrium is known in advance by all implementations of `AWESOME` to avoid equilibrium selection issues in self play. There are other situations where it

[2] The `determined` algorithm need not play an action from a Nash equilibrium. For example, it could instead choose the action whose best response yields the algorithm the highest payoff. Note that this differs from a stage-game Nash equilibrium because the `determined` algorithm need not itself play a best response. Such an outcome amounts to an equilibrium of the Stackelberg version of the stage game. That is, we can change the game so that instead of the two players moving simultaneously, the `determined` agent moves first.

acts in a similar fashion to `fictitious play`, and there are still other discrete modes of play that it engages in depending on its beliefs.

Because both of these algorithms switch between simpler strategies depending on the situation, they can be viewed as portfolio algorithms. Note that both manage similar portfolios that include a `determined`-style algorithm and a `fictitious play` algorithm.

2.4. Q-LEARNING ALGORITHMS

A broad family of MAL algorithms are based on `Q-learning` (Watkins and Dayan, 1992), which is a algorithm for finding the optimal policy in Markov Decision Processes (MDPs). This family of MAL algorithms does not explicitly model the opponent's strategy choices. They instead settle for learning the expected discounted reward for taking an action and then following a stationary policy encoded in the Q-function. In order to learn the Q-function, algorithms typically take random exploratory steps with a small (possibly decaying) probability.

Each algorithm in this family has a different way of selecting its strategy based on this Q-function. For instance, one could try a straight forward adaptation of single-agent `Q-learning` to the multiagent setting by ignoring the impact that the opponent's action makes on the protagonist's payoffs. The algorithm simply updates its reward function whenever a new reward observation is made, where the new estimate is a convex combination of the old estimate and the new information:

$$Q(a_i) = (1-\alpha_t)Q(a_i) + \alpha_t \left[r + \gamma \max_a Q(a) \right]. \tag{1}$$

This algorithm essentially considers the opponent's behavior to be an unremarkable part of a noisy and non-stationary environment. The non-stationarity of the environment makes learning difficult but this idea is not entirely without merit: `Q-learning` has been shown to work in other non-stationary environments (see, for instance, Sutton and Barto (1999)).

`Minimax-Q` (Littman, 1994) is one of the first explicitly multiagent applications of the Q-learning idea. The Q-function that it learns is based on the action profile and not just the protagonist's action: it learns $Q(a_i, a_{-i})$. Minimax-Q uses the mixed maxmin strategy calculated from the Q-function as its strategy:

$$Q(a_i, a_{-i}) = (1-\alpha_t)Q(a_i, a_{-i}) + \alpha_t \left[r + \gamma \max_{\sigma_i \in \prod(A_i)} \left[\min_{a_{-i} \in A_{-i}} \sum_{a_i} \sigma_i(a_i) Q(a_i, a_{-i}) \right] \right]. \tag{2}$$

Such a strategy is sensible to the extent that the protagonist believes that the opponent aims to minimize his payoff, or that the protagonist cares about worst-case guarantees. It should be noted that since its maxmin strategies are calculated from learned Q-values, they may not be the game's actual maxmin strategies and thus fail to reflect the security value. Like `Q-learning`, `minimax-Q` also takes the occasional exploration step.

There are further modifications to this general scheme. `Nash-Q` (Hu and Wellman, 2003) learns different Q-functions for itself and its opponents and plays a stage-game Nash equilibrium strategy for the game induced by these Q-values. `Correlated-Q` (Greenwald and Hall, 2003) does something similar except that it chooses from the set of correlated equilibria using a variety of different selection methods. Both of these algorithms assume that they are able to observe not only the opponents' actions but also their rewards, and additionally that they have the computational wherewithal to compute the necessary solution concept.

2.5. Gradient Algorithms

Gradient ascent algorithms, such as GIGA-WoLF (Bowling, 2004a) and RV$_{\sigma(t)}$ (Banerjee and Peng, 2006), maintain a mixed strategy that is updated in the direction of the payoff gradient. The specific details of this updating process depend on the individual algorithms, but the common feature is that they increase the probability of actions with high reward and decrease the probability of unpromising actions. This family of algorithms is similar to Q-learning because they do not explicitly model their opponent's strategies and instead treat them as part of a non-stationarity environment.

GIGA-WoLF is the latest algorithm in a line of gradient learners that started with IGA (Singh et al., 2000). GIGA-WoLF uses an adaptive step length that makes it more or less aggressive about changing its strategy. It compares its strategy to a baseline strategy and makes the update larger if it is performing worse than the baseline. GIGA-WoLF guarantees non-positive regret in the limit (regret is discussed in greater detail in §6.1) and strategic convergence to a Nash equilibrium when playing against GIGA (Zinkevich, 2003) in two-player two-action games.

There are two versions of GIGA-WoLF. The first version assumes prior knowledge of personal reward and the ability to observe the opponent's action—this is the version used in the proofs for GIGA-WoLF's no-regret and convergence guarantees. There is also a second version—on which all the experiments were based—that makes limited assumptions about payoff knowledge and computational power. Instead, like Q-learning, it merely assumes that it is able to observe its own reward.

RV$_{\sigma(t)}$ (Banerjee and Peng, 2006) belongs to a second line of gradient algorithms that started with ReDVaLeR (Banerjee and Peng, 2004). This algorithm also uses an adaptive step size when following the payoff gradient, like GIGA-WoLF, but does so on an action-by-action basis. This means that, unlike GIGA-WoLF, RV$_{\sigma(t)}$ can be aggressive in updating some actions while being cautious about updating others. These updates are performed by comparing current reward to the reward at a Nash equilibrium. Therefore, RV$_{\sigma(t)}$ requires complete information about the game and sufficient computational power to discover at least one stage-game Nash equilibrium. RV$_{\sigma(t)}$ also guarantees no-regret in the limit and additionally provides some convergence results for self play in a restricted class of games.

GIGA-WoLF and RV$_{\sigma(t)}$ differ in the way that they ensure that their updated strategies remain valid probability distributions. GIGA-WoLF *retracts*: it maps an unconstrained vector to the vector on the probability simplex that is closest in ℓ_2 distance. This approach has a tenancy to map vectors to extreme points of the simplex, reducing some action probabilities to zero. RV$_{\sigma(t)}$ *normalizes*, which is less prone to removing actions from its support.

2.6. Previous Experimental Results

As discussed in the introduction, surprisingly little past work has aimed primarily to use large-scale experiments to compare the performance of MAL algorithms. Nevertheless, a considerable number

Table I. This table shows a summary of the experimental setup for a selection of papers. The summary includes the number of algorithms, the number of game distributions, the number of game instances drawn from these distributions, the number of runs or trials for each instance, and the number of iterations that the simulations were run for. In some cases, the setup was unclear, indicated with a '?'. In many cases, fewer than $[Algorithms \times Distributions \times Instances \times Runs]$ runs were simulated, due to some sparsity in the experimental structures.

Paper	Algorithms	Distributions	Instances	Runs	Iterations
Littman (1994)	6	1	1	?	?
Claus and Boutilier (1997)	2	3	1 - 100	?	50-2500
Greenwald and Hall (2003)	7	5	1	2500 - 3333	1×10^5
Bowling (2004b)	2	6	1	?	1×10^6
Nudelman et al. (2004)	3	13	100	10	1×10^5
Powers and Shoham (2005)	11	21	?	?	2×10^5
Banerjee and Peng (2006)	2	1	1	1	16000
Conitzer and Sandholm (2007)	3	2	1	1	2500

of papers from the literature describe experimental comparisons, often in the context of arguing for a particular MAL algorithm or approach. We briefly survey that literature here.

Setting up a general-sum repeated two-player game experiment requires a number of design choices. What set of algorithms should be considered? On what set of games should these algorithms be run? If one is dealing with randomized algorithms (which includes any algorithm that is able to submit a mixed strategy), how many different runs should be simulated? For a particular game, for how many iterations should a simulation be run? As can be seen in Table I, experiments from the literature varied in all of these dimensions. Additionally, some papers do not describe all experimental parameters, making it difficult to compare results.

Overall, most of the tests performed in these papers considered few algorithms. In most cases, a newly proposed algorithm was evaluated by playing against one or two opponents. Some papers superficially appear to have used many algorithms, but in fact considered algorithms that varied only in small details. For example, in Littman (1994) two versions of `minimax-Q` and two versions of `Q-learning` were tested, with each version differing only in its training regime. In Greenwald and Hall (2003), four versions of `Correlated-Q` were tested against `Q-learning` and `Friend-Q` and `Foe-Q` (Littman, 2001). `Foe-Q` is the same as `minimax-Q`.

To our knowledge, the experiment that considered the greatest variety of algorithms was Powers and Shoham (2005). While four of the eleven algorithms tested in this study were simple stationary-strategy baselines, the remaining seven were MAL algorithms including `Hyper-Q` (Tesauro, 2004), `WoLF-PHC` (Bowling and Veloso, 2002), and a joint action learner (Claus and Boutilier, 1997).

Previous experiments have tended to investigate only small numbers of game instances, and these instances have tended to come from an even smaller number of game distributions. For example, Banerjee and Peng (2006) used only a single 3×3 action "simple coordination game" and Littman (1994) probed algorithm behavior with a single grid-world version of soccer. Initially, this limitation was partly due to the difficulty of creating a large number of diverse game instances. However with the creation of GAMUT (Nudelman et al., 2004), a suite of game generators, generating large game sets is now easy. Indeed, Nudelman et al. (2004) also performed one of the largest previous MAL experiments, using three MAL algorithms (`minimax-Q`, `WoLF` (Bowling and Veloso, 2001), and `Q-learning`) on 100 game instances from each of thirteen distributions. Some recent papers have also leveraged GAMUT, such as Powers and Shoham (2005).

Finally, previous experiments have differed substantially in the number of iterations considered, ranging from 50 (Claus and Boutilier, 1997) to 1×10^6 (Bowling, 2004b). Iterations in a repeated game are typically divided into "settling in" (also called a "burn-in" period) and "recording" phases, allowing the algorithms time to converge to stable behavior before results are recorded. Powers and Shoham (2005) recorded the final $20\ 000$ of $200\ 000$ iterations and Nudelman et al. (2004) used the final $10\ 000$ of $100\ 000$ iterations.

3. Platform

The empirical experiments just described were generally conducted using one-off code tailored to the investigation of a particular feature of a given algorithm. This experimental design has a number of negative consequences. First, it decreases the reproducibility of experiments by, for instance, obscuring the details of algorithm implementation. Even when source code for the original experiment is available, its special-purpose nature can make it difficult to repurpose for follow-on studies or new experiments. Finally, rewriting similar code again and again wastes time that could be spent running more comprehensive experiments.

In this section, we describe our solution to this problem: an open and reusable platform called MALT (MultiAgent Learning Testbed) 2.0. It is available for free download at `http://www.cs.ubc.ca/~kevinlb/malt`. This platform is designed for running two-player, general-sum, repeated-game MAL experiments. Basic visualization and analysis features are also included, as is support for running experiments using a computer cluster. Version 1.0 of MALT was introduced by Lipson (2005); the version described here is a complete reimplementation of that work in a faster programming language (Java vs. Matlab), offering a wide variety of new features, bug fixes, and efficiency gains. Overall, we hope that other researchers will see MALT not as a finished product, but as a growing repository of tools, algorithms and experimental settings, and that they will use it as a base upon which to build (e.g., for the study of N-player repeated games or stochastic games). We have worked hard to make MALT easily extensible. For example, adding a new algorithm to the MALT GUI is as simple as providing a text file with a list of parameters, and adding an algorithm to the engine requires very little coding beyond the implementation of the algorithm itself.

3.1. Definitions

We now define some terms. An ordered pair of two algorithms is a *pairing*. This pair is ordered because many two-player games are asymmetric: the payoff structure for the row player is different than the payoff structure for the column player. The case where an algorithm is paired with a copy of itself (but with different internal states and independent random seeds) is called *self play*.

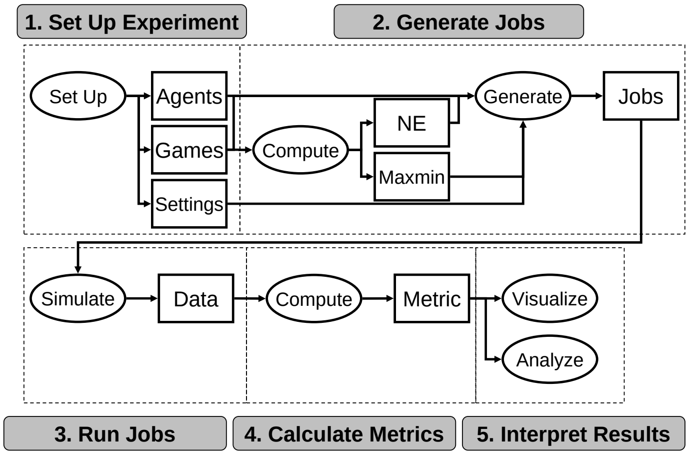

Figure 1: The five steps for running an analyzing an experiment using MALT.

We concentrate on drawing games from distributions called *game generators*. A particular sample from a game generator is a *game instance*. *Prisoner's Dilemma* is a game generator and an example game instance is a particular set of payoffs that obey the *Prisoner's Dilemma* preference ordering. Other game generators are more heterogeneous; for example, one that we will discuss later samples from the space of all strategically distinct 2×2 games.

A pairing and a game instance, taken together, are called a *match*. A match with one of the algorithms in the pairing left unspecified is a *partially specified match* (PSM). If two algorithms play the same PSM, we conclude that any differences between their performances are due to the algorithms themselves (including any internal randomization) because all else is held constant.

