Empirically Evaluating Multiagent Reinforcement Learning Algorithms in Repeated Games

by

Asher G. Lipson

BScHons University of the Witwatersrand, 2001

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Master of Science
in
THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University of British Columbia
September 2005
© Asher G. Lipson, 2005
Abstract

This dissertation presents a platform for running experiments on multiagent reinforcement learning algorithms and an empirical evaluation that was conducted on the platform. The setting under consideration is game theoretic in which a single normal form game is repeatedly played.

There has been a large body of work focusing on introducing new algorithms to achieve certain goals such as guaranteeing values in a game, converging to a Nash equilibrium or minimizing total regret. Currently, we have an understanding of how some of these algorithms work in limited settings, but lack a broader understanding of which algorithms perform well against each other and how they perform on a larger variety of games.

We describe our development of a platform that allows large scale tests to be run, where multiple algorithms are played against one another on a variety of games. The platform has a set of built-in metrics that can be used to measure the performance of an algorithm, including convergence to a Nash equilibrium, regret, reward and number of wins. Visualising the results of the test can be automatically achieved through the platform, with all interaction taking place through graphical user interfaces.

We also present the results of an empirical test that to our knowledge includes the largest combination of game instances and algorithms used in the multiagent learning literature. To demonstrate the usefulness of the platform, we provide evidence for a number of claims and hypotheses. This includes claims related to convergence to a Nash equilibrium, reward, regret and best response metrics and claims dealing with estimating an opponent’s strategy. Some of our claims include that (1) no algorithm does best across all metrics and over all opponents, (2) algorithms do not often converge to an exact Nash equilibrium, but (3) do often reach a small window around a Nash equilibrium, (4) there is no apparent link between converging to a Nash equilibrium and obtaining high reward and (5) there is no linear trend between reward and the size of the game for any agent.

The two major contributions of this work are a software platform for running large experimental tests and empirical results that provide insight into the performance of various algorithms.
Contents

Abstract ii
Contents iii
List of Tables v
List of Figures vi
Acknowledgements viii

1 Introduction 1

2 Related Work 3
  2.1 Single Agent Reinforcement Learning Algorithms 3
    2.1.1 Q-learning 4
  2.2 Game Theory 4
  2.3 Multiagent Learning Algorithms 6
    2.3.1 Game Theoretic Algorithms 7
    2.3.2 Extending Q-learning to Repeated Games 8
    2.3.3 Gradient-Based Algorithms 9
    2.3.4 Optimization Algorithms 13
    2.3.5 Estimating an Opponent’s Strategy 14
  2.4 Existing Testing Methodology 16
    2.4.1 Software Packages 16
    2.4.2 Experimental Methods 17
    2.4.3 Metrics 18
  2.5 Conclusion 20

3 A Platform for Multiagent Learning 21
  3.1 Aims and Objectives 21
    3.1.1 Setting 22
  3.2 The Platform Architecture 22
    3.2.1 Algorithms/Agents 23
3.2.2 Games
3.2.3 Performance Metrics
3.3 Setting Up and Running an Experiment
3.4 Visualization
  3.4.1 Step 1: Select the Agents, Runs and Iterations
  3.4.2 Step 2: Select the Generators
  3.4.3 Step 3: Select the Free Variables
  3.4.4 Step 4: Select the Metric
  3.4.5 Step 5: Select the Visualization Routine
  3.4.6 Step 6: Select the Aggregation Method
3.5 Limitations of the Platform
3.6 Conclusion

4 Empirical Testing
  4.1 Experimental Setup
  4.2 Observations from the Empirical Study
  4.3 Reducing the Size of the Experiment Space
    4.3.1 Collapsing the Iterations Dimension
    4.3.2 Collapsing the Instances Dimension
    4.3.3 Collapsing the Generators Dimension
    4.3.4 Collapsing the Agents Dimension
  4.4 Experimental Results: Reward-based Metrics
    4.4.1 Raw Reward
    4.4.2 Regret
    4.4.3 Incentive to Deviate
    4.4.4 Minimax-Q and Domination
  4.5 Experimental Results: Convergence to a Nash Equilibrium
    4.5.1 Linking Reward and Convergence
    4.5.2 Exact convergence to a Nash equilibrium
    4.5.3 Convergence in self play
    4.5.4 Convergence in Cooperative Games
  4.6 Experimental Results: Estimating an Opponent's Strategy
  4.7 Directions for Future Study

5 Conclusion
# List of Tables

2.1  Testing methodologies amongst different research papers .......................... 17  
3.1  Summary of implemented metrics ............................................................. 27  
4.1  Experiment: Algorithm parameters ........................................................... 38  
4.2  Experiment: Game generators with ten actions ........................................... 38  
4.3  Experiment: Metrics recorded ................................................................. 38  
4.4  Linear regression test for trend in reward and game size ........................... 40  
4.5  Kolmogorov-Smirnov test on 2x2 reward ................................................ 50  
4.6  Kolmogorov-Smirnov test on regret ......................................................... 52  
4.7  Kolmogorov-Smirnov test on minimax-Q and minimax-Q-IDR reward distributions 56  
4.8  Kolmogorov-Smirnov test on reward and Nash convergence for an agent ...... 61  
4.9  Kolmogorov-Smirnov test on the different estimation techniques ................ 68  
4.10 Kolmogorov-Smirnov test on the reward received by the estimation techniques . 69
# List of Figures

2.1 Two-player normal form game of Prisoner's Dilemma .................................................. 4  
2.2 Fictitious play pseudo code ......................................................................................... 7  
2.3 Determined agent pseudo code ................................................................................... 8  
2.4 Pareto agent pseudo code ........................................................................................... 8  
2.5 Infinitesimal Gradient Ascent pseudo code ................................................................. 10  
2.6 GIGA-WoLF pseudo code ............................................................................................ 11  
2.7 Projecting invalid strategies to the valid line ............................................................... 11  
2.8 Pseudo code for the retraction algorithm ...................................................................... 12  
2.9 Simulated Annealing pseudo code ................................................................................ 13  
2.10 The count-based approach to estimating an opponent's strategy .................................. 15  
2.11 Estimating your opponent's strategy based on past play ............................................ 15  
3.1 Format of a data object ................................................................................................ 23  
3.2 Class hierarchy of the algorithms ................................................................................ 24  
3.3 Game setup and error windows .................................................................................... 25  
3.4 Pipeline for setting up experiments in the platform ....................................................... 28  
3.5 The graphical user interface for phase one of an experiment ....................................... 29  
3.6 The visualization pipeline ............................................................................................. 30  
3.7 The graphical user interface for controlling the visualization pipeline ......................... 30  
3.8 GUIs for selecting the agents, runs and iterations in the visualization process .......... 31  
3.9 GUI for step 2 of the visualization process, selecting the generators ......................... 32  
3.10 GUI for step 3 of the visualization process, selecting the free variables .................... 33  
3.11 GUI for step 4 of the visualization process, selecting the dependent variable/metric .... 34  
3.12 GUI for step 5 of the visualization process, selecting the type of graph ..................... 34  
3.13 GUI for step 6 of the visualization process, selecting the method of aggregation ......... 35  
4.1 Reward vs game size ..................................................................................................... 40  
4.2 Reward over a single instance of Majority voting ....................................................... 43  
4.3 Reward and $\ell_1$ Nash distance for the Arms race generator .................................... 44  
4.4 Average reward on a subset of instances on the Majority voting generator ............... 45  
4.5 Convergence to a Nash equilibrium in supermodular games ..................................... 47  
4.6 Reward distribution in supermodular games ................................................................ 48
4.7 Average reward obtained by each agent against each other agent in the set of 2x2 (top) and 10x10 (bottom) generators.

4.8 Distribution of reward obtained in the 2x2 and 10x10 sets of generators.

4.9 Distribution of reward obtained in both generators.

4.10 Distribution of regret obtained by each agent.

4.11 Average incentive to deviate value in the 2x2 games.

4.12 Average incentive to deviate value in the 10x10 games.

4.13 Reward distributions received by minimax-Q and minimax-Q-IDR.

4.14 Percentage of wins in a 2x2 Traveller’s Dilemma game.

4.15 Percentage of wins in a 10x10 Traveller’s Dilemma game.

4.16 Reward and Nash convergence in the 2x2 set of games.

4.17 Reward and Nash convergence in the 10x10 set of generators.

4.18 Reward and Nash convergence in the Arms race generator.

4.19 Similar Nash convergence rates imply similar reward distributions for an agent.

4.20 Percentage of algorithms converging to a Nash equilibrium.

4.21 Percentage of algorithms converging near to a Nash equilibrium.

4.22 Convergence to a Nash equilibrium in self and non-self play.

4.23 Convergence to an epsilon window around the equilibrium in self and non-self play.

4.24 $\ell_1$ distance to a Nash equilibrium in the Minimum effort generator.

4.25 Distributions of the $\ell_1$ distance between actual and estimated strategy.

4.26 Reward distribution obtained by the different estimation techniques.
Acknowledgements

First to my parents who have supported me throughout my Canadian sojourn. They have managed to deal with me from afar and helped me with what ever I needed.

To all my labmates in LCI who have more or less kept me sane, thank you. I have had many conversations on topics ranging from research to the utterly bizarre with Eric Brochu, Andrea Bunt, Jacek Kisynski, Dustin Lang, Kasia Muldner and Robert St-Aubin.

My supervisors Kevin Leyton-Brown and Nando de Freitas have helped me throughout this long process. The research and work would not have happened without their guidance. I also thank Kevin Murphy who read different versions of this thesis and provided many useful comments.

To the various Witsies around the world, from Amherst to Edinburgh to Houston to Johannesburg to London to Los Angeles. Thanks for being there at any time of the day, no matter the time difference. We are really spreading across the globe.

Lastly, to Sarah who has helped me during this time more than she will ever know.

Asher G. Lipson

The University of British Columbia
September 2005
Chapter 1

Introduction

This dissertation describes a platform for running experiments with algorithms for multiagent learning. We used this platform in an empirical evaluation of multiagent algorithms in repeated games.

Recently, there has been much interest in the design and analysis of algorithms for game theoretic settings, for example by Singh et al. [2000], Bowling and Veloso [2001a], Tesauro [2003], Bowling [2004] and Powers and Shoham [2004]. Such work has typically introduced new algorithms, comparing them against one or two existing algorithms on a small set of well-known games. Algorithm performance is then evaluated using one or two metrics. This focus has led to a wealth of different algorithms for multiagent learning in repeated games, but a relative lack of general understanding of these algorithms' strengths and weaknesses.

The two standard methods for analyzing algorithms are theoretical and empirical analyses. These two methods are complementary, with a large body of work providing theoretical guarantees about algorithms’ performance in self play and infinite play settings, followed by a small set of empirical tests.

There are two limitations with theoretical analysis. Firstly, algorithms are not always in self play situations: they also play against other algorithms. The second limitation is the infinite play setting, since in practice algorithms are run in a finite play setting. There is not necessarily a clear mapping from infinite play guarantees to finite play guarantees. Thus, while theoretical guarantees provide important insight into the performance of an algorithm, they are not always practical. Further, the guarantees that can be proven are often weaker than what can be shown empirically.

In contrast to theoretical testing, empirical tests are designed to demonstrate how different algorithms perform against each other in different scenarios. In the multiagent setting, such a test entails different algorithms playing a variety of games against each other. The resulting data is then analyzed in a number of different ways. However, empirical tests are often performed with a small number of algorithms on one or two games. Since an exhaustive test of all algorithms and games is not possible, a subset of algorithms and games needs to be selected. Decisions also need to be made on methods for judging performance. Once these decisions are made, all of the algorithms need to be implemented and a method needs to be found for running combinations of algorithms and storing the results. The final step in an empirical test is to analyze all of the results, but given the large number of different opponents and settings, this is not an easy task.
The purpose of this research is to design a platform to test multiagent learning algorithms in a repeated game setting and to analyze the results of such testing.

The research described in this dissertation makes two contributions. First, we describe the design and implementation of a platform for running large experiments on multiagent reinforcement learning algorithms. A standardized platform offers several advantages over one-off solutions for running experiments. Second, we present an analysis of an empirical test that was conducted using our platform. We make a series of claims and argue for them on the basis of our experimental evidence. For example, we show how algorithm performance differs depending on the game and opponent. Furthermore, we identify algorithms which perform well according to different measures of performance such as the amount of reward attained by the agent and whether the agent converges to playing a Nash equilibrium strategy. These contributions are different from previous work where the primary motivation is usually to introduce a new algorithm.

The remainder of this dissertation describes the work involved with these contributions. In Chapter 2, background work is presented. In particular, there is a brief introduction to game theory and descriptions are given of existing software packages in Artificial Intelligence research. Chapter 3 describes the architecture of the testing platform. This includes information on the various algorithms, games and metrics that have been implemented. This chapter also describes the visualization process used in the platform and presents the various graphical user interfaces used for interaction. Chapter 4 describes the empirical test that was performed using our testing platform. The focus of this chapter is the analysis of the experimental results, which are presented as a list of claims. Lastly, Chapter 5 summarizes this dissertation.
Chapter 2

Related Work

This chapter describes the background and related work for this project. The background is split into four sections: single agent reinforcement learning algorithms, game theory, multiagent learning algorithms and existing testing methodologies.

2.1 Single Agent Reinforcement Learning Algorithms

In a single agent domain, an agent interacts with a static but stochastic environment. With reinforcement learning algorithms, an agent receives information about the environment, takes an action and then obtains feedback in the form of a payoff or reward [Sutton and Barto, 1999]. An agent's reward is dependent only on her own actions. The aim is to find the optimal actions, that is the actions that receive the highest reward. If there are multiple states in the environment, then the goal is to find the action which receives the highest reward in each state. The environment is stationary, meaning that the probability of an agent receiving a specific reward in a state does not change over time. Thus, once the optimal action is found, this action stays optimal for that state. However, it is difficult to find the optimal actions because the agent does not know the probabilities of transferring between different states based on her actions.

Every step, the agent has two choices: explore the environment or exploit current knowledge about the environment. There is a constant tradeoff between these two because they affect the agent's ability to find the optimal actions. The descriptions, advantages and disadvantages are given below:

**Exploitation** Assume that the current best action is optimal and play it, ignoring other possible actions. The disadvantage is that the agent may be ignoring the true optimal action. The advantage is that the action being played may be the true optimal action.