A particular simulation of a match is called a *run* or *trial*. For pairs of deterministic algorithms, a single run is sufficient to characterize a match; for randomized algorithms (including any algorithm that plays a mixed strategy) multiple runs may each yield different behavior. In such cases, the match must be characterized by a solution quality distribution (SQD)—the empirical distribution of a performance metric.[3] Each run consists of a number of *iterations*. In each iteration, the algorithms select strategies and then receive some feedback: e.g., their reward; the action choice of their opponent. Algorithms are allowed to select mixed strategies; in this case, a single action is sampled from the mixing distribution by the game. The iterations are separated into *settling-in iterations* and *recorded iterations*.

3.2. Platform Structure

In this section we give an overview of the structure of the platform. The five steps for running an experiment with the platform are summarized in Figure 1. There are three major components to

[3] We use the term SQD because it is standard in the empirical study of algorithms. We note nevertheless that in MAL there is no clear notion of a game having a 'solution', and that these distributions might be more meaningfully called 'metric distributions'.

this platform: the configuration GUI, the experiment engine (the piece that simulates the repeated games) and the visualization GUI. We describe each in turn.

The first step is to set up the experiment. First, a group of algorithms must be picked and algorithm parameters set. Second, a set of GAMUT game distributions must be selected and parameters for these games chosen. Third, general experimental parameters must be established, such as the number of iterations for each simulation. These decisions are encoded in human-readable text files, and can either be generated using a provided GUI or using batch scripts.

The second step is to generate a job file for each desired match. Each job file references the agent, game, equilibrium, and maxmin-strategy files. These files are referenced, making altering the job files simple even after they have been generated.

The third step is to run the jobs. This primarily involves running the MALT "engine"; however, MALT calls GAMBIT's (McKelvey et al., 2004) implementation of Lemke-Howson (Lemke and Howson, 1964) when an algorithm needs to find the set of Nash equilibria for a game instance, and CPLEX when an algorithm needs a maxmin strategy. Jobs may be run in several ways. The most basic is to run them in a batch job on a single machine. However, for large experiments this can be prohibitively expensive. Because each job is independent, it is straightforward to use a compute cluster. To facilitate such parallelization, each job creates an individual data file upon completion that records the history of play. For each recorded iteration and for each agent in the pair, the strategy, sampled action, reward received, and beliefs about the opponents are recorded.

Step four is to compute performance metrics based on these data files. A plain-text file specifies the metrics to be calculated, based on an extensible library of available metrics. As above, metrics can be computed in a batch or can be distributed across a cluster.

Finally, step five is to analyze and visualize these results. To make this task easier, MALT includes some basic analysis tools and a visualization GUI.

3.3. Algorithm Implementations

To carry out this study, we selected and implemented eleven MAL algorithms, most of which we discussed previously in §2.6. In cases where reference code was available, we performed extensive validation experiments to ensure that our implementation was correct.

3.3.1. *Fictitious play*

Parameters for **fictitious play** are given in Table II. We note that the initial action frequencies were set to one for each action, which is a uniform and easily overwhelmed prior. Actions were selected from non-singleton best-response sets by favoring an action that was played in the previous iteration if present, and selecting uniformly at random otherwise.

3.3.2. *Determined*

Our implementation of **determined** (see Table III) repeatedly plays the Nash equilibrium that obtains the highest personal reward, but if there are multiple equilibria with the same protagonist reward, then the equilibrium with the highest opponent reward is selected. If there are any equilibria that are still tied we use the one found first by GAMBIT's implementation of Lemke-Howson.

Table II. Design decisions for `fictitious play`

Design Decision	Setting
BR Tie-Breaking	Previous action if still BR Uniform otherwise
Initial Beliefs	Unit virtual action count

Table III. Design decisions for `determined`

Design Decision	Setting
NE Tie-Breaking	Highest opponent utility

Table IV. Design decisions for `AWESOME`

Design Decision	Setting
Special Equilibrium(π_p^*)	First found
Epoch period ($N(t)$)	$\left\lceil \frac{\lvert A\rvert_\Sigma}{\left(1-\frac{1}{2^{t-2}}\right)\left(\epsilon_e^t\right)^2} \right\rceil$
Equilibrium threshold ($\epsilon_e(t)$)	$\frac{1}{t+2}$
Stationarity threshold ($\epsilon_s(t)$)	$\frac{1}{t+1}$

Table V. Design decisions for `meta`

Design Decision	Setting
Security threshold (ϵ_0)	0.01
Bully threshold (ϵ_1)	0.01
"Generous" BR parameter (ϵ_2)	0.005
Stationarity threshold (ϵ_3)	0.025
Coordination/exploration period (τ_0)	90 000
Initial period (τ_1)	10 000
Secondary period (τ_2)	80 000
Security check period (τ_3)	1 000
Switching probability (p)	0.00005
Window (H)	1 000
$\lVert\cdot\rVert$	ℓ_2

3.3.3. *AWESOME*

AWESOME is implemented according to the pseduo-code in Conitzer and Sandholm (2007), and uses parameter settings given there; see Table IV. For the 'special' equilibrium we use the first equilibrium found by GAMBIT's implementation of Lemke-Howson. It would be interesting to compare our implementation of AWESOME to one that used the more computationally-expensive approach of picking, say, a socially optimal equilibrium.[4]

[4] In our validation experiments we observed a small but statistically significant difference between the behavior of our implementation of AWESOME and the original implementation from Conitzer and Sandholm (2007). (The original implementation was in C and MALT 2.0 is written in Java, so the original implementation could not be used directly.) Specifically, a test involving ten different game instances and 100 runs against the random agent showed a significant difference between solution quality distributions on three instances. We used a two-sample Kolmogorov-Smirnov independence test (see §4.2) with $\alpha = 0.05$ to check for significance. For these three game instances, our implementation probabilistically dominated (see §4.5) the original implementation in terms of reward (i.e., every reward quantile was higher for our implementation). We were not able to track down the source of this behavior difference; however, we spent a considerable amount of time verifying our implementation against the pseudocode in the paper and were unable to find any difference, suggesting that the bug may be in the original C implementation.

Table VI. Design decisions for GIGA-WoLF.

Design Decision	Setting
Learning rate ($\alpha(t)$)	$\frac{1}{\sqrt{\frac{t}{10}+100}}$
Step size ($\eta(t)$)	$\frac{1}{\sqrt{10^4 t+10^8}}$

Table VII. Design decisions for GSA.

Design Decision	Setting
Learning rate ($\alpha(t)$)	$\frac{1}{\sqrt{\frac{t}{10}+100}}$
Step size ($\eta(t)$)	$\frac{1}{\sqrt{10^4 t+10^8}}$
Noise Weight ($\lambda(t)$)	$\frac{1}{\sqrt{10^5 t+10^8}}$

3.3.4. *Meta*

Meta is implemented according to the pseduo-code in Powers and Shoham (2005). The Powers and Shoham (2005) implementation of meta used a distance measure based on the Hoeffding Inequality, even though the pseudo-code called for using an ℓ_2 norm. We follow the pseudo-code and use the ℓ_2 norm. We do not adjust the default threshold level (ϵ_3) for distance, leaving it at the original value. All parameters for metaare summarized in Table V.

3.3.5. *Gradient Algorithms*

Our implementation of **GIGA-WoLF** follows the original pseudo-code and uses the learning rate and step size schedules from the original experiments by Bowling (2004a) as defaults; see Table VI. We note, however, that these step sizes were set for drawing smooth trajectories and may not necessarily yield strong performance, and furthermore that the original experiments for GIGA-WoLF involved more iterations than we simulated (10^6 as compared to 10^5). For GIGA-WoLF's retraction map operation (the function that maps an arbitrary vector in $\Re^n$ to the closest probability vector in terms of ℓ_2 distance) we used an algorithm based on the method described in Govindan and Wilson (2003). GIGA-WoLF has two variants: in one it assumes that it can counterfactually determine the reward for playing an arbitrary action in the previous iteration, and in the other it only knows the reward for the the action that it played and has to approximate the rewards for the other actions. We implemented the latter approach, as all of GIGA-WoLF's experimental results are produced by this version. The formula for the approximation is given by

$$\forall \dot{a} \in A_i \ \hat{r}^{(t+1)}_{\dot{a}} = (1-\alpha) r^{(t)} \mathbb{I}_{\dot{a}=a^{(t)}} + \alpha(\hat{r}^{(t)}_{\dot{a}}). \tag{3}$$

In this equation, $r^{(t)}$ is the reward that the algorithm experienced while playing action $a^{(t)}$ in iteration t. The vector $\hat{r}^{(t)}$ is an $|A_i|$-dimensional vector that reflects the algorithm's beliefs about rewards.

We also tested the Global Stochastic Approximation algorithm, **GSA**, of Spall (2003); see Table VII. To our knowledge we were the first to suggest its use in a MAL setting (Lipson, 2005). This algorithm is a stochastic optimization method that resembles GIGA, but takes a noisy, rather than deterministic, step. The GSA strategy is updated as

$$x^{(t+1)} = P(x^{(t)} + \eta^{(t)} r^{(t)} + \lambda^{(t)} \zeta^{(t)}), \tag{4}$$

Table VIII. Design decisions for $\mathtt{RV}_{\sigma(t)}$.

Design Decision	**Setting**
σ-schedule ($\sigma(t)$)	$\frac{1}{1+\frac{1}{25}\sqrt{t}}$
Step size ($\eta(t)$)	$\frac{1}{\sqrt{1000t+10^5}}$

Table IX. Design decisions for `Q-learning`.

Design Decision	**Setting**
Learning rate ($\alpha(t)$)	$\left(1-\frac{1}{2000}\right)^t$
Exploration rate ($\epsilon(t)$)	$\frac{1}{5}\left(1-\frac{1}{500}\right)^t$
Future discount factor (γ)	0.9

where x_t is the previous mixed strategy, r_t is the reward vector, ζ_t is a vector where each component is sampled from the standard normal distribution (with variance controlled by the parameter $\lambda^{(t)}$), and $P(\cdot)$ is the same retraction function used for `GIGA-WoLF`.

`RV`$_{\sigma(t)}$ is a implementation of the algorithm given in Banerjee and Peng (2006). Some initial experiments showed that the settings of the algorithm used in the paper performed very poorly, and so we used some hand-picked parameter settings that were more aggressive and seemed to perform better. These are given in Table VIII.

3.3.6. *`Q-Learning`*

Our implementation of **`Q-learning`** is very basic.; see Table IX. Since in a repeated game there is only one ‘state’, `Q-learning` essentially keeps track of Q-values for each of its actions. We use an ϵ-greedy exploration policy (perform a random action with probability ϵ) with a decaying ϵ. 400 exploration steps are expected for this ϵ-schedule, and ϵ drops below a probability of 0.05 at approximately iteration 2800. It is negligible at the end of the settling-in period (less than $3E{-}9$). The learning rate (α) decays to 0.01 at the end of the settling in period. The discount factor of $\gamma = 0.9$ was set rather arbitrarily. There is no need to trade off current reward with future reward: all actions take the algorithm back to the same state.

3.3.7. *`Minimax-Q` and `Minimax-Q-IDR`*

For **`minimax-Q`**, we solved a linear program to find the mixed maxmin strategy based on the Q-values. This program was

$$\begin{array}{ll} \textbf{Maximize} & U_1 \\ \textbf{subject to} & \sum_{j\in A_1} u_1(a_1^j, a_2^k)\cdot\sigma_1^j \geq U_1 \quad \forall k \in A_2 \\ & \sum \sigma_1^j = 1 \\ & \sigma_1^j \geq 0 \qquad\qquad\qquad\qquad\qquad \forall j \in A_1 \end{array}$$

(see, for example, Shoham and Leyton-Brown (2008)). We also considered a variant of `minimax-Q` in which iterative domination removal (IDR) is used as a preprocessing step. To our knowledge, we were the first to propose this algorithm in Lipson (2005); we dubbed it `minimax-Q-IDR`. In each step of the iterative IDR algorithm mixed-strategy domination is checked using a linear program (see, for example, Shoham and Leyton-Brown (2008)). Both LPs are solved with CPLEX 10.1.1. For both `minimax-Q` and `minimax-Q-IDR`, the learning rate, exploration rate, and future discount factor were set as in `Q-learning`; see Table IX.

3.3.8. *Random*

The final algorithm, **random**, is an simple baseline that uniformly mixes over the available actions. Specifically, it submits a mixed strategy σ where $\forall a \in A,\ \sigma(a) = \frac{1}{|A|}$.

4. Experimental Setup and Statistical Methods

As described in the preamble, this paper makes two main contributions. The first is the MALT platform, which we have now explained. The second is a demonstration of what MALT can do. Specifically, we conducted an large-scale experiment with the goal of investigating the empirical relationship between average reward and other performance metrics (e.g., equilibrium convergence; regret) that have been considered in the literature. In this section we describe the setup of this experiment and some of the statistical tools we used in our analysis.

We studied all eleven of the algorithms described in §3.3, and set their parameters as described there. We note in passing that this choice was important, as some algorithms are very sensitive to parameter settings. Nevertheless, we considered the issue of parameter optimization to be beyond the scope of our study, and took parameter settings from the literature as given.

We selected thirteen game generators from the GAMUT game collection; these are summarized in Table X. Details of each generator are available in GAMUT's online documentation; see gamut.stanford.edu. We normalized the rewards of all game instances to the $[0, 1]$ interval in order to make the results more interpretable and comparable. We generated a total of 600 different game instances. Specifically, we generated games of five different sizes: 2×2, 4×4, 6×6, 8×8 and 10×10. For each size, we generated 100 game instances, drawing uniformly from the first twelve generators. We drew an additional 100 instances from the last distribution, D13, which spans all strategically distinct 2×2 games (Rapoport et al., 1976). We call the distribution induced by mixing over all 13 GAMUT generators the *grand distribution*.

With eleven algorithms and 600 game instances there were $11 \times 11 \times 600 = 72\,600$ matches. We ran each match once[5] for $100\,000$ iterations, recording the last $10\,000$ iterations. This generated $143\,GB$ of data and took about a third of a CPU-year to run. In order to interpret the results we relied upon a variety of different empirical methods. We briefly describe some of them below.