**Exploration** Play the other actions and see if any of them obtain higher expected rewards. The disadvantage is that the agent may already have found the optimal action and will now obtain lower reward. The advantage is that the exploration may find the optimal action.

If the agent receives reward $r_t$ at time $t$, then her total reward over some finite time period $T$ is $\sum_{t=0}^{T} r_t$ and her average reward is $\frac{1}{T} \sum_{t=0}^{T} r_t$. Rewards can also be discounted; a given reward
may be worth more to the agent today than the same reward tomorrow. The discounted reward is
\[ \sum_{t=0}^{T} \gamma^t r_t, \]
with \( \gamma \) being a discount factor and \( 0 < \gamma \leq 1 \). For further discussion, see Kaelbling et al. [1996].

### 2.1.1 Q-learning

Q-learning [Watkins and Dayan, 1992] is a model-free method, in which the agent does not know
what rewards are assigned to actions in the different states. The probability of transitioning from
state \( s \) to state \( s' \) is also unknown, as this is a property of the environment. Q-learning uses the
rewards obtained from taking actions to learn a policy of how actions perform in different states.

At each iteration/time step, the agent does an update based on the state \( s \) she is in, the action
\( a \) taken and the obtained reward \( r(s, a) \). Let \( Q(s, a) \) be the discounted value of taking action \( a \) in
state \( s \). The update at each step is then:

\[
Q(s, a) = (1 - \alpha)Q(s, a) + \alpha[r(s, a) + \gamma \max_{a'} Q(s', a')]
\]

The value of state \( s \) is \( V(s) = \max_a Q(s, a) \) and the action policy is \( \pi(s) = \arg \max_a Q(s, a) \).
\( \gamma \) is the discount factor and \( \alpha \) is the learning rate, which is traditionally decayed over time. It is
known that if \( \alpha \) is decayed and each action is played in each state an infinite number of times, then
\( Q(s, a) \) will converge to the optimal \( Q^*(s, a) \) [Kaelbling et al., 1996].

To counter the exploitation/exporation trade off, Q-learning agents typically assign some prob­
ability of taking an exploratory or random step at each iteration. This probability can be decreased
over time, making it more likely that the agent will select her current optimal action from \( \pi(s) \).

### 2.2 Game Theory

Game theory is a branch of Mathematics for analyzing strategic interactions amongst multiple play­
ers. The concepts behind game theory were initially introduced by von Neumann and Morgenstern
[1944] and gained prominence with the work of John Nash [1950]. This dissertation deals with
normal form games, where the payoffs are represented as matrices. Each dimension of the matrix
corresponds to the actions of a particular player. In the two-player case, the first agent’s action se­
lects a row and the second agent’s action selects a column. An example of a normal form game, the
well-known Prisoner’s Dilemma, is shown in Figure 2.1. The values in each entry correspond to the
first and second players’ reward.

\[
\begin{array}{c|cc}
\text{Player 1} & C_1 & D_1 \\
\hline
C_2 & 1,1 & -1,4 \\
D_2 & 4,-1 & 0,0 \\
\end{array}
\]

Figure 2.1: Example of a two-player normal form game. Player 1 selects amongst actions \( C_1 \) and
\( D_1 \) and player 2 selects from actions \( C_2 \) and \( D_2 \).
Formally, to define a normal form game, we follow the conventions of Shoham and Leyton-Brown [To appear]. Each game $G$ is a tuple:

$$G = (n, A, u)$$

where $A = (A_1, \ldots, A_n)$, with $n$ being the number of players and each player $i$ having a set of actions $A_i$. For any game, we can give an action profile $a \in A_1 \times \ldots \times A_n$, with $a = (a_1, \ldots, a_n)$ representing the action profile for all players and $a_{-i} = (a_1, a_2, \ldots, a_{i-1}, a_{i+1}, \ldots, a_{n-1}, a_n)$ representing the action profile for all players except player $i$. The set of rewards for all players is $u = (u_1, \ldots, u_n)$. Player $i$’s reward $u_i : A \rightarrow \mathbb{R}$ is determined by the actions of all agents and corresponds to a cell in the payoff matrix. The game theory community calls this value the payoff or utility, whereas the reinforcement learning community uses the term reward, but in both cases the meaning is the same.

If a player plays a single action $a_i$ deterministically, then she is playing a pure strategy. If however, the player chooses from a set of actions according to a probability distribution $\sigma_i$, then this is playing a mixed strategy. $\sigma_i$ is a probability distribution over player $i$’s actions. Similar to the pure action profile, we define a mixed strategy profile as $\sigma = (\sigma_1, \ldots, \sigma_n)$ and the mixed strategy profile excluding player $i$ as $\sigma_{-i} = (\sigma_1, \ldots, \sigma_{i-1}, \sigma_{i+1}, \ldots, \sigma_n)$. $\sigma_i(a_i)$ is the probability of player $i$ playing her action $a_i$ from the profile $a$ under the mixed strategy $\sigma_i$. We use $S_i$ to represent the support of player $i$’s mixed strategy, which is those actions with a positive probability of being played under the current strategy. A pure strategy is a mixed strategy having all probability mass on a single action and thus a singleton support.

The expected utility or payoff of a game for player $i$ is the expected reward given the mixed strategies of all players, and is defined in Equation (2.1). The reward for a pure action profile is multiplied by the probability of that action profile and all action profiles are summed over. The probability of an action profile is the probability of each player playing the corresponding action, based on their current mixed strategy.

$$E[u_i(\sigma)] = \sum_{a \in A} u_i(a)p(a|\sigma), \text{ with } p(a|\sigma) = \prod_{i=1}^{n} \sigma_i(a_i)$$ \hspace{1cm} (2.1)

A best response strategy for player $i$ against an action profile $\sigma_{-i}$ is:

$$br_i(\sigma_{-i}) = \arg \max_{\sigma_i} E[u_i(\sigma_{-i}, \sigma_i)]$$ \hspace{1cm} (2.2)

A Nash equilibrium has every player playing a best response to her opponents’ strategies, such that $\forall i \in n, \sigma_i = br_i(\sigma_{-i})$. Nash [1950] proved that every finite normal form game has a Nash equilibrium.

Three other concepts that are relevant to this work are dominated strategies, Pareto optimality and the security level of a game. Dominated strategies are strategies which will obtain a lower or equivalent reward than some other strategy, regardless of what an opponent plays. Strict dominance occurs for a strategy $\bar{\sigma}_i$ over all $\sigma_i$, if $\forall \sigma_{-i}, u_i(\sigma_{-i}, \bar{\sigma}_i) > u_i(\sigma_{-i}, \sigma_i)$. The iterated removal
dominated strategies is an iterative process, removing dominated strategies first from player $i$ and then removing dominated strategies for player $i + 1$ from the resulting reduced payoff matrix. This process is repeated until no player has any dominated strategies. If each agent is left with a single undominated action after iteratively removing dominated strategies, then the resulting profile will always be a pure strategy Nash equilibrium.

A strategy $\sigma$ is Pareto optimal if there is no other strategy profile in which all players do at least as well and at least one player does better. Formally, a strategy profile $\sigma$ is Pareto optimal, if for all players $i \in n$ there is no strategy profile $\sigma'$, such that $u_i(\sigma) \leq u_i(\sigma')$ and for at least one player $j$, $u_j(\sigma) < u_j(\sigma')$.

The security level of a game is the reward that an agent can guarantee herself in the game. This is also referred to as the maxmin value of a game. The security level of a game for player $i$ is $\max_{\sigma_i} \min_{\sigma_{-i}} u_i(\sigma_{-i}, \sigma_i)$. This assumes that the opponent will play the strategy that will hurt the player the most, with the player attempting to maximise the output of this minimisation process.

All of the descriptions given so far are for a single interaction amongst agents in a normal form game. In that case, no learning or modelling of the opponent is necessary. If playing the same player multiple times, then learning from past play may allow an agent to obtain better results in the future. In repeated games, the same players repeatedly play against each other on the same game. The game that is being repeatedly played is called a stage game.

### 2.3 Multiagent Learning Algorithms

We have now described the use of reinforcement learning in a single agent environment (Section 2.1). We have extended the environment to a game theoretic multiagent setting with normal form games, which can be repeatedly played in repeated games. We now discuss algorithms for playing on repeated games.

Moving from the single agent to multiagent domain affects many of the properties discussed in Section 2.1. In a single agent domain, there is a clear definition of an optimal action and the environment is stationary. In a multiagent domain, an agent is still able to obtain information about her environment, play an action and then receive a reward. However, an agent's reward is now dependent on both her own action and those of all other agents in the environment. There is no longer a clear definition of an optimal action. The environment is no longer stationary because opponents have an effect on the environment. An action may appear optimal at some stage game, but if the other agent changes her strategy, then the action may not be optimal in the future. Without taking into account an opponent's actions, it is impossible to state that a specific action is optimal.

In the repeated game setting, an agent can attempt to estimate her opponent's strategy and then learn what action to play. This learning is based on the actions that were played in the past and the reward that the agent obtained. Through this method, an agent tries to find optimal strategies in response to what an opponent is playing. The repeated game setting can also involve teaching because an algorithm can attempt to teach an opponent to play to a specific outcome, for example a particular Nash equilibrium that the teacher favours. This aspect of "teaching" is normally implicit in a learning algorithm with an agent selecting an action based on the past play of the opponent.
The action that was selected is influenced by the opponent's past play [Shoham et al., 2004].

Two possible methods for dealing with an opponent are to ignore the problem and not be concerned with the existence of other agents, or to make an attempt to estimate what strategy the opponents are playing. The first solution is akin to playing a single agent learning algorithm, for example Q-learning (Section 2.1.1). The second approach of opponent tracking is discussed in Section 2.3.5.

2.3.1 Game Theoretic Algorithms

2.3.1.1 Fictitious Play

Fictitious play is the earliest example of a learning algorithm for repeated games Brown [1951]. The agent observes the actions of her opponent and estimates the opponent’s (mixed) strategy by counting the number of times the opponent has played each of her actions so far. The agent then chooses to play the action (i.e., the pure strategy) which maximizes her expected payoff given her estimate of the opponent’s strategy.

Each iteration, the fictitious play agent determines the action that returns the highest expected payoff based on the estimated strategy of the opponent. The cost of this step is thus linear in the number of actions, multiplied by the cost of calculating the expected payoff.

At time $t + 1$, the player selects a best response pure strategy $\tilde{a}_{t+1}$ to the estimate of the opponent’s strategy $\tilde{C}_t$:

$\tilde{a}_{t+1} = \text{br}_i(\tilde{C}_t) = \arg\max_{a_i} E[u_i(\tilde{C}_t, a_i)]$

Player then updates her estimate of the opponent’s strategy based on the opponent’s action $o_{t+1}$:

$C_{t+1} = C_t + e_{o_{t+1}}; \tilde{C}_{t+1} = \text{normalize}(C_{t+1})$

Figure 2.2: The fictitious play algorithm for a two player game.

2.3.1.2 Determined Agent

Determined agent is an algorithm that calculates all of the Nash equilibria in a game and then stubbornly plays the strategy that corresponds to the equilibrium with the highest payoff for herself. This is a selfish agent that does not take into account the opponent. The assumption is that the opponent is a learning algorithm that will play a best response to the determined agent’s strategy. Given that the determined agent is playing her portion of a Nash equilibrium strategy, a learning opponent should always converge to her corresponding strategy, which by definition will be a best response. The algorithm is given in Figure 2.3. In order to solve for the Nash equilibria of a game, the algorithm requires knowledge of the opponent’s payoff matrix.

The determined agent algorithm is similar to Bully [Littman and Stone, 2001], which plays an action which the opponent plays a best response to that leads to the highest payoff for Bully. In this way, Bully looks at the opponent playing a best response, but it does not necessarily play a best response to the opponent.
2.3. Multiagent Learning Algorithms

Determine which equilibrium strategy from the set of equilibria \( \{Eqm\} \) leads to the highest expected payoff for player \( i \):

\[
\sigma = \arg \max_{\sigma \in \{Eqm\}} u_i(\sigma)
\]

Player \( i \) then plays her corresponding strategy \( \sigma_i \)

Figure 2.3: The determined agent algorithm for a two player game.

2.3.1.3 Pareto Agent

The Pareto agent algorithm finds all Pareto optimal outcomes in a game and plays the action that corresponds to the Pareto optimal outcome with the highest payoff to herself. The algorithm assumes that the opponent will respond by playing to the same Pareto outcome. However, this assumption is not always valid as a best response to the agent’s action may not lead to the Pareto outcome. This is due to the definition of Pareto optimality, as the opponent may achieve a higher reward and the Pareto agent a lower reward if a different outcome is played. The algorithm is given in Figure 2.4. This algorithm requires knowledge of both payoff matrices in order to solve for the Pareto optimal outcomes.

\[
\bar{a} = \arg \max_{\bar{a} \in \{Pareto\}} u_i(\bar{a})
\]

Player \( i \) then plays her corresponding action \( \bar{a}_i \)

Figure 2.4: The Pareto agent algorithm.

2.3.2 Extending Q-learning to Repeated Games

2.3.2.1 Minimax-Q

One of the most popular approaches in single agent environments is Q-learning (Section 2.1.1). If an opponent’s strategy converges, then there is the guarantee that the Q-learning algorithm will also converge. There has also been research to extend the method to the multiagent environment. The basis for these extensions is to take into account the opponent’s actions and store discounted “Q-values” for the joint actions of the agent and her opponent.

Minimax-Q follows a conservative strategy by assuming the opponent will try to prevent the agent from obtaining a high reward. The algorithm does not take into account how the opponent is playing and whether she could receive a higher reward by playing a different action.