4.1. Bootstrapping

If we conduct an experiment where two algorithms are run on a number of PSMs then a natural way to compare their performance is to compare the sample means of some measure of their performance (average reward, for example). However, if we have the conclusion that 'the sample

[5] We note that each match could have been run multiple times instead of just once, and indeed that doing so would have been essential if we wanted to understand the behavior of randomized algorithms in individual matches. However, holding CPU time constant, conducting more runs per match would have meant either experimenting with fewer games or with fewer algorithms. Indeed, we show in Appendix A that not stratifying (holding one experimental variable fixed while varying another; as opposed to varying both) on game instances reduces variance for sample estimates of summary statistics like mean and median. Thus, we ran each match only once, and therefore use the terms 'run' and 'PSM' interchangeably in what follows.

Table X. The number and name of each game generator.

D1	A Game With Normal Covariant Random Payoffs
D2	Bertrand Oligopoly
D3	Cournot Duopoly
D4	Dispersion Game
D5	Grab the Dollar
D6	Guess Two Thirds of the Average
D7	Majority Voting
D8	Minimum Effort Game
D9	Random Symmetric Action Graph Game
D10	Travelers Dilemma
D11	Two Player Arms Race Game
D12	War of Attrition
D13	Two By Two Games

mean of algorithm A is higher than the sample mean of algorithm B', how robust is this claim? If we ran this experiment again are we confident that it would support the same conclusion?

A good way to check the results of an experiment is to run it multiple times. For example, imagine that we ran an experiment 100 times and found that 95 of the experiments had a sample mean for algorithm A of between $[\underline{a}, \overline{a}]$, and that 95 of the experiments had a sample mean for algorithm B of between $[\underline{b}, \overline{b}]$. If $\underline{a} > \overline{b}$ (the lower bound of A's interval was greater than the upper bound of B's) then we can be confident that A is significantly better in terms of mean. (Specifically, these intervals are the 95% percent confidence intervals of the sample mean distribution, and the fact that they do not overlap serves as sufficient evidence that there is a significant relationship between the means.)

While such repeated experimentation can be used to ensure that results are significant, it is also expensive. To verify the summary statistics from one experiment, we had to run many more. This is not always possible (e.g., our experiments took 7 days on a large computer cluster, so to rerun them a hundred more times would have taken the better part of two years). Bootstrapping is a technique that allows us to use the data from a *single* experiment to construct confidence intervals of summary statistics. Given an experiment with m data points, we can 'virtually' rerun the the experiment by subsampling from the empirical distribution defined by those m points. For example, if we have a sample with 100 data points, we could subsample 50 data points (with replacement) from these 100 and look at the statistic for this subsample. We can cheaply repeat this procedure as many times as we like, creating a distribution for each estimated statistic. From these bootstrapped estimator distributions we can form bootstrapped confidence intervals and check for overlap.

There are two parameters that control the bootstrapped distribution: we form the distribution by subsampling l points from the original m, and we repeat this process k times. For our analysis we chose l to be $\lfloor m/2 \rceil$ and k to be around 2 500. These particular parameters were chosen to ensure that there would be diversity among the subsamples (this explains the moderate size of l) and that the empirical distributions would be relatively smooth (this explains the large k).

4.2. Kolmogorov-Smirnov Test

While bootstrapping is useful for seeing if summary statistics are significantly different, we will also want to check if two distributions are themselves significantly different. A beta distribution and a Gaussian distribution might coincidentally have the same mean, but are nevertheless different distributions. We use the KolmogorovSmirnov (KS) test for determining whether two distributions are different. This test is nonparametric, meaning that it does not assume that the underlying data is drawn from a known (e.g., normal) probability distribution. The KS test works by examining the maximum vertical distance between two CDFs. Two distributions are considered significantly different if this maximum vertical distance exceeds a given significance level, α. In our analysis we use the standard $\alpha = 0.05$ unless otherwise noted.

4.3. Spearman's Rank Correlation Test

Spearman's rank correlation test is a way to establish whether or not there is a significant monotonic relationship between two paired variables. For example, we might want to show that there is some significant monotonic relationship between the size of a game's action set size and an agent's average reward. Like the KS test, the Spearman's rank correlation test is non-parametric. The relationship between the two variables can be positive (high values of one variable are correlated with high values of the other variable) or negative (high values of one variable are correlated with low values of the other).

4.4. Assessing Convergence

We are interested in studying the convergence behavior of MAL algorithms. One issue in doing so based on empirical data is dealing with runs that appear "not quite" to have converged because of random fluctuations in the empirical action frequency. A natural solution to this problem is to perform a statistical test to determine whether one part of the run exhibits the same action distribution as a later part. For example, we might check whether a later empirical action distribution was drawn from the same distribution as an earlier sample (establishing that empirical mixed strategies were stationary) or that an empirical action distribution profile was drawn from a given mixed-strategy profile (establishing convergence to a Nash equilibrium).

Two obvious candidates for such a test are the Fisher exact test (FET) and Pearson's χ^2-test, which can be used for checking whether two multinomial samples are drawn from a distribution. However, each test was unfortunately inappropriate for our problem. The χ^2 test does not handle situations where some of the actions are rare or not present. The FET is very computationally expensive, and the implementation of it that we used (R Development Core Team, 2006) failed on some of the larger and more balanced action vectors (typically in the 10×10 case).

Instead, we used the incomplete set of FET results to calibrate a threshold based on vector distance, where we considered any two vectors that were closer than the threshold θ to be the same. We calibrated θ using a receiver operating characteristic (ROC) curve. We used the incomplete

FET results as ground truth, and plotted the change in true positive rate and false positive rate as we varied θ. We picked the threshold that led to an equal number of false positives and false negatives. Based on this ROC analysis, we picked a θ of 0.02.

4.5. Probabilistic Domination

The concept of probabilistic domination can be used to argue that one distribution should be preferred to another in terms of a given performance metric. Specifically, a solution quality distribution (SQD) A dominates another SQD B if $\forall q \in [0, 1]$, the q-quantile of A is higher than the q-quantile of B. If there are two algorithms, A and B, that are trying to maximize reward, and A's SQD probabilistically dominates B then regardless of the reward value r, there are more runs of A than of B that attain a reward of at least r. Probabilistic domination is stronger than a claim about the mean of the distributions: domination implies higher means.

Checking for probabilistic domination between two samples can be performed visually. If one of the CDF curves is below the other curve everywhere, than the former dominates the latter. Intuitively, this is because the better SQD has less probability mass on low solution qualities, and more mass on higher solution qualities; better distributions are right-shifted.

5. Empirical Evaluation of MAL Algorithms: Average Reward

As we discussed at the beginning of this paper, we consider average reward to be the most fundamental metric for assessing the performance of a MAL algorithm. We take the average with respect to the sampled actions rather than the submitted mixed strategy. Formally, where the iterations 1 to T refer to the $10\ 000$ iterations we recorded, we define the average reward an algorithm i obtains in a single match as $\bar{r}_i^{(T)} = \frac{1}{T}\sum_{t=1}^{T} r_i^{(t)}$.

In this section, we investigate the average reward metric in detail. We begin in §5.1 by comparing algorithms according to their "raw" average reward, averaging also across both generators and opponents. Next, we investigate each of these dimensions separately. In §5.2 we explore algorithm performance across different generators, and also examine the effect of game size. In §5.3 we explore algorithm performance across different opponents, and also analyze the equilibria of the "algorithm game", in which available actions are different choices of MAL algorithms. §5.4 investigates probabilistic domination relationships between different algorithms and §5.5 considers each algorithm's performance in self play. Finally, §5.6 explores similarities between different algorithms.

5.1. "Raw" Average Reward

First we consider each algorithm's "raw" performance, averaged across both games and opponents.

OBSERVATION 1. *`Q-Learning` and $\mathtt{RV}_{\sigma(t)}$ attained the highest rewards on the grand distribution.*

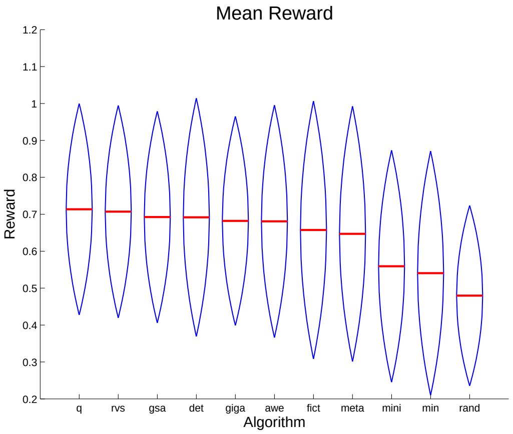

Figure 2: A plot that shows the mean reward (bar) for each algorithm and one standard deviation in either direction (the size of the lens).

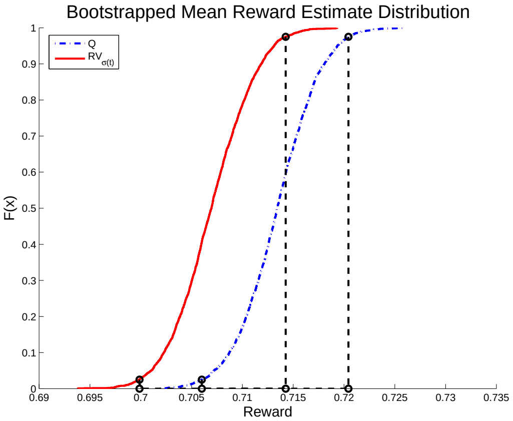

Figure 3: The distribution of mean reward estimates for Q-learning and $\mathsf{RV}_{\sigma(t)}$, constructed by bootstrapping. The 95% confidence intervals are indicated by the dark circles and dashed-lines.

Q-learning had the highest mean reward at 0.714, although $\mathsf{RV}_{\sigma(t)}$ was close with an average of 0.710 (see Figure 2). We noticed considerable variation within the reward data, and all of the other algorithms' sample means still were within one standard deviation of Q-learning, including random (which obtained a sample mean of 0.480).

These rankings were not all significant. The slight difference in means between Q-learning and $\mathsf{RV}_{\sigma(t)}$ does not in fact indicate that Q-learning was a better algorithm (in terms of means) on the grand distribution of games and opponents. These two algorithms attained significantly higher reward than any other algorithm, however. We determined this by examining the 95% percentile intervals on bootstrapped mean estimator distributions (see §4.1) and seeing which intervals overlapped (see Figure 3). We obtained the distributions by subsampling $2\,500$ times, where each subsample consisted of $6\,600$ runs (half as many as the $13\,200$ runs that each algorithm participated in).

The distribution of reward was not symmetric, and specifically tended to exhibit negative skewness, indicating that the proportion of runs that attained high reward was larger than the proportion of runs that attained low reward. (random was the only exception). Q-learning's distribution had the highest skewness, -0.720.

5.2. Per-Generator Average Reward and the Effect of Game Size

Now we go beyond performance on the grand distribution. First we consider each algorithm's performance across individual game distributions. As can be seen in Figure 4, every algorithm's performance varies considerably across the different game generators. However, this figure makes it difficult to determine the best algorithm for generators that all algorithms found challenging. Thus, we also present a normalized version of these per-generator reward results, obtained by dividing the results for each algorithm on a particular generator by the maximum reward attained by any

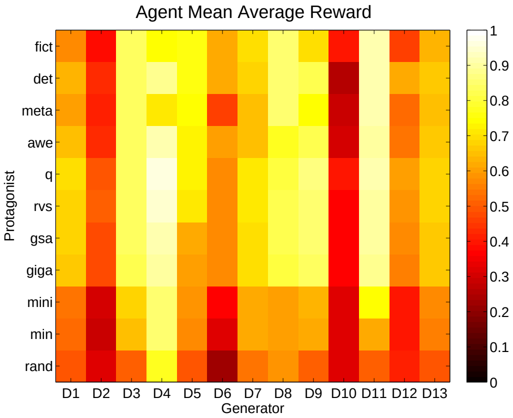

Figure 4: A heatmap showing the reward for the protagonist algorithm playing PSMs from a particular generator, averaged over both iterations and PSMs.

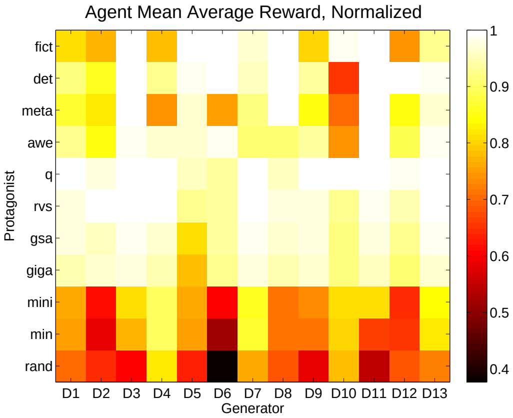

Figure 5: A heatmap showing the mean reward for the protagonist algorithm, playing against the opposing algorithm. These cells have been normalized. Each column has been divided by the maximum average reward attained by any algorithm on that particular generator.

algorithm (Figure 5). We can see that minimax-Q, minimax-Q-IDR and random were all worse than the other algorithms across a broad range of generators, and Q-learning and $\mathrm{RV}_{\sigma(t)}$ tended to do well.

OBSERVATION 2. *Q-Learning was the best or one of the best algorithms to use for most generators.*

We define the set of best algorithms for a generator as the set of algorithms whose bootstrapped mean estimator 95% percentile intervals overlapped with the algorithm with the best sample mean. Q-Learning was the unique best algorithm or was one of the best algorithms for 10 of our 13 generators (see Table XI). It was the only algorithm that was the unique best choice for any generator, taking this role for generators D1, D4, and D9. Furthermore, Q-learning also belonged to the set of best algorithms for generators D2, D3, D7, D10, D11, D12 and D13. While Q-learning most frequently was a member of a generator's best algorithm set, fictitious play and determined were also frequently in these sets (6 and 7 generators respectively).