Minimax-Q [Littman, 1994] is particularly prominent in the literature. Instead of computing a Q-value \( Q(s,a) \) over the state and action (see Section 2.1.1), the Q-value is over state, action and opponent’s action \( Q(s,a,o) \), with \( o \) representing the opponent’s action at that iteration. Equation (2.3) gives the Q-learning update step and Equation (2.4) gives the minimax-Q update step. The difference is that the \( Q \) and \( r \) values are now referenced by the opponent’s action as well as the
agent's action. The max-operation in Q-learning is replaced by a maxmin operation in minimax-Q, obtaining the safety level of the game described in Section 2.2.

\[
Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r(s, a) + \gamma \max_a Q(s', a'))
\]

(2.3)

\[
Q(s, a, o) = (1 - \alpha)Q(s, a, o) + \alpha(r(s, a, o) + \gamma \max_{a' \in \mathcal{A}_i} \min_{a' \in \mathcal{A}_{-i}} Q(s, a, o))
\]

(2.4)

As in the Q-learning algorithm, minimax-Q allows the agent to explore. When an agent selects an action, she explores randomly with some probability, otherwise taking the action with highest Q-value.

2.3.3 Gradient-Based Algorithms

This section is split into two parts. First, it discusses two gradient-based algorithms and then describes some of the problems with these algorithms.

A gradient-based learning algorithm views the strategies of both agents in a multidimensional space with a surface representing the agent’s expected payoff. Each dimension of the space corresponds to one of the agents’ actions and ranges over the probability of taking that action [0, 1]. In a game where each agent has three actions, the space has 2 x 3 dimensions. A point in the space has a value equal to the expected payoff of an agent, for the strategies corresponding to that point.

The goal of gradient algorithms is to incrementally update an agent’s strategy in order to move in the direction of increasing expected payoff. However, for the agent to determine where on the surface she is, she requires knowledge of both her own and her opponent’s strategies. Since an agent only has direct control over her own strategy, a method is required for determining the opponent’s strategy.

2.3.3.1 Infinitesimal Gradient Ascent

Gradient-based algorithms were first explored in repeated games with the Infinitesimal Gradient Ascent (IGA) algorithm [Singh et al., 2000]. The initial work required knowledge of an opponent’s actual strategy, something that is not practical. As will be discussed in Section 2.3.5, this problem can be overcome by estimating the distribution based on observations of the opponent’s actions.

IGA uses a decaying step size \( \eta \) that represents the size of the step to take in the payoff space. IGA was initially presented for the two-player, two-action repeated game case. The strategies of the two players form a linear dynamical system and convergence proofs can be shown for the system. Extending the IGA approach to larger games introduces either more complex update equations or non-linearity in the system. Figure 2.5 gives pseudo code for the algorithm for the two-player, two-action case.

2.3.3.2 Win-or-Learn-Fast

The Win-or-Learn-Fast (WoLF) approach uses a variable step size that changes according to how well an agent is doing. This approach was used on the IGA algorithm to produce the WoLF-IGA
2.3. Multiagent Learning Algorithms

For the following payoff matrix:
\( r_{ij} \) is the payoff to the row agent, \( c_{ij} \) is the payoff to the column agent

\[
\begin{bmatrix}
  r_{11} & c_{11} \\
  r_{21} & c_{21}
\end{bmatrix}
\]

For the row player, the algorithm is written as:
\( \alpha \) is the probability of the row agent playing their first action
\( \beta^e \) is the estimated probability of the column agent playing their first action

We can write the expected payoff of the row player for the overall strategy \((\alpha, \beta^e)\) as:
\[
V_r(\alpha, \beta^e) = r_{11}(\alpha \beta^e) + r_{22}(1 - \alpha)(1 - \beta^e) + r_{12}(1 - \beta^e)\alpha + r_{21}(1 - \alpha)\beta^e
\]

Letting: \( u = (r_{11} + r_{22}) - (r_{21} + r_{12}) \)
The gradient is:
\[
\frac{\partial V_r(\alpha, \beta^e)}{\partial \alpha} = \beta^e u - (r_{22} - r_{12})
\]
The row player can then use the update rule, with step size \( \eta_r \):
\[
\alpha_{t+1} = \alpha_t + \eta_r \frac{\partial V_r(\alpha_t, \beta^e)}{\partial \alpha}
\]

Figure 2.5: *Pseudo code for the Infinitesimal Gradient Ascent algorithm in a two action game* (Singh et al., 2000)

algorithm [Bowling and Veloso, 2001a]. If an agent is achieving high payoffs, she uses a small step size, thereby slowing down her learning rate. The smaller step is an attempt by the agent to exploit her current strategy. If the agent is doing badly, then she uses a larger step size allowing the agent to adapt faster. This is an attempt by the agent to explore her action space, since her current strategy is not achieving a high reward. In WoLF-IGA, the agent judges how well she is playing based on the payoff she would have received under a Nash equilibrium. In games with multiple equilibria, which equilibrium is chosen as the basis for comparison will affect the algorithm’s performance, as different equilibria could have different payoffs.

An improved version of WoLF-IGA does not compare performance to a Nash equilibrium, but rather keeps track of the agent’s regret with respect to the counterfactual possibility of having played a stationary pure strategy Bowling [2004]. This version, GIGA-WoLF, offers the theoretical guarantee that long-run regret will not be positive. This algorithm does not directly utilize a variable step size, but rather the step size is affected by the magnitude of the gradient. Figure 2.6 gives the pseudo code for this algorithm.

2.3.3.3 Projection Operators

With gradient algorithms, it is possible for an agent’s strategy to leave the valid probability space. This occurs when an update step changes the probability of an action being taken and this process moves the entire strategy from the valid probability range. Probabilities are defined on the \([0, 1]\) range, but updates can move beyond these bounds. This invalid distribution could have one of the probabilities > 1 or < 0 or the distribution of actions may not sum to 1. Figure 2.7 presents an example for two actions, where the line represents the range of valid probability distributions.
2.3. Multiagent Learning Algorithms

Store two strategies, the actual strategy $\sigma_t$ and a strategy for comparison $z_t$, with the agent selecting an action from $\sigma_t$.

$\gamma_t$ is the step size and $r_t$ is a vector containing the rewards that the agent would have received, for each of the pure actions she could have played at $t$.

\[
\begin{align*}
\hat{\sigma}_{t+1} & = P(\sigma_t + \gamma_t r_t) \\
z_{t+1} & = P(\sigma_t + \frac{1}{3} \gamma_t r_t) \\
\delta_{t+1} & = \min(1, \frac{||z_{t+1} - \hat{\sigma}_{t+1}||}{||z_t - \hat{\sigma}_t||}) \\
\sigma_{t+1} & = \hat{\sigma}_{t+1} + \delta_{t+1} (z_{t+1} - \hat{\sigma}_{t+1})
\end{align*}
\]

where $P(X)$ maps $X$ back to the valid probability distribution, since taking a step in the payoff space may lead outside the valid probability region.

Figure 2.6: Pseudo code for the GIGA-WoLF algorithm [Bowling, 2004]. $z_{t+1}$ is a strategy obtained from $\sigma_t$ with a smaller step $\frac{1}{3} \gamma_t$ than the step $\gamma_t$ used to obtain $\hat{\sigma}_{t+1}$.

Points off of the line are invalid and need to be projected to a valid distribution on the line.

Figure 2.7: Projecting invalid strategy $\sigma$ to the valid line. The invalid distributions $\sigma$ do not satisfy the constraints that the elements sum to one and each individual element is in the range $[0, 1]$. The point on the line is the closest to $\sigma$ in terms of the Euclidean distance.

The method used in this dissertation is based on the retraction operator described in Govindan and Wilson [2003]. The retraction mapping $r : \mathbb{R}^m \rightarrow \sum$, is from any real valued vector of $m$ actions to the set of valid probability distributions $\sum$. An invalid distribution needs to increase (reduce) its overall sum if there is a deficit (excess) of 'probability mass'. An iterative approach is taken, where each iteration, the deficit (excess) probability is calculated and then added (subtracted) uniformly to the actions having non-zero probability. The approach has cost that is worst case linear.
in the number of actions, which occurs when the probability of taking one action is set to zero at each iteration. One step of this process is shown in figure 2.8.

<table>
<thead>
<tr>
<th>$\sigma^{(k)} = (\sigma_{1:M}^{(k)})$</th>
<th>Current invalid strategy with $M$ elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>while $\left(\sum_{p=1}^{M} \sigma_p^{(k)} \neq 1\right)$ OR $\left(\left(\sigma_p^{(k)} &gt; 1\right) \text{ OR } \left(\sigma_p^{(k)} &lt; 0\right)\right)$</td>
<td>Condition on loop</td>
</tr>
<tr>
<td>$\sigma_{\neq 0}^{(k)} = (\sigma_{1:M}^{(k)} \neq 0)$</td>
<td>Find the non-zero elements of $\sigma^{(k)}$</td>
</tr>
<tr>
<td>$v = \frac{\sum_{p=M}^{(k)}(\sigma_p^{(k)}) - 1}{\sum_{p=0}^{\sigma_{\neq 0}}}$</td>
<td>Calculate the deficit or excess probability to be added or subtracted to each action with non-zero probability</td>
</tr>
<tr>
<td>$\sigma_{=0}^{(k)} = 0$</td>
<td>Set the elements $\sigma^{(k)} = 0$ to 0</td>
</tr>
<tr>
<td>$\sigma_{\neq 0}^{(k)} = \max(\sigma_{\neq 0}^{(k)} - v, 0)$</td>
<td>Set the elements of $\sigma^{(k)}$ which are non-zero</td>
</tr>
</tbody>
</table>

Figure 2.8: Pseudo code for the retraction algorithm, based on Govindan and Wilson [2003]. $\sigma^{(k)}$ is the strategy at time $k$. $M$ is the number of actions.

### 2.3.3.4 Gradient Approximation

Gradient algorithms depend on the gradient of the payoff space. In this space, the gradient is a vector with an element for each dimension of the space and thus a larger number of actions means more calculations to compute the gradient. To speed up this step, it is possible to approximate the gradient using difference schemes, for example finite difference (FD) approximations such as forward difference, backward difference or central difference.

However, the finite differences formulas need to be applied once per dimension in order to obtain the gradient. For example, to obtain the gradient in dimension $z$, the forward difference approximation is:

$$gradient_z = \lim_{c_k \to 0} \frac{f(\theta + c_k e_z) - f(\theta)}{c_k}$$

In order to use an analytical expression for the gradient, we firstly need to be able to obtain this expression and then be able to evaluate it every iteration. This requires symbolic computation, which in Matlab is extremely slow. For these reasons, it is not used.

Simultaneous Perturbation Stochastic Approximation (SPSA) [Spall, 2003] is a method similar to the finite difference techniques, but instead of requiring the formula to be applied once per dimension, it requires two calculations regardless of the number of dimensions. The SPSA scheme is used to calculate any gradients in the platform described in this dissertation.
2.3. Multiagent Learning Algorithms

2.3.4 Optimization Algorithms

Multiagent learning has the goal of maximising the reward that an agent receives. In this way, it can be viewed as an optimisation process to maximise a payoff function. The variables that can be changed are the strategy of an agent and the function to maximise is the utility of the agent. This section presents two optimization algorithms. The first is simulated annealing and the second is a stochastic approximation algorithm that uses aspects of simulated annealing.

2.3.4.1 Simulated Annealing

Simulated annealing is a method used to find the global optimum of a function and is based upon a model of cooling a liquid. The algorithm utilises random perturbations to avoid local maxima, attempting to reach the global maximum. After the algorithm has run for a large amount of time, there is some probability that it will select a value that is worse than its current state. However, the chances of this occurring decrease the longer the algorithm runs [Spall, 2003]. Using a simulated annealing algorithm in a multiagent environment requires knowledge of the opponent's strategy.

The simulated annealing algorithm is given in Figure 2.9. $\theta$ is the agents' strategies or probability distributions over actions. $L$ is the function to be maximised, which in multiagent terms would be the agent's utility. $L(\theta)$ is the expected value of $\theta$ in this function. Thus by changing the strategy $\theta$, the algorithm attempts to obtain a higher expected payoff. Since $\theta$ contains the strategies of both agents, the algorithm can only change her own portion of the strategy. $c_0$ is a normalising constant. $T$ is the "temperature" of the system, starting high and decreasing over time. This allows the system to accept a lower values state (lower expected payoff) at a higher temperature in early stages of the running of the algorithm.

| Step 0 | Set an initial temperature $T$ and vector $\theta_{\text{curr}} = \theta_0$. Determine $L(\theta_{\text{curr}})$. |
| Step 1 | Relative to $\theta_{\text{curr}}$, randomly select a new value $\theta_{\text{new}}$ by adding a random perturbation and then obtain $L(\theta_{\text{new}})$. |
| Step 2 | Let $\delta = L(\theta_{\text{new}}) - L(\theta_{\text{curr}})$. If $\delta > 0$, $\theta_{\text{curr}} = \theta_{\text{new}}$, else select a uniform random variable $U$ and if $U \leq \exp(\frac{\delta}{c_0 T})$, then $\theta_{\text{curr}} = \theta_{\text{new}}$. |
| Step 3 | Repeat steps 1 and 2 for some budgeted number of steps. |
| Step 4 | Lower $T$ according to an annealing schedule, for example $T = \frac{1}{\log(\text{iteration})}$. Repeat steps 1-4 for some number of iterations. |

Figure 2.9: The algorithm for Simulated Annealing, taken from Spall [2003, pg. 212].

2.3.4.2 Global Stochastic Approximation

Global Stochastic Approximation (GSA) Spall [2003] utilizes aspects of annealing and stochastic approximation. It is very similar to IGA, but adds a random perturbation term to the stochastic approximation update equation. This adds some "jumps" to the update equation in order to explore
2.3. Multiagent Learning Algorithms

the space and avoid local maxima. To our knowledge, the GSA algorithm has not previously been applied to a repeated game setting.

The update equation is

$$\sigma_{t+1} = \sigma_t + \beta_t G(\sigma_t) + \alpha_t w_t$$  \hspace{1cm} (2.5)

with $\sigma$ being an agent's strategy, $\beta_t$ the decreasing step size and $G(\sigma_t)$ the gradient of the function at $\sigma_t$. The random perturbation term $w_t$ is often a vector with each element drawn from a Gaussian distribution. $w_t$ is scaled by $\alpha_t$, which decreases faster than $\beta_t$. This is a policy of annealing by reducing the effect of $w$ over time. If $\alpha$ is decreased too quickly, then the algorithm will resemble the standard stochastic approximation. There is a large literature on selecting conditions and values for $\alpha$ and $\beta$, but this is beyond the scope of this discussion. Section 4.1 describes the parameter values used in this algorithm for the experimental testing.