The gradient algorithms were especially strong on D7; indeed, this was the only generator for which all three gradient algorithms were in the best algorithm set. D5, D6, and D8 were interesting distributions for AWESOME and meta. In D5, neither AWESOME nor meta managed to be one of the best algorithms despite the fact that both fictitious play and determined—two of the algorithms that they manage—were. In D6, AWESOME joined fictitious play and determined but meta did not, and in D8 the reverse happened: meta, fictitious play and determined were the three best algorithms. These three generators illustrate situations where portfolio algorithms failed to capitalize on one of their managed algorithms. It would be interesting to run further experiments to determine why this occurred and if it could be remedied.

Table XI. The set of best algorithms for each generator.

Gen	Set of Best Algorithms
D1	Q-learning
D2	Q-learning, RV$_{\sigma(t)}$
D3	AWESOME, determined, fictitious play, GSA, meta, Q-learning, RV$_{\sigma(t)}$
D4	Q-learning
D5	determined, fictitious play
D6	AWESOME, determined, fictitious play
D7	GSA, Q-learning, RV$_{\sigma(t)}$
D8	determined, fictitious play, meta
D9	Q-learning
D10	fictitious play, Q-learning
D11	determined, fictitious play, meta, Q-learning
D12	determined, Q-learning
D13	AWESOME, determined, GSA, Q-learning, RV$_{\sigma(t)}$

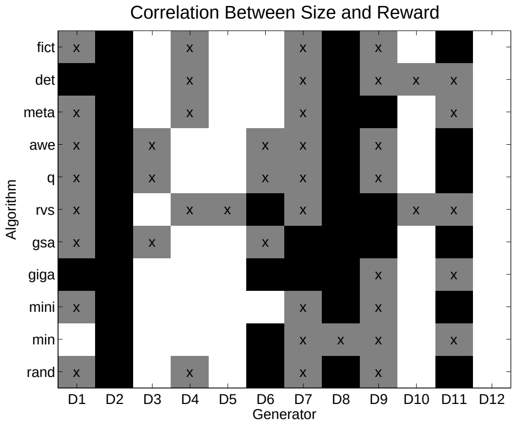

Figure 6: A heatmap summarizing the correlations between size and reward for different agents on different generators. A white cell indicates positive correlation, a black cell indicates negative correlation, and a gray cell with an ‘x’ indicated an insignificant result.

For all but one of our generators (D13: 2×2 games) we generated games of varying sizes. Now we consider how the size of a game’s action set affected performance. Our hypothesis was that larger action spaces raise the possibility of more complicated game dynamics, and that such complex dynamics can slow learning. Thus, we expected to see average reward decreasing as the size of the game grew.

OBSERVATION 3. *There was no general relationship between game size and reward: for some generators algorithms achieved higher rewards on larger games, and for other generators algorithms achieved higher rewards on smaller games.*

Our experiment showed that this intuition did not always hold. First, for many algorithms on many generators we could not reject the null hypothesis of the Spearman rank correlation test—that there was no significant correlation between size and performance—at a significance level of $\alpha = 0.05$. For instance, in D7 only GSA and GIGA-WoLF had significant trends (both exhibited negative correlation; reward was lower in larger games). Second, even when a significant correlation did exist, it was not always negative. We did observe that for most distributions, significant correlations were either entirely negative or entirely positive. For D2, D7, D8, D9, and D11 the correlation was negative; for D3, D4, D5, D10, and D12 it was positive. D1 and D6 exhibited both kinds of correlation for different algorithms.

Overall, the relationship between game size and reward appears to depend strongly on the choice of generator. It could be the case that when the action spaces increase in size, important game features tied with high reward become more common, or it could be that larger actions spaces

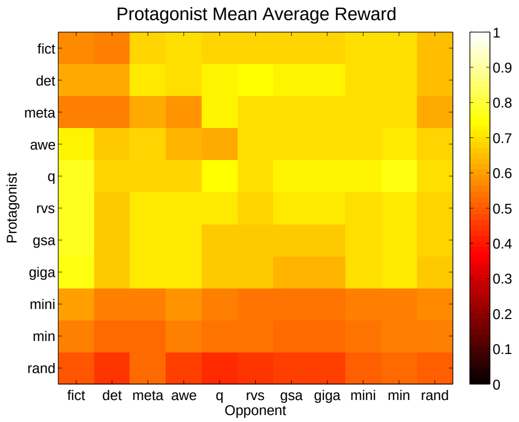

Figure 7: A heatmap showing the mean reward for each protagonist algorithm (ordinate) playing against each opposing algorithm (abscissa).

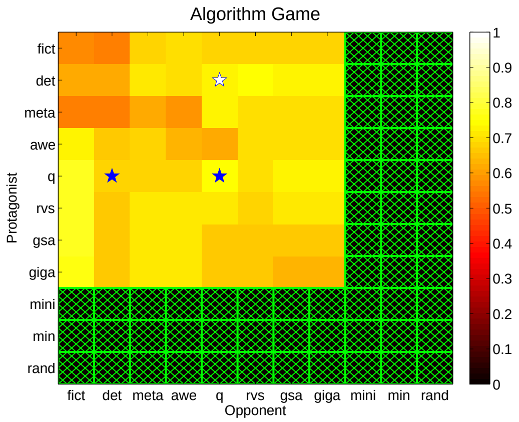

Figure 8: Interpreting the mean reward results as a one-shot game. The cells that are cross-hatched are dominated and the '⋆'s indicate pure-strategy Nash equilibria. Because the `determined` vs. `Q-learning` equilibrium is asymmetric, it appears twice. To indicate this, we make one of the corresponding stars hollow.

make it easier for MAL algorithms to miscoordinate, which is desirable for some games. Indeed, D4—*Dispersion Games*—are show positive correlation between the number of actions and reward, and this is a game where agents need to miscoordinate to do well.

As Figure 6 shows, D2 and D12 were the only two distributions on which we could reject the null hypothesis for all algorithms, and they supported opposite conclusions. On instances from D2, correlation was completely and strongly negative: the larger the game, the worse every algorithm performed. The least correlated algorithm was `random` with a Spearman's coefficient of correlation $\rho = -0.329$. Correlation was entirely positive for D11, but some of the coefficients were smaller. `Fictitious play` was the least sensitive to size ($\rho = 0.07$), but it was anomalous. The algorithm with the next smallest coefficient was `GIGA-WoLF`, with $\rho = 0.267$.

5.3. Per-Opponent Average Reward and the Algorithm Game

We now consider each algorithm's average reward on a per-opponent basis.

OBSERVATION 4. *Algorithm performance depended substantially on which opponent was played.*

Figure 7 shows the mean reward achieved by each algorithm against every possible opponent. One striking feature of this figure is that `minimax-Q`, `minimax-Q-IDR` and `random` were all relatively weak against a broad range of opponents. We also observe that `fictitious play` and `determined` tended to get lower reward in self play and against each other than against other opponents. `Meta`—an algorithm that manages a profile of algorithms including `fictitious play` and `determined`—also appear to have inherited these performances issues, while `AWESOME`—the other portfolio algorithm—substantially avoided them.

Table XII. The different algorithms and their best-response sets

Opponent	Best-Response Set
AWESOME	GIGA-WoLF, GSA $\mathrm{RV}_{\sigma(t)}$
Determined	AWESOME, GIGA-WoLF, GSA, Q-learning $\mathrm{RV}_{\sigma(t)}$
Fictitious play	GSA, Q-learning $\mathrm{RV}_{\sigma(t)}$
GIGA-WoLF	determined, Q-learning $\mathrm{RV}_{\sigma(t)}$
GSA	determined, Q-learning $\mathrm{RV}_{\sigma(t)}$
Meta	determined, GIGA-WoLF, GSA $\mathrm{RV}_{\sigma(t)}$
Minimax-Q	Q-learning
Minimax-Q-IDR	Q-learning
Q-Learning	determined, Q-learning $\mathrm{RV}_{\sigma(t)}$
Random	determined, Q-learning $\mathrm{RV}_{\sigma(t)}$
$\mathrm{RV}_{\sigma(t)}$	determined

Table XIII. The proportion of subsampled algorithm games in which each algorithm was strictly dominated (SD) or weakly dominated (WD).

Algorithm	SD	WD
AWESOME	10.8%	11.7%
Determined	0.0%	0.0%
Fictitious play	35.9%	36.4%
GIGA-WoLF	54.1%	55.1%
GSA	0.4%	0.4%
Meta	28.8%	28.2%
Minimax-Q	100.0%	100.0%
Minimax-Q-IDR	100.0%	100.0%
Q-Learning	0.0%	0.0%
Random	100.0%	100.0%
$\mathrm{RV}_{\sigma(t)}$	0.0%	0.0%

If we know what algorithm the opponent is using, which algorithm should we use? We constructed “best-response sets” for each possible opponent using bootstrapped percentile intervals. We call the algorithm with the highest mean against a particular opponent a best response, but also assign any algorithm with a overlapping bootstrapped 95% percentile interval to the set—we cannot claim that these algorithms do significantly worse than the apparent best algorithm. These best response sets are summarized in Table XII. Q-learning and $\mathrm{RV}_{\sigma(t)}$ were most frequently best responses, while fictitious play, meta, minimax-Q, minimax-Q and random were never best responses.

One interesting way to interpret these best response results is to consider the one-shot “algorithm game”: a single-shot normal-form game in which the actions correspond to our 11 algorithms and the payoff for using algorithm A against algorithm B is the mean reward that algorithm A attained against B. There were three algorithms that were strictly dominated in this grand distribution algorithm game: minimax-Q, minimax-Q-IDR and random. Strict domination of algorithm A' by A means that regardless of what algorithm the opponent selects, A is always a better choice than A'. As with best responses, we required domination to be significant: we wanted to be confident that if the experiment were repeated, we would get a similar result. We used bootstrapping to check this, subsampling 6 600 PSMs 10 000 times and from these forming 10 000 ‘subsampled’ games. We checked for strict domination in each game, and considered an algorithm dominated if it was dominated in at least 95% of the subsampled games. The proportion of subsampled algorithm games in which each algorithm was dominated is shown in Table XIII; we also distinguish strict domination from weak domination.

OBSERVATION 5. *Determined and Q-learning were the only algorithms to participate in pure-strategy Nash equilibria of the algorithm game.*

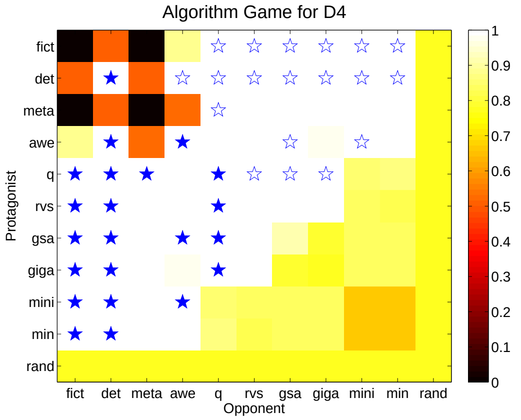

Figure 9: Interpreting the mean reward results for D4 (*Dispersion Game*) as a one-shot game. No cells were dominated; the '⋆'s indicate pure-strategy Nash equilibria. Asymmetric equilibria appear twice; to indicate this we make one of the corresponding stars hollow.

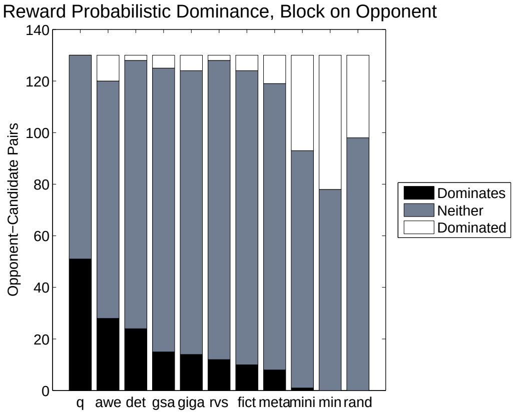

Figure 10: For each algorithm, the number of opponents and candidate algorithms the algorithm dominated, was dominated by, or neither.

Only two pure-strategy Nash equilibria ever occurred in the subsampled games for the grand distribution: Q-learning in self play, and Q-learning against determined. The Q-learning–Q-learning equilibrium is particularly convincing because it is symmetric and so does not require that the players coordinate to playing different strategies, and furthermore because it occurred in 90.2% of the subsampled games. The other equilibrium occurred in the remaining 9.8% of games. (Because both equilibria involved Q-learning, we did not observe them together in the same subsampled games.)

We looked more deeply into the algorithm games by restricting attention to individual generators. The generators varied substantially in their pure-strategy Nash equilibria. Overall, Determined in self play constituted the most common symmetric pure-strategy Nash equilibrium. It was a significant Nash equilibrium for seven of the generators. (That is, determined in self play was a pure-strategy Nash equilibrium in more than 95% of the subsampled games for these each of these generators.) Q-Learning in self play was the second most common symmetric pure-strategy Nash equilibrium, arising in the algorithm games for four generators.

Generators also differed substantially in their *number* of pure-strategy Nash equilibria. For instance D1 (*A Game with Normal Covariant Payoffs*) had no significant pure-strategy Nash equilibrium. D4 (*Dispersion Game*), at the other extreme, had 22 pure-strategy Nash equilibria (see Figure 9). Part of the reason for the large number of equilibria in $D4$ was that a majority of runs for many of the algorithms yielded a reward of 1 (e.g., 84.6% of AWESOME's runs yielded a reward of 1). This meant that in many of the subsampled games, the majority of payoffs were exactly 1 and so there were many weak Nash equilibria. For example, both RV$_{\sigma(t)}$ and Q-learning attained a reward of 1 against fictitious play, and fictitious play itself attained a reward of 1 against RV$_{\sigma(t)}$ and fictitious play. Therefore both RV$_{\sigma(t)}$–fictitious playand Q-learning–fictitious play were pure Nash equilibria.

5.4. Probabilistic Domination of One Algorithm by Another

Now we consider the following question: given a fixed opponent, is a given algorithm probabilistically dominated by any alternative algorithm in terms of average reward?

OBSERVATION 6. *`Q-Learning` was the only algorithm that was never probabilistically dominated by any other algorithm when playing any opponent.*

`Q-Learning` had the best performance in terms of probabilistic domination. `Determined` and $\texttt{RV}_{\sigma(t)}$ were the next-least-dominated algorithms: `determined` was only probabilistically dominated by `AWESOME` against a `fictitious play` opponent, which was in turn dominated by `Q-learning`; $\texttt{RV}_{\sigma(t)}$ was dominated by `Q-learning` when playing against the `minimax-Q` variants, and also by `determined` when playing against $\texttt{RV}_{\sigma(t)}$. On the whole, domination by another algorithm in self play was a common trend; only `AWESOME`, `determined` and `Q-learning` avoided being dominated by another algorithm when playing themselves. It is interesting that `determined` was not dominated: we see this as a property of the specific game distributions that we studied.

Overall, while we observed some strong domination relationships, these were the exceptions while ambiguity was the rule. For most algorithm pairs against most opponents, no probabilistic domination relationship existed (see Figure 10). Furthermore, there was no opponent for which one algorithm probabilistically dominated all others.

5.5. Self Play

We have already seen evidence that self-play was challenging for many algorithms (e.g., see the tendency towards ‘cool’ cells on the main diagonal of Figure 7). A closer analysis shows that for most algorithms there was indeed a significant relationship between self play and low reward.

OBSERVATION 7. *Most algorithms attained lower average reward in self play.*

The distribution of reward in self-play runs for `AWESOME`, `determined`, `fictitious play` and `meta` were probabilistically dominated by the distribution of reward in non-self-play runs. While the same was not true for the gradient algorithms (they achieved fewer low-reward runs in self play), their self-play means were nevertheless significantly lower than their non-self-play means. We verified this by looking at the 95% bootstrapped percentile intervals. There was no significant relationship for `minimax-Q` and `minimax-Q-IDR`, and this self-play trend was reversed for `Q-learning`: its self-play runs probabilistically dominated its non-self-play runs. Furthermore, `Q-learning` achieved a higher mean reward in self play than any other algorithm (see Figure 11).

Interestingly, `AWESOME` was one of the algorithms with poorer self-play runs, despite its machinery for converging to a special equilibrium in self play. We wonder whether this occurred because `AWESOME` did not converge due to an overly-conservative threshold for detecting whether its opponent was playing part of an equilibrium, or because `AWESOME` did converge to the special equilibrium but that equilibrium did not offer high reward. (Note that our implementation

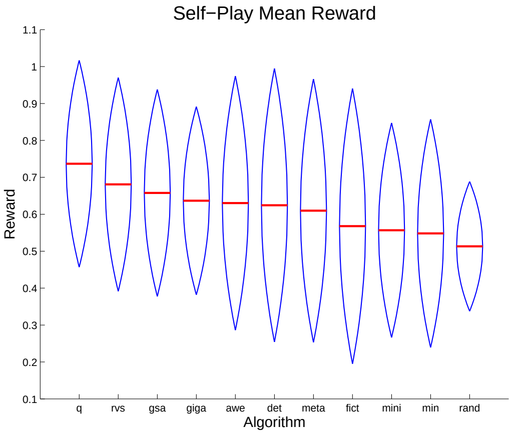

Figure 11: A plot that shows the mean reward (bar) for each algorithm in self play and one standard deviation in either direction (the size of the lens).

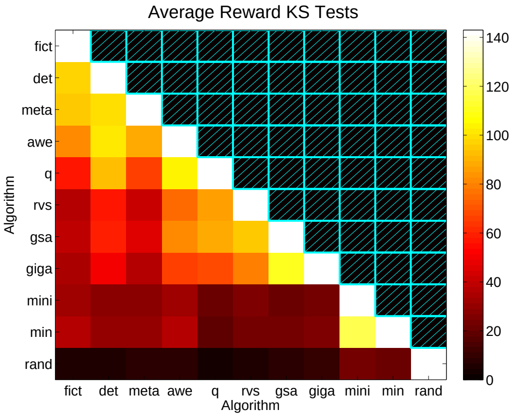

Figure 12: A heatmap that summarizes the number of opponent/generator pairs two algorithms are similar on in terms of reward distribution. This relationship is symmetric, so only the lower half of the plot is presented. The hotter the cell, the more situations the two algorithms are similar in.

of AWESOME coordinates to the first Nash equilibrium found by GAMBIT's implementation of Lemke-Howson.) At the risk of keeping the reader in suspense, we defer the answer to §6.3, in which we examine equilibrium convergence results.

5.6. Algorithm Similarity

Finally, we investigate similarities between algorithms' abilities to achieve high reward. We can assign some of our algorithms to one of three major blocks. First, AWESOME and meta are similar because they both manage portfolios incorporating fictitious play and determined; likewise, we expect them to be similar to the fictitious play and determined algorithms themselves. Second, GIGA-WoLF, GSA and $\mathrm{RV}_{\sigma(t)}$ are similar because they all follow a reward gradient. Finally, minimax-Q and minimax-Q-IDR are similar because the latter is the same as the former except for the addition of an IDR preprocessing step. We call these the portfolio, gradient, and minimax blocks. We also might suspect that Q-learning, an algorithm that does not explicitly model the opponent, might be similar to the gradient algorithms. Nevertheless, we do not assign Q-learning to a block; likewise, we leave random unassigned.

We tested all pairs of algorithms for similarity by comparing their average reward distributions for all generator–opponent pairs. Thus, we tested each algorithm pair $13 \times 10 = 130$ times—every algorithm is of course similar to itself and so we did not check these cases. Failing to reject the null hypothesis of the KS test (that both samples were drawn from the same population) is some evidence for the samples being similar. This rough-and-ready approach does not establish significant similarity and is merely suggestive of similarity; failing to reject a null hypothesis is not the same as having shown that the null hypothesis is true. However, with this caveat in mind, we observed some interesting trends.

OBSERVATION 8. *Similar algorithms tended to exhibit similar performance.*

All three predicted blocks emerge, as can be seen in Figure 12. First, `meta`, `AWESOME`, `fictitious play` and `determined` were all similar to each other on a large number of opponent–generator pairs. Both `meta` and `AWESOME` were similar in more cases to `determined` than to `fictitious play`. For instance, `AWESOME` was similar to `determined` in 101 out of 130 cases while similar to `fictitious play` in only 81 cases. `Meta` and `AWESOME` were also quite similar to each other (88 cases). `Q-learning` was similar to the algorithms in this block, especially `determined` and `AWESOME`, which we had not expected. `AWESOME` was more similar to `Q-learning` than to any other algorithm: they were similar in 103 cases, while even `determined` and `AWESOME` were only similar in 101 cases.

The block of algorithms consisting of $\texttt{RV}_{\sigma(t)}$, `GIGA-WoLF` and `GSA` were all similar in a large number of cases, with a particularly tight relationship evident between `GIGA-WoLF` and `GSA` (similar in 111 cases). `Q-Learning` also bore similarities to the gradient-algorithm block. These algorithms also showed somewhat weaker similarity to `determined` and `AWESOME`.

The connection between `minimax-Q` and `minimax-Q-IDR` was particularly strong (similar in 118 cases). These were also the algorithms most similar to `random`—indeed, similar almost twice as often as the next-most-similar algorithm (`AWESOME`: it was similar to `random` in 11 cases, as compared to `minimax-Q`'s 21 cases).

6. Empirical Evaluation of MAL Algorithms: Other Metrics

So far, all of our experimental discussion has concerned the average reward metric. However, a wide variety of other metrics have also been proposed and studied in the literature. Here we consider many of the most prominent. This allows us to understand our experimental results in different ways, and furthermore sheds light on the extent to which each metric correlates with high reward in practice. In §6.1 we investigate regret, specifically considering mean regret, probabilistic domination of one algorithm by another, and the relationship to reward. In §6.2 we assess algorithms' tendencies to converge to stationary strategies. §6.3 considers convergence to Nash equilibrium of the stage game, and relates this metric to reward. In §6.4 we consider algorithms' abilities to achieve at least their maxmin payoffs, and consider both per-opponent maxmin performance and the relationship to reward. Finally, in §6.5, we measure algorithms' tendency to converge to payoff profiles consistent with Nash equilibria of the infinitely-repeated stage game.

6.1. Regret

Regret is the difference between the reward that an algorithm could have received by playing the best static pure strategy and the reward that it did receive:

$$Regret(\vec{\sigma}_i, \vec{a}_{-i}) = \max_{a \in A_i} \sum_{t=1}^{T} \left[r(a, a_{-i}^{(t)}) - \mathbb{E}\left[r(\sigma_i^{(t)}, a_{-i}^{(t)}) \right] \right]. \quad (5)$$

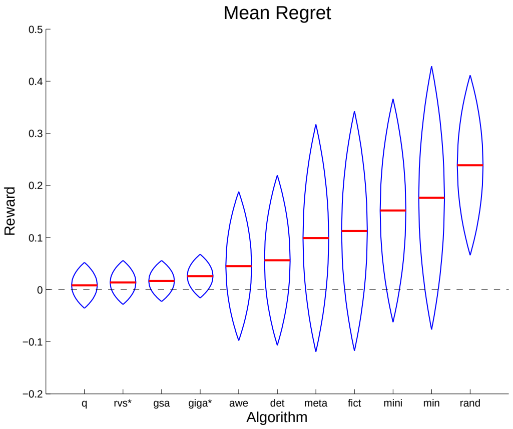

Figure 13: A plot that shows the mean regret (bar) for each algorithm and one standard deviation in either direction (the size of the lens). Algorithms with an asymptotic no-regret guarantee are indicated '$*$'.

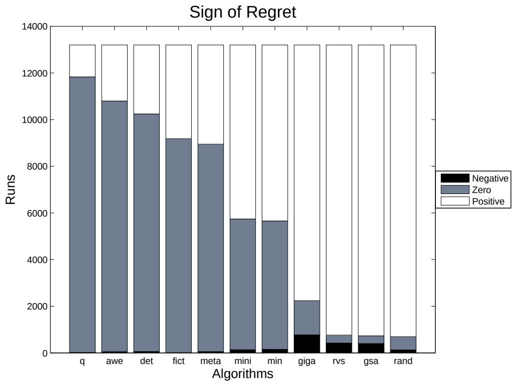

Figure 14: The number of runs in which each algorithm achieved negative, zero, or positive regret.

The best static pure strategy is determined after the run, based on the assumption that the opponent's actions choices in each round would not change. We use the expected reward formulation of regret—as opposed to one that uses the actual actions that the algorithm played—following Bowling (2004a). Rather than looking at the total sum of regret over all $10\,000$ recorded iterations, we will discuss the mean regret over these iterations. Since player payoffs are restricted to the $[0, 1]$ interval, mean regret can give a better sense of the magnitude of regret with respect to possible reward.

Regret has been suggested as a measure of how exploitable an algorithm is. If an agent accrues significant regret one possible explanation is that it did the wrong thing. However, in some games (e.g., the Traveler's dilemma) ignoring regret can lead to greater long-term reward.

Some algorithms, including GIGA-WoLF and $\mathrm{RV}_{\sigma(t)}$, are *no-regret* learners: they come with the guarantee that they will always approach zero regret as the number of iterations approaches infinity. However, to our knowledge it has not been shown experimentally how the regret achieved by these algorithms compares to the regret achieved by other algorithms that lack such a guarantee; nor has it been demonstrated whether these algorithms achieve better than zero regret in practice.

OBSERVATION 9. *Q-Learning best minimized regret. GIGA-WoLF most frequently achieved negative-regret runs.*

In our experiment, all algorithms achieved positive mean regret (Figure 13), though they differed substantially in the fraction of their matches in which they achieved positive regret (Figure 14). All the means were significantly different, based on overlaps in the 95% percentile intervals (there was none). Of these, Q-learning had the lowest regret, at 0.008. The gradient algorithms—GIGA-WoLF, GSA and $\mathrm{RV}_{\sigma(t)}$—had the next lowest mean regret after Q-learning. Among

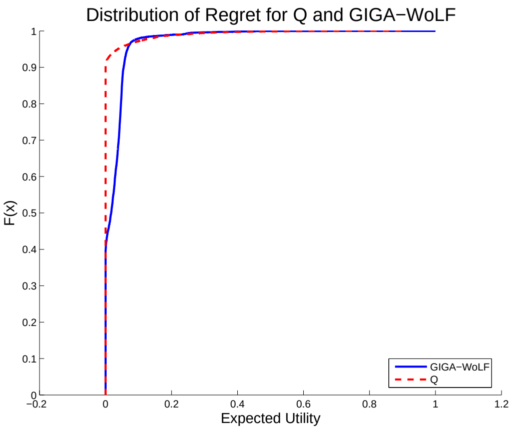

Figure 15: The distribution of regret for `Q-learning` and `GIGA-WoLF`.

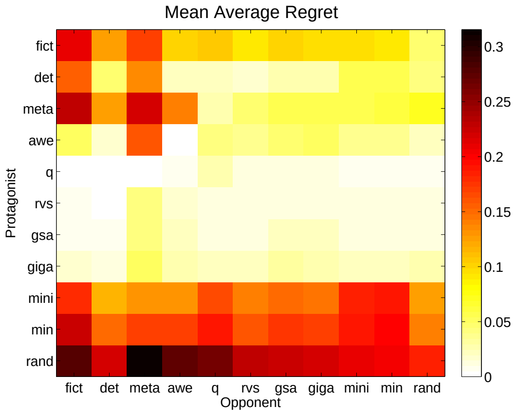

Figure 16: Mean average regret, blocked by opponent.

the gradient algorithms, `GSA` achieved the lowest mean regret, followed by $\mathsf{RV}_{\sigma(t)}$ and then by `GIGA-WoLF`. These empirical results are concordant with `GIGA-WoLF` and $\mathsf{RV}_{\sigma(t)}$'s theoretical no-regret guarantees—not only are these algorithms guaranteed zero regret in the limit, but they also achieved low regret in practice. At the same time, it is interesting that the algorithm with the best results, `Q-learning`, comes with no such guarantee.