2.3.5 Estimating an Opponent’s Strategy

The gradient-based algorithms require knowledge of the opponent’s strategy. Some of the early work assumed that an agent could directly view her opponent’s strategy [Singh et al., 2000; Bowling and Veloso, 2001a]. This approach is impractical, as an agent does not have this degree of knowledge about the other agent. However, an agent is able to view what actions her opponent plays. This history of past play can be used to estimate an opponent’s strategy.

Two approaches to estimating what strategy an opponent is playing are (1) a count-based or undiscounted stochastic approximation and (2) a discounted stochastic approximation. The undiscounted approach assumes the opponent is selecting her actions from a stationary distribution. If this assumption is correct, then the undiscounted estimate will converge to the actual strategy of the opponent [Fudenberg and Levine, 1999], but stationarity in games is a questionable assumption.

In the count-based approach, agents do not differentiate between actions based on the iteration in which they were played. An agent keeps track of a count over all of her opponent’s actions, incrementing the counter when an action is played. At time $t$, the count $C_{t+1}$ is updated with $C_{t+1} = C_t + e_{a_t}$. The vector $C$ has length $|C|$ equal to the number of available actions and $C_0$ is initialised to $[0 \ldots 0]$. $e_{a_t}$ is a zero-vector of length $|C|$, with a 1 at position $a_t$. An example of the count-based approach is given in Figure 2.10. A history over 15 iterations for an agent with five actions is shown, along with the resulting estimate for the strategy.

In the discounted stochastic approximation approach, the time when the action was played is taken into account. This is similar to the Q-learning update where more weight is put on an algorithm’s recent play. The update equation at time $t + 1$ is:

$$D_{t+1} = (1 - \beta_t)D_t + \beta_t e_{a_t}; \hspace{0.5cm} D_{t+1} = \text{normalize}(D_{t+1}); \hspace{0.5cm} \beta_{t+1} = \beta_t \beta_{\text{decay}}$$  \hspace{1cm} (2.6)

$D_t$ is the estimate of probabilities over actions and $a_t$ is the action that the opponent took at time $t$. $\beta_t$ is the belief update that has $\beta_1$ set $\leq 1$ and decayed by $\beta_{\text{decay}}$ at each iteration.
One problem with this approach is that it is sensitive to the parameters $\beta$ and $\beta_{\text{decay}}$. Figure 2.11 plots the probabilities of the actions given in Figure 2.10(a) according to different values of the parameters in the discounted update equation. The graph legend gives $\left(\beta, \beta_{\text{decay}}\right)$. The probabilities from the count-based approach are plotted for comparison.

Figure 2.10: Count-based approach to tracking an opponent’s strategy

Figure 2.11: Plots of the estimated strategy for an agent that plays the actions in Figure 2.10(a), over fifteen iterations. Four of these are stochastic approximations using different values of initial $\beta$ and $\beta_{\text{decay}}$, as given in the legend. The fifth plot is that of the count-based approximation. The legend is labelled as $\beta, \beta_{\text{decay}}$. 
2.4 Existing Testing Methodology

Using a well-described testing methodology can help in running experiments. Within the theoretical Computer Science community, there has been some work on developing best practices for the empirical evaluation of algorithms, for example by Johnson [2002]. Within the multiagent learning community, researchers use a number of different methodologies.

It is difficult and often impossible to compare algorithms from different papers that were tested differently. Using a standardized experimental methodology allows results to be more easily compared between different experiments. A standardized testing methodology also allows an experiment to be reproduced. Buckheit and Donoho [1995] emphasise the importance of reproducibility by stating that an "article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures".

We first describe some software platforms that have been previously developed. A discussion is given on two of the features involved in an experimental methodology: how to run the experiment and how to measure algorithm performance. The focus here is on how these features have been chosen in previous work.

2.4.1 Software Packages

Though there is no previously published research of which we are aware on experimental platforms for multiagent learning, there has been similar work in related areas. This section relates some of these software packages to the work described in this dissertation.

The issue of an experimental platform has been raised in the single agent reinforcement learning community. One of the discussions took place as a workshop at the 2004 Neural Information Processing conference. The "Reinforcement Learning Benchmarks and Bakeoffs" workshop [Sutton and Littman, 2004] discussed using a single platform with standardized problems to test reinforcement learning algorithms. The two implementations mentioned were the Closed Loop Simulation System (CLS$^2$) [Reidmiller et al., 2005] and RL Bench [Langford and Bagnell, 2005] a text-based system.

CLS$^2$ is arguably the closest package to the platform presented in this dissertation. It currently contains implementations of three agents (controllers) and six environments (plants) and allows a number of statistics to be recorded and plotted. Making multiple runs on the same controller is possible, but any averaging of those statistics needs to be done outside of the package. Running different controllers requires running multiple versions of CLS$^2$.

A second package from Artificial Intelligence is the Bayes Net Toolbox (BNT) [Murphy, 2001]. BNT was created to solve problems involving graphical models. The package is written in object-oriented Matlab and interaction is through Matlab's command line. BNT served as the inspiration for using Matlab's object-oriented language to develop the package described in this dissertation.

Two packages that are directly relevant to this research are GAMUT [Nudelman et al., 2004] and Gambit [McKelvey et al., 2004]. GAMUT is a suite of game generators for producing normal
form games. Each generator corresponds to a type of normal form game, for example Rock Paper Scissors and Prisoner's Dilemma have two different generators. An instance of a given game follows the rules or format for the game, but may differ in payoffs from other instances created by the same generator. GAMUT can be used to quickly produce a large number of games from different generators, providing a wealth of benchmark data for testing.

Gambit is a package for computing solution concepts of normal form and extensive form games. These concepts include Nash equilibria and dominated strategies. Gambit allows terminal-based interaction, where the program can be called to solve a game for a specific concept.

From these packages, the Bayes Net Toolbox provided some insight into software design in Matlab while GAMUT and Gambit cover part of the work involved in developing a platform for testing multiagent learning algorithms.

2.4.2 Experimental Methods

A wide variety of choices about experimental setup must be made before an experiment can be run. Unfortunately, papers in the literature vary widely in the way they make these choices. Furthermore, some papers do not even discuss the parameters used in their implementations of the algorithms, which can make it difficult to reproduce experiments. Table 2.1 shows how six recent research papers ran experiments, illustrating the fact that the researchers used very different experimental setups.

<table>
<thead>
<tr>
<th>Paper</th>
<th># iterations</th>
<th># games</th>
<th># runs/trials</th>
<th>Settling in/ recording period</th>
</tr>
</thead>
<tbody>
<tr>
<td>Littman [1994]</td>
<td>1.1M</td>
<td>1</td>
<td>3</td>
<td>1M/100k</td>
</tr>
<tr>
<td>- minimax-Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Claus and Boutilier [1997]</td>
<td>50 or 2500</td>
<td>3</td>
<td>100</td>
<td>0/50 or 2500</td>
</tr>
<tr>
<td>- Joint Action Learners</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bowling [2004]</td>
<td>1M</td>
<td>1</td>
<td>unknown</td>
<td>0/1M</td>
</tr>
<tr>
<td>- GIGA-WoLF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nudelman et al. [2004]</td>
<td>100k</td>
<td>13 GAMUT distributions</td>
<td>100 instances/game 10x per instance</td>
<td>90k/10k</td>
</tr>
<tr>
<td>- GAMUT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Powers and Shoham [2004]</td>
<td>200k</td>
<td>21 or 35 GAMUT distributions</td>
<td>unknown</td>
<td>180k/20k</td>
</tr>
<tr>
<td>- MetaStrategy</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tesauro [2003]</td>
<td>1.6M</td>
<td>1</td>
<td>unknown</td>
<td>0/1.6M</td>
</tr>
<tr>
<td>- Hyper-Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Testing methodologies in different research papers.