Considering only mean regret masks an interesting difference between `Q-learning` and the gradient algorithms: they achieve low mean regret in different ways (see Figures 14 and 15). `Q-Learning` achieved low mean regret by attaining zero regret in most (89.5%) of its runs. It had the fewest positive-regret runs (10.4%; the next lowest was `AWESOME` at 18.2%), and also had the second-fewest negative-regret runs (0.1%; only `fictitious play` had (slightly) fewer). On the other hand, the gradient algorithms rarely achieved zero regret (the algorithms with the fewest zero runs were $\mathsf{RV}_{\sigma(t)}$, `GSA`, `random` and `GIGA-WoLF`) but often achieved negative regret (the three algorithms with the most negative regret runs were `GIGA-WoLF` (5.8%), $\mathsf{RV}_{\sigma(t)}$ (3.2%) and `GSA` (3.0%)).

Overall, no algorithm achieved less than very slightly negative regret: the very smallest was an average regret of -2×10^{-6}. The converse was not true for positive regret: in 440 different runs some algorithm attained average regret of 1, meaning that it took precisely the wrong action in every round. 48.6% of these runs involved `fictitious play` or one of the algorithms that wrap around `fictitious play`(awesome or `meta`) in self play, and were on generator D4 (*Dispersion Games*), which reward miscoordination. We can conclude that in these cases `fictitious play` became stuck in pathological cycling between the symmetric outcomes (where both agents play the same action), which yield zero reward. Such cycling is a well-known problem with `fictitious play`; based on claims in the literature, a judicious application of noise to the algorithm would have broken the cycle and improved `fictitious play`'s performance.

Considering regret on a per-generator basis, `Q-learning` achieved the lowest mean regret on every generator except for D13 (strategically distinct 2×2 games), on which $\mathsf{RV}_{\sigma(t)}$ was the

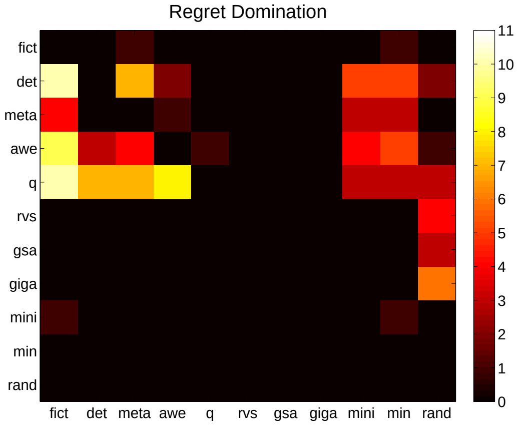

Figure 17: The number of opponents for which the algorithm on the ordinate probabilistically strictly dominates the algorithm on the abscissa. For example, `Q-learning` probabilistically dominates `fictitious play` on PSMs involving ten out of eleven possible opponents.

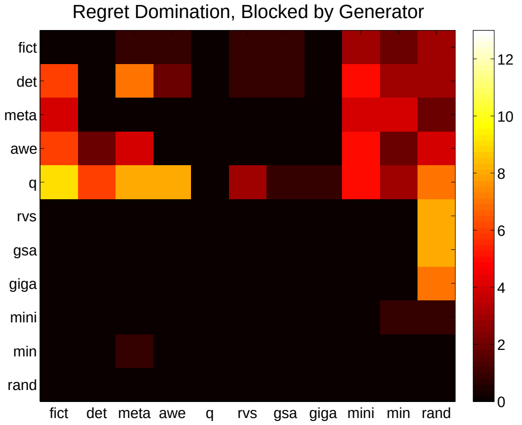

Figure 18: The number of generators for which the algorithm on the ordinate probabilistically strictly dominates the algorithm on the abscissa.

best. `Q-learning` was also the best algorithm to use against almost every opponent. There were only two exceptions: $\mathsf{RV}_{\sigma(t)}$ was better against `Q-learning` and `AWESOME` was better against itself. Another interesting pairing was when `Q-learning` played against `fictitious play`: `Q-learning` attained zero regret in every single game. This indicates that `Q-learning` (uniquely among our algorithms) converged to a pure-strategy best response in every game against `fictitious play`.

6.1.1. *Probabilistic Domination of One Algorithm by Another*

When we consider regret distributions on a per-opponent basis, some strong probabilistic dominance trends emerge.

OBSERVATION 10. *On a per-opponent basis, `Q-learning`, `GIGA-WoLF`, `GSA` and $\mathsf{RV}_{\sigma(t)}$ were rarely probabilistically dominated in terms of regret.*

First, say that algorithm A dominates B k times if there are k opponents C such that A's regret distribution for matches against C probabilistically dominates B's regret distribution for matches against C. Under this notion of domination, we found that the gradient algorithms were never dominated by any other algorithm (Figure 17). `Q-learning` was only dominated once, by `AWESOME` in the case of an `AWESOME` opponent. We were not surprised by this, since `AWESOME` has special machinery for converging to a stage-game Nash equilibrium in self play. (In a Nash equilibrium, of course, both agents play best responses to each other and hence both accrue zero regret.) On the other hand, `fictitious play` was frequently dominated, especially by `AWESOME`, `determined`, `Q-learning` and to a lesser degree `meta`. Both `determined` and `Q-learning` dominated `fictitious play` against 10 opponents (`Q-learning` was the

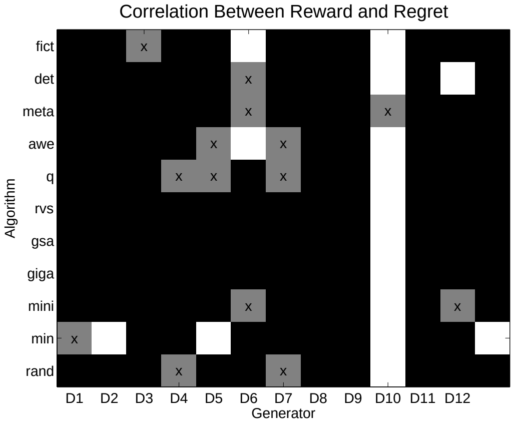

Figure 19: The sign of correlation between reward and regret for each algorithm and each game generator. A white cell indicates positive correlation, a black cell indicates negative correlation, and a grey cell with an ‘x’ indicates insignificant correlation.

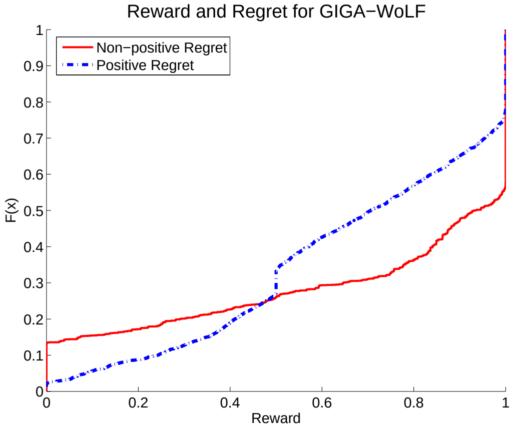

Figure 20: A CDF plot showing `GIGA-WoLF`’s average reward obtained on runs in which it obtained either positive or non-positive reward. Notice that positive-regret runs were less likely to yield zero reward.

exception for `determined` and vice versa), and `AWESOME` dominated `fictitious play` on 9 opponents (`GIGA-WoLF` and `meta` were the only opponents for which `AWESOME` did not dominate `fictitious play`).

We can also define probabilistic domination in another way, saying that algorithm A dominates B k times if there are k *generators* G such that A’s regret distribution on games from G probabilistically dominates B’s regret distribution on games from G. Considering domination in this sense, we can draw similar conclusions (Figure 18). `Q-Learning` dominated other algorithms frequently—particularly `fictitious play` (on 9 generators), `meta` (8 generators), and `AWESOME` (on 8 generators)—while avoiding domination by any other algorithm. `Fictitious play` was dominated frequently: by `Q-learning` (9 generators), `determined` (6), `AWESOME` (6) and `meta` (4).

6.1.2. *Links Between Regret and Reward*

What is the connection between regret and reward? We expected that high reward should be correlated with low regret, and vice versa. This intuition was largely supported by our experimental data. Regret and reward were negatively correlated for all algorithms (Spearman’s rank correlation test; $\alpha = 0.05$): high reward was linked with low regret. On a per-generator basis, we observed that D10 (*Traveler’s Dilemma*) induced *positive* correlation between regret and reward for all algorithms except `determined` (Figure 19). This makes sense: in this game, algorithms do better when they do not play best responses, and indeed the unique Nash equilibrium is one of the worst outcomes of the game.

We compared the average reward each algorithm obtained in positive-regret runs and non-positive-regret runs. For most of the algorithms, the distribution of average reward obtained in non-positive-regret runs probabilistically dominated the distribution of average reward obtained in positive regret runs. There were some exceptions. For example, Q-learning exhibited a relatively minor crossover. The same phenomenon occurred with GIGA-WoLF, but in a more pronounced fashion: runs that attained positive regret less often attained zero reward (Figure 20). Even more dramatically, the positive-regret run distributions probabilistically dominated the non-positive run distributions for GSA and $\mathrm{RV}_{\sigma(t)}$. These two (gradient) algorithms exhibited behavior different from the other nine: runs with positive regret had better reward characteristics than runs with zero or negative regret. This phenomenon did not seem to arise in the context of a single generator or opponent. However, we did note that the probabilistic domination seemed the weakest when PSMs involving *Traveler's Dilemma* were omitted.

6.2. Strategic Stationarity

All of the metrics we have discussed so far have been based on reward. We now consider several that are based on empirical frequency of action, and that ask whether these frequencies converge. The first—and weakest—notion of convergence that we consider measures whether or not an algorithm converges to a stationary strategy profile. This is interesting in its own right, and is also a necessary condition for stronger forms of convergence.

We consider a run to have been stable if the joint distribution of actions was the same in the first and second halves of the recorded iterations, tested according to the threshold criterion described in §4.4 and using ℓ_∞-distance. Stability is a property of a run rather than a single algorithm's play in a run, so even algorithms that always play stationary strategies can still participate in unstable runs.

To check how successful our threshold criterion was at detecting stationarity, we began by examining the results for our two algorithms that always play stationary strategies. Our criterion found determined to be stable in 99.5% of self-play matches and random to be stable in 92.0% of self-play matches. When playing each other, they were found to be stable 94.8% of the time. The differences between these cases are likely because determined tends to adopt mixed strategies with smaller supports than randomdoes, and such a mixed strategy is more likely to yield an empirical action distribution that closely resembles it.[6]

We found GIGA-WoLF and GSA to be the least likely to be stable—particularly in self play, against each other, or against meta (see Figure 21). Their striking instability with meta was potentially because they tripped meta's internal stability test and changed its behavior. However, AWESOME also has a similar internal check, but the stability of GIGA-WoLF and GSA were not noticeably different between matches with AWESOME and with Q-learning (which has no such check). $\mathrm{RV}_{\sigma(t)}$, the other gradient algorithm, was more stable than GIGA-WoLF and GSA. This might be because $\mathrm{RV}_{\sigma(t)}$ had a more aggressive step length: the parameters used in this experiment for GIGA-WoLF and GSA were taken from (Bowling, 2004a), who indicated that these parameters were intended to produce smooth trajectories rather than fast convergence.

Meta, determined, fictitious play and AWESOME were, for the most part, quite good at achieving stationarity. Meta and fictitious play were particularly strong against each other, and always reached a stationary strategy profile. The only exception to the rule of stability

[6] We note that a false positive rate of between 0.5% and 8% is larger than might be hoped, but nevertheless defer consideration of improved criteria for measuring empirical convergence to future work.

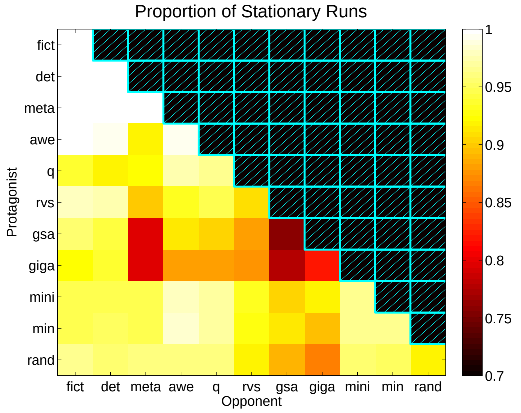

Figure 21: Proportion of stationary runs, blocked on opponent. This intensity map is symmetric; we removed redundant entries for clarity.

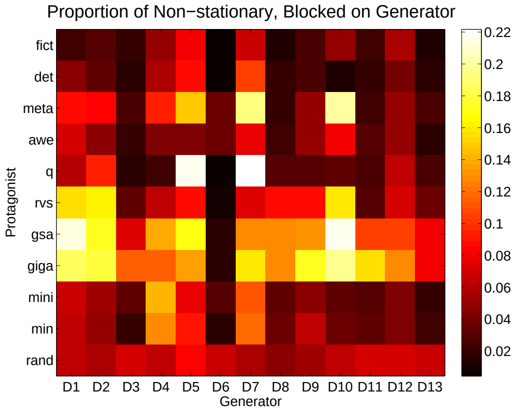

Figure 22: Proportion of non-stationary runs, blocked on generator and protagonist

in this group was `AWESOME` vs. `meta`; this pairing was unstable in 10.3% of runs. We are not sure why this occurred, but conjecture that it arose because of the discrete behavioral changes that both algorithms undergo when their internal states are updated.

There were a number of problem generators for the different algorithms (see Figure 22). For example: generators D1, D2, and D10 created instances that were particularly difficult for the gradient algorithm in terms of strategic stability; `Q-Learning` was weak on both D5 and D7; and `meta` tended to be unstable on D5, D7 and D10. However these unstable instances were rare regardless of the algorithm paring. The vast majority of runs found a stationary strategy profile. Even `GIGA-WoLF`, which was the algorithm least likely to stabilize, found stationarity in 87.0% of its runs (see Figure 23).

6.3. Convergence to Stage-Game Nash Equilibrium

Stable runs are those that converged to any strategy; we now consider which of these selected a (possibly mixed-strategy) stage-game Nash equilibrium. For some algorithms, Nash equilibrium convergence was reasonably common. `AWESOME` converged in 54.3% of its runs, and `determined` converged in 53.1% of its runs. `Determined` was better at `AWESOME` at converging to a Pareto-optimal Nash equilibrium (a Nash equilibrium that was not Pareto-dominated by any other Nash equilibrium). `AWESOME` most frequently converged to a Pareto-dominated equilibrium. This was likely influenced by the way that our implementation of `AWESOME` picked its ‘special’ equilibrium:[7] the first equilibrium found by the Lemke-Howson algorithm, without attention to whether it was, e.g., Pareto-dominated. `AWESOME` also tended to attain lower reward when it converged to a Pareto-dominated Nash equilibrium than when it did not converge or converged to a non-dominated Nash equilibrium.

[7] The original paper, Conitzer and Sandholm (2007), left the method of picking the ‘special’ equilibrium unspecified.

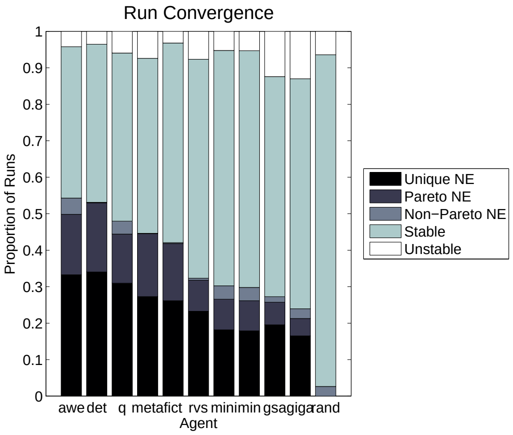

Figure 23: The proportion of runs that were stationary, converged to a non-Pareto-optimal Nash equilibrium, or converged to a Pareto-optimal Nash equilibrium.

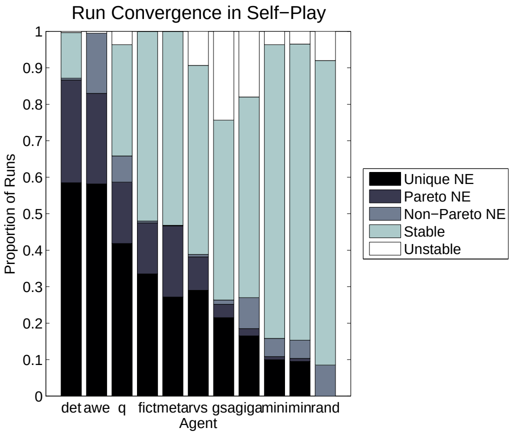

Figure 24: The proportion of self-play runs that were stationary, converged to a non-Pareto-optimal Nash equilibrium, or converged to a Pareto-optimal NE.

Figure 24 shows the extent to which each algorithm converged to a stage-game Nash equilibrium in self play. Notice how often determined converged: this indicates that the games we studied often possessed one Nash equilibrium that was the best for both agents. Indeed, we can see that a surprisingly high number of games had a *unique* stage-game Nash equilibrium (58.5%). We expect that convergence results would look qualitatively different with generators that were much less likely to produce games with unique equilibria.

Observe that AWESOME nearly always converged. Recall that we previously found that AWESOME received lower average reward in self-play than non-self-play runs ($\S$ 5.5). Now we can conclude that this failure to achieve high rewards was not due to a failure to reach equilibrium. An interesting modification of the AWESOME algorithm would be to use its self-play machinery to converge to stable strategies that are not stage-game Nash equilibria, such as the socially-optimal outcome or the Stackelberg game equilibrium. The aim of this adjustment would be to improve self-play reward results while maintaining AWESOME's resistance to exploitation by other algorithms.

6.3.1. *Links Between Nash Equilibrium Convergence and Reward*

Much work in the literature has aimed at MAL algorithms that converge to a stage-game Nash equilibrium. However, if the goal is high average reward, is such convergence desireable? More generally, is proximity to stage-game Nash equilibrium correlated with obtaining high reward?

OBSERVATION 11. *Strategic proximity to stage-game Nash equilibrium was correlated with average reward for all algorithms and most algorithm–generator pairs.*

For all algorithms, reward was negatively correlated with ℓ_∞-distance to the closest Nash equilibrium (Spearman's rank correlation test; $\alpha = 0.05$). Furthermore, most algorithms were negatively correlated even on a per-generator basis (Figure 25). The most notable exceptions were D6, D12, and (especially) D10, where we saw *positive* correlations between distance and reward.

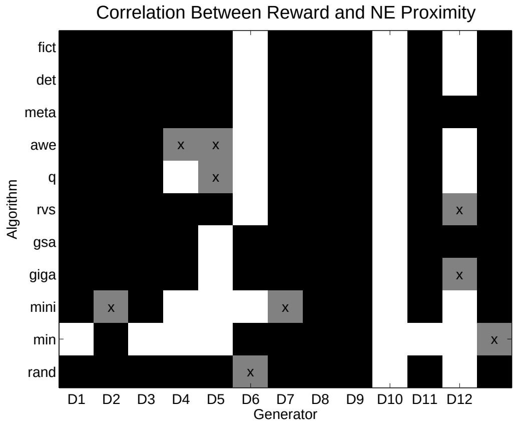

Figure 25: The sign of correlation between reward and ℓ_∞-distance to the closest Nash equilibrium for each algorithm and each game generator. A white cell indicates positive correlation, a black cell indicates negative correlation, and a grey cell with an 'x' indicates insignificant correlation.

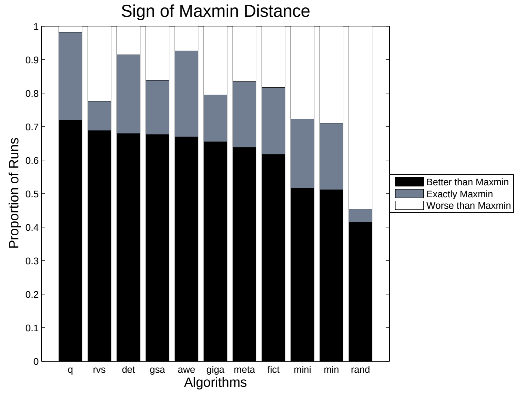

Figure 26: The sign of the maxmin distance of each run, by algorithm.

6.4. Maxmin Distance

An agent's maxmin value is the largest amount that it can guarantee itself regardless of its opponent's behavior. Thus, achieving average reward of at least this amount is widely seen as a necessary condition for sensible MAL behavior. Furthermore, the famous Folk Theorem of game theory demonstrates that enforceable payoffs (those with non-negative maxmin distances) are precisely those payoffs that can be achieved in equilibrium of an infinitely repeated game. We build on our results here to investigate this notion of convergence in §6.5. In this section we consider the difference between average reward and the maxmin value of the underlying game instance:

$$MaxminDistance(\vec{r}_i) = \frac{\sum_{t=1}^{T} r_i^{(t)}}{T} - \max_{a_i \in A_i} \min_{a_{-i} \in A_{-i}} u(a_i, a_{-i}). \qquad (6)$$

We call this difference *maxmin distance*, noting that it can be negative.

OBSERVATION 12. *`Q-Learning` attained an enforceable payoff more frequently than any other algorithm.*

`Q-Learning` most frequently attained an enforceable payoff, with a negative maxmin distance in only 1.8% of its runs (Figure 26). The runs on which `Q-learning` failed to attain an enforceable payoff mostly came from either D4 (*Dispersion Game*; 37.6% of `Q-learning`'s unenforceable runs) or D13 (*Two by Two Game*; 33.3%). They also occurred predominantly against `random` (29% of the unenforceable runs), `minimax-Q` (17.3%) and `minimax-Q-IDR` (16.0%). The next-best algorithm, `AWESOME`, attained enforceable payoffs considerably less often, with a negative maxmin distance in 7.4% of its runs.

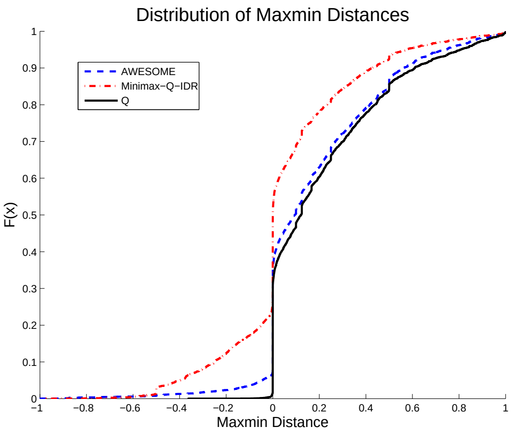

Figure 27: The distribution of maxmin distances for `AWESOME`, `minimax-Q` and `Q-learning`.

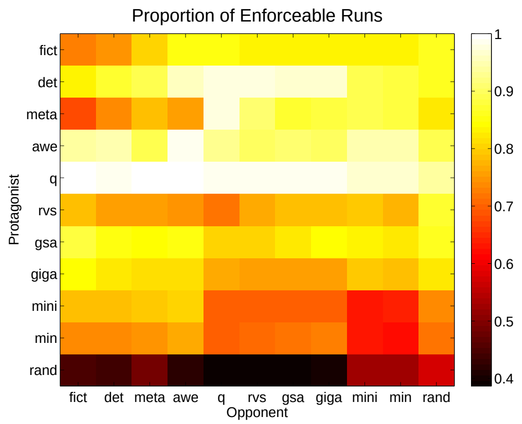

Figure 28: The proportion of enforceable runs, blocked by opponent.

After `random`, `Minimax-Q` and `minimax-Q-IDR` were the *least* likely to attain enforceable payoffs, failing to do so in 28.9% and 27.7% of their runs respectively. This is interesting because these algorithms explicitly attempt to do well against adversarial opponents. One possible explanation is that they may have trouble learning accurate payoffs , leading them to have difficulty obtaining their maxmin values.

`Minimax-Q` and `minimax-Q-IDR` were especially poor in self play, where conservative play can impair payoff learning. There is also a greater proportion of enforceable runs on 2×2 games (75.2%) than on 10×10 games (68.5%)—larger games have more payoffs to learn. Working on a more sophisticated exploration scheme looks like an especially promising place to improve our implementation of `minimax-Q` and its variant.

While `Q-learning` was successful against a broad range of opponents, some other algorithms were less consistent. For example, `meta` was quite good against all opponents except for `fictitious play`, `determined`, `AWESOME` and itself. `Meta` was especially bad against `fictitious play`; in this pairing only 68.0% of `meta`'s runs were enforceable. Compare this to `meta`'s excellent performance against `Q-learning`, where it attained enforceable payoffs in 97.7% of it runs. `Fictitious play` also had trouble playing against `meta`, `determined` and itself. On the other hand, neither `AWESOME` nor `determined` shared this problem.

$\mathsf{RV}_{\sigma(t)}$ had problems attaining enforceable runs too, and although it received payoffs well above the maxmin value frequently (it had the second highest proportion of runs with strictly positive distances at 68.8%) there were also a large number of instances where $\mathsf{RV}_{\sigma(t)}$'s maxmin distance was close to but below zero. This contrasts with `GIGA-WoLF`, which had fewer non-enforceable runs with greater negative minimax distance (see Figure 29). We conjecture that this phenomenon occurred because $\mathsf{RV}_{\sigma(t)}$ maintains a small amount of probability mass on all of its actions, causing it to 'tremble'. More specifically, $\mathsf{RV}_{\sigma(t)}$, like all gradient algorithms, updates its mixed strategy by moving along the reward gradient. When the updated vector does not sum to one, it must be mapped back to the probability simplex. $\mathsf{RV}_{\sigma(t)}$ does this by normalizing the updated vector, while

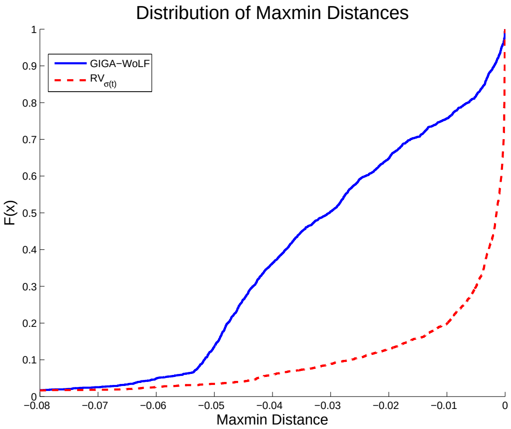

Figure 29: The distribution of negative maxmin distances for GIGA-WoLF and RV$_{\sigma(t)}$.

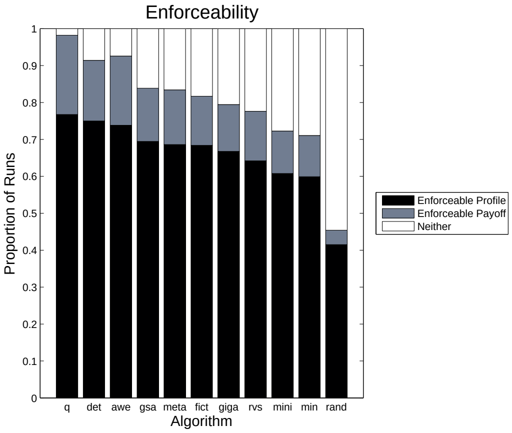

Figure 30: Proportion of PSMs with enforceable payoffs and payoffs profiles achieved, by algorithm.

GSA and GIGA-WoLF use a retraction operator that tends to drop actions from the mixed strategy's support (see § 2.5). We conjecture that modifying RV$_{\sigma(t)}$ to use GIGA-WoLF's retraction operator would improve RV$_{\sigma(t)}$'s ability to achieve enforceable payoffs.

6.4.1. *Links Between Maxmin Distance and Reward*

Is there a connection between enforceable runs and high average rewards? It would being strange if some such relationship did not exist, since enforceability implies reward higher than the maxmin value. Indeed, we did observe that for all algorithms, maxmin distance was positively correlated with average reward. (Spearman's rank correlation test (§4.3); $\alpha = 0.05$ significance level). On a per-generator basis, we again largely observed significant positive correlations. There were two deviations from this pattern. First, we found no significant correlation for half of the algorithms on D11, and for minimax-Q on D3. Second, there was a significant *negative* correlation for minimax-Q on D11, though minimax-Q-IDR still exhibited significant positive correlation.

6.5. Convergence to Repeated-Game Nash Equilibrium

In §6.3 we considered algorithms' tendencies to converge to equilibria of the stage game. The algorithms actually played a repeated game, however. We now turn to analyzing this repeated game's properties. The payoff profiles achievable in Nash equilibrium of a repeated game are precisely the enforceable profiles (see, e.g., Osborne and Rubinstein (1994)). In order to determine whether a given strategy profile is an equilibrium of a repeated game, it is also necessary to consider how these strategies behave off the equilibrium path (e.g., how they punish deviations by the other agent). While the algorithms that we studied lack punishment mechanisms, it is still meaningful to assess how frequently they converged to payoff profiles consistent with repeated game Nash equilibria. We therefore build on the results from § 6.4, asking how often *both* algorithms achieved enforceable payoffs.

OBSERVATION 13. *`Q-Learning` was involved in matches whose payoff profiles were consistent with a repeated game Nash equilibrium more often than any other algorithm.*

Of the algorithms that we examined, `Q-learning` most frequently had runs that were consistent with a repeated game Nash equilibrium (Figure 30). It was consistent with a repeated game equilibrium in 76.8% of its runs. `Determined` and `AWESOME` were the next most frequently consistent (75.0% and 73.8% of their runs respectively). Overall, consistency with a repeated game Nash equilibrium was common, but not universal. It is worth emphasizing that an enforceable payoff profile depends on both agents' actions, and so the behavior of weak agents like `random` lowered the scores for stronger opponents.

7. Discussion and Conclusion

In this article we described MALT, a standardized testbed for multiagent experimentation. This testbed allows researchers to focus on experimental design and analysis instead of implementation. We also presented an in-depth analysis of a large experiment we conducted ourselves using MALT.

The most striking conclusion from our experiment was that `Q-learning` achieved consistently excellent results, in many senses outperforming algorithms based on deeper insights about the multiagent setting (e.g., `GIGA-WoLF`, `AWESOME`, and `meta`). We were surprised by this finding, since we had taken for granted the idea that modern, multiagent algorithms would do better in a repeated-game environment than a classical, single-agent algorithm. The evidence we have shown to the contrary suggests that it should be possible to considerably improve the empirical performance of MAL algorithms. We suggest four areas in which efforts could be worthwhile.

First, a more experimentally-driven focus seems crucial. Our experiment was large, but there are many empirical questions that it does not answer. Some promising future directions include:

- More examination of the relationship between performance and game properties like size;
- More detailed investigation of algorithm behavior on instances from single generators;
- Investigation of additional algorithms like Hyper-Q (Tesauro, 2004) and Nash-Q (Hu and Wellman, 1998);
- Study of N-player repeated games and stochastic games (along the lines of Vu et al. (2005)).

Second, the more sophisticated algorithms have many tunable parameters. Finding optimal settings for them was beyond the scope of our paper, and we instead relied on published parameter settings. Nevertheless, it is possible that some algorithms would have performed considerably better if they had been configured differently. Indeed, `Q-learning` had only three parameters and all were easy to set, which might partly explain its strong performance. Tuning the other algorithms would require considerable experimental effort; hopefully MALT will be of assistance. There are some interesting questions to ask:

- Is one parameter setting good for many problems, or is it the case that some parameter settings are effective on some matches and poor on others?
- Which of an algorithm's (e.g., `meta`'s) parameters are the most important?

- Does AWESOME's performance change radically when it selects the socially optimal Nash equilibrium as its special equilibrium? How about the 'Stackelberg' equilibrium?
- For gradient algorithms, is it better to perform retraction or normalization?
- Do parameter settings that yield high reward also yield low regret?

Third, we presented two different tweaks to existing algorithms: minimax-Q-IDR and GSA. These algorithms offered several improvements over their "parent" algorithms, and in many cases probabilistically dominated them. It would be interesting to explore similar modifications of other existing algorithms.

Finally, managing a portfolio of existing algorithms seems like a promising approach for designing algorithms with good empirical properties. AWESOME and meta can both be seen as portfolio algorithms: they switch between different components based on the opponent's behavior. Much remains to be learned about the best framework for building portfolio algorithms, especially if we insist on frameworks that do not require hand-construction of a portfolio. Again, this direction of research invites a host of empirical questions. What features of a game and of game play should a portfolio track? In what situations does adding an algorithm to a portfolio improve performance?

Acknowledgements

Thanks to Nando de Freitas for his involvement in the early stages of this project, and for helping us to develop the argument in Appendix A. Thanks to Holger H. Hoos and Yevgeniy Vorobeychik for feedback on drafts, and to anonymous reviewers at MLJ for helpful suggestions. Finally, thanks to Vincent Conitzer for providing us with code for AWESOME and helping us to use it.

Appendix A. Independent vs. Stratified Sampling

For all of the experiments described in this article, we were concerned with the expected performance of a match, denoted by $f(\mu, \zeta)$. Here, f is some metric function, $\mu \sim M$ is a match, and $\zeta \sim Z$ is a random seed that completely determined any non-deterministic behavior in both algorithms. The game instance/seed pairing uniquely define a run. When designing our experiment, we needed to choose whether to stratify runs based on the match. For instance, if we had enough time to run 100 simulations, we could either have sampled a single run on 100 matches, or 10 runs on 10 matches. Stratification clearly yields more detailed data about the role that randomization plays in each match. However, for estimating common summary statistics—means and quantiles—stratification should be avoided.

Formally, consider two schemes of sampling from M and Z. Under *independent sampling*, M and Z are sampled separately each time, yielding a set of samples $\{(M_1, Z_1), \ldots, (M_n, Z_n)\}$. Under *stratified sampling*, k samples are taken from M and for each sample of M, Z is sampled s_i times, yielding a set of samples $\{(M_1, Z_{1,1}), \ldots, (M_1, Z_{1,s_1}), \ldots, (M_k, Z_{k,s_k})\}$. In both schemes, the sample mean is used as an estimate for the population mean. Since G and Z are sampled independently, both schemes yield unbiased estimators. However, the following result shows that the schemes differ in terms of variance.

LEMMA 7.1. *Independent sampling yields a lower-variance estimate of* $\mathbb{E}\left[f_{(M,Z)}\right]$ *than stratified sampling.*

Proof First, independent random variables have no covariance.

$$Cov\left[f(M_i, Z_i), f(M_j, Z_j)\right] = Cov\left[f(M_k, Z_{k,l}), f(M_m, Z_{m,n})\right] \tag{7}$$

On the other hand, if two samples share the same stratum (the same sample $\mu \sim M$) then they have weakly higher covariance.

$$Cov\left[f(M_k, Z_{k,l}), f(M_k, Z_{k,m})\right] \geq Cov\left[f(M_i, Z_i), f(M_j, Z_j)\right] \tag{8}$$

Using Equations (7) and (8) we can write

$$\begin{aligned} Var\left[\sum_i f(M_i, Z_i)\right] &= \sum_{i,j} Cov\left[f(M_i, Z_i), f(M_j, Z_j)\right] \\ &\leq \sum_{i,j,k,l} Cov\left[f(M_i, Z_{i,j}), f(M_k, Z_{k,l})\right] \\ &= Var\left[\sum_{i,j} f(M_i, Z_{i,j})\right]. \qquad \square \end{aligned}$$

We also claimed that stratifying increases the variance of quantile point estimation. This result can be found (albeit without proof) in Heidelberger and Lewis (1984).

References

Airiau, S., S. Saha, and S. Sen: 2007, 'Evolutionary Tournament-Based Comparison of Learning and Non-Learning Algorithms for Iterated Games'. *Journal of Artificial Societies and Social Simulation* **10**(3), 7.

Axelrod, R.: 1987, 'The Evolution of Strategies in the Iterated Prisoner's Dilemma'. In: L. Davis (ed.): *Genetic Algorithms and Simulated Annealing*. Morgan Kaufman, Los Altos, CA, pp. 32–41.

Banerjee, B. and J. Peng: 2004, 'Performance bounded reinforcement learning in strategic interactions'. In: *AAAI 11*.

Banerjee, B. and J. Peng: 2006, 'RV: a unifying approach to performance and convergence in online multiagent learning'. In: *AAMAS '06*. pp. 798–800.

Bowling, M.: 2004a, 'Convergence and no-regret in multiagent learning'. In: *NIPS 17*.

Bowling, M.: 2004b, 'Convergence and no-regret in multiagent learning'. Technical Report TR04-11, University of Alberta.

Bowling, M. and M. Veloso: 2001, 'Rational and convergent learning in stochastic games'. In: *IJCAI 17*.

Bowling, M. H. and M. M. Veloso: 2002, 'Multiagent learning using a variable learning rate'. *Artificial Intelligence* **136**(2), 215–250.

Brown, G.: 1951, 'Iterative solution of games by ficticious play'. In: *Activity Analysis of Production and Allocation*. New York.

Claus, C. and C. Boutilier: 1997, 'The dynamics of reinforcement learning in cooperative multiagent systems'. In: *AAAI 4*. pp. 746 – 752.

Conitzer, V. and T. Sandholm: 2007, 'AWESOME: A general multiagent learning algorithm that converges in self-play and learns a best response against stationary opponents'. *Machine Learning* **67**(1), 23–43.

Fudenberg, D. and D. M. Kreps: 1993, ‘Learning Mixed Equilibria’. *Games and Economic Behavior* **5**(3), 320–367.

Govindan, S. and R. Wilson: 2003, ‘A global newton method to compute Nash equilibria’. *Journal of Economic Theory* **110**(1), 65 – 86.

Greenwald, A. and K. Hall: 2003, ‘Correlated-Q learning’. In: *ICML 20*.

Heidelberger, P. and P. Lewis: 1984, ‘Quantile estimation in dependent sequences’. *Operations Research* **32**(1), 185–209.

Hu, J. and M. Wellman: 1998, ‘Multiagent reinforcement learning: theoretical framework and an algorithm’. In: *ICML 15*. pp. 242 – 250.

Hu, J. and M. P. Wellman: 2003, ‘Nash Q-learning for general-sum stochastic games’. *Journal of Machine Learning Research* **4**, 1039–1069.

Lemke, C. and J. Howson: 1964, ‘Equilibrium points of bimatrix games.’. In: *Journal of the Society for Industrial and Applied Mathematics*, Vol. 12. p. 413423.

Lipson, A.: 2005, ‘An empirical evaluation of multiagent learning algorithms’. Master’s thesis, University of British Columbia, Vancouver, Canada.

Littman, M.: 1994, ‘Markov games as a framework for multi-agent reinforcement learning’. In: *ICML 11*. pp. 157 – 163.

Littman, M.: 2001, ‘Friend-or-foe Q-learning in general-sum games’. In: *ICML 18*. pp. 322 – 328.

McKelvey, R., A. McLennan, and T. Turocy: 2004, ‘Gambit: software tools for game theory’. Version 0.97.0.6. `http://econweb.tamu.edu/gambit`.

Monderer, D. and A. Sela: 1996, ‘A 2× 2 game without the fictitious play property’. *Games and Economic Behavior* **14**, 144–148.

Monderer, D. and L. Shapley: 1996, ‘Fictitious play property for games with identical interests’. *Journal of Economic Theory* **68**(1), 258–265.

Nudelman, E., J. Wortman, K. Leyton-Brown, and Y. Shoham: 2004, ‘Run the GAMUT: a comprehensive approach to evaluating game-theoretic algorithms’. In: *AAMAS 3*.

Osborne, M. and A. Rubinstein: 1994, *A Course in Game Theory*. MIT Press.

Powers, R. and Y. Shoham: 2005, ‘New criteria and a new algorithm for learning in multi-agent systems’. In: *NIPS*, Vol. 17. pp. 1089–1096.

R Development Core Team: 2006, ‘R: a language and environment for statistical computing’. R Foundation for Statistical Computing, Vienna, Austria.

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

Sandholm, T.: 2007, ‘Perspectives on multiagent learning’. *Artificial Intelligence* **171**(7), 382–391.

Shoham, Y. and K. Leyton-Brown: 2008, *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. New York: Cambridge University Press.

Shoham, Y., R. Powers, and T. Grenager: 2007, ‘If multi-agent learning is the answer, what is the question?’. *Artificial Intelligence* **171**(7), 365–377.

Singh, S., M. Kearns, and Y. Mansour: 2000, ‘Nash convergence of gradient dynamics in general-sum games’. In: *UAI 16*.

Spall, J. C.: 2003, *Introduction to Stochastic Search and Optimization: Estimation, Simulation and Control*. Hoboken, New Jersey: John Wiley & Sons.

Sutton, R. and A. Barto: 1999, *Reinforcement Learning, An Introduction*. Cambridge, Massachusetts: The MIT Press.

Tesauro, G.: 2004, ‘Extending Q-learning to general adaptive multi-agent systems’. In: *NIPS 16*.

Vu, T., R. Powers, and Y. Shoham: 2005, ‘Learning against multiple opponents’. In: *AAMAS*.

Watkins, C. and P. Dayan: 1992, ‘Q-learning: technical note’. *Machine Learning* **8**, 279–292.

Zinkevich, M.: 2003, ‘Online convex programming and generalized infinitesimal gradient ascent’. In: *ICML’03*.