Overall, most of the tests performed in these papers (and in other papers from the literature) were quite small in terms of the number of algorithms considered. Tesauro [2003] and Bowling [2004] reported tests of their new algorithms against two and one algorithms respectively. Nudelman et al. [2004] used three agents but many games; however, the purpose of that paper was primarily the demonstration of the GAMUT software. The experiments by Littman [1994] were performed with four algorithms, two minimax-Q and two Q-learning algorithms. Claus and Boutilier [1997] utilized two algorithms. Powers and Shoham [2004] had one of the largest experiments but only used reward as a metric.
2.4. Existing Testing Methodology

Tests have also tended to be small in terms of the number of different games considered. For example, many papers (Bowling and Veloso [2001b], Tesauro [2003], Bowling [2004]) tested on a single game. Some used a single new game to demonstrate properties of an algorithm, e.g., Littman [1994], Claus and Boutilier [1997], Littman [2001] and Bowling and Veloso [2002]. To some extent, this reflects the difficulty of creating a large number of different games for use in tests. Recently, some papers including Nudelman et al. [2004] and Powers and Shoham [2004] have tested on larger sets of games, using the GAMUT software Nudelman et al. [2004] to produce test data.

Experiments also differed substantially in terms of the number of iterations (i.e., repetitions of the normal form game). First, note that the iterations in a repeated game are often split into "settling in" and "recording" phases, allowing the algorithms time to determine a strategy before results are recorded. This is done to reduce the sensitivity of experimental results to different algorithms' learning rates. Littman [1994] used a simple two-player soccer game that took a variable number of iterations to play once; nevertheless, tests ran for a fixed number of iterations rather than a fixed number of games. The experiments in Claus and Boutilier [1997] were run with different numbers of iterations (though far fewer than were used by any other researchers). In Powers and Shoham [2004], different results were shown in the paper for a 21 game distribution and an "all" game distribution from GAMUT, which currently includes 35 generators.

2.4.3 Metrics

We define a metric as a numerical measure of an algorithm's performance. Metrics are necessary for making quantitative comparisons between algorithms. This section is split into subsections based on the type of metric being discussed.

2.4.3.1 Single Agent Metrics

In the single agent setting, Kaelbling et al. [1996] present three measures of learning performance. Eventual convergence to optimal measures whether an agent finds the optimal strategy. However, an agent that reaches an almost optimal solution quickly may be preferable to an agent that achieves the optimal solution, but takes ages to do so. For this reason it is stated that this measure "is reassuring, but useless in practical terms" [Kaelbling et al., 1996, pg 241]. Speed of convergence to near-optimality tests how quickly the algorithm reaches a near-optimal solution. However, this requires a definition of near-optimal, which can be hard to give. There are also problems with an algorithm that tries to reach optimality as quickly as possible, but along the way incurs large penalties. Regret is a measure of how an algorithm played compared to playing optimally from some class of policies. The basis for all of these metrics is the definition of an optimal solution, which is lacking in a multiagent setting.

2.4.3.2 Reward-based Metrics

The most basic metric is reward maximisation, which is the goal of multiagent learning. However, using reward as an isolated metric can cause problems. If an agent obtains a reward of 0.5, is
this good or bad? The reward needs to be compared or judged against some value to determine how "good" it is. An agent's reward can be compared against her opponent's reward. This corresponds to the number of wins metric discussed in the next paragraph. An agent's reward can also be compared against the possible rewards that she could have obtained under a different set of strategies and this gives a measure of regret.

The number of wins metric judges a stage game win by which agent achieved a higher reward. This implicitly assumes that a game is zero-sum with one agent doing well to the detriment of the other agent. A payoff matrix may also have higher payoffs for one player in all outcomes. This problem can be partly solved by allowing the agents to play as both the row and column player and averaging over the results.

Regret asks the question "Could the agent have done better?" It also judges the agent's actions in response to the opponent's play, taking into account the multiagent aspect. Bowling [2004] judges regret as the difference between the reward obtained with the agent's current strategy and the reward that could have been obtained under the best pure strategy in the repeated game. This metric does not look at whether the opponent's strategy would have changed as a result of this pure strategy. Another form of regret measures an agent's incentive to deviate to any mixed strategy. This compares the algorithm's play to what would have been the best response action in each stage game. We can also ask what percentage of the reward from a best response action did the agent receive. An agent may have received a small reward, but this could be a high percentage of the reward that she would have received from a best response action.

It is also possible to compare an agent's payoff to her safety level of the game. If the agent's reward is equal or greater than the safety level, then the agent scores a 1, else the agent scores a 0. This measure can be used to calculate the percentage of stage games in a repeated game where an agent received a payoff above the safety level.

2.4.3.3 Convergence to a Nash Equilibrium

Nash equilibria are sometimes considered the multiagent version of optimal play because an opponent's game play is explicitly taken into account. An agent not in a Nash equilibrium could be achieving a reward close to or better than that obtained in the equilibrium. Recent work has questioned whether the focus on converging to a Nash equilibrium is correct [Shoham et al., 2004].

Walsh et al. [2003] use the $\ell_2$ norm as a measure of the distance between the agent's strategy and the Nash equilibria of the game. This metric allows measurement of how close an agent's strategy is to a Nash equilibrium during the repeated game. This is as opposed to a single measure at the end of the game which is "there was convergence" or "there was no convergence". A problem with this approach is that it measures how close an individual agent's strategy is to a Nash equilibrium strategy. It does not look at whether the opponent is playing close to the same Nash equilibrium.

A proposed update on this method is to use the $\ell_1$ norm, which is more meaningful than the $\ell_2$ norm in high dimensions (for example Aggarwal et al. [2001]). The norm is calculated for all agents on each equilibrium and the distances of the agents are summed per equilibrium. By taking the minimum of these joint distance sums, we measure how far the agents are jointly from
2.5. Conclusion

an equilibrium. When this metric has a value of 0, the agents have converged to an exact Nash equilibrium.

2.4.3.4 Estimating the Opponent’s Strategy

To measure how well an agent is estimating her opponent’s strategy, we have the actual and the estimated strategy. Viewing these two strategies in a payoff space allows us to use a norm distance measure as a comparison. Equation (2.7) gives the formula for the \( \ell_1 \) norm. \( \delta_t \) is the vector of differences between the estimate and actual probability distribution for the opponent's actions. \( N_a \) is the total number of actions and \( \delta_t(a) \) is the difference for the \( a^{th} \) action at time \( t \).

\[
\ell_1 \text{ norm} = \sum_{a=1}^{N_a} |\delta_t(a)|
\]  

(2.7)

In any of the above metrics, if an experiment involves multiple runs and/or instances of different games, then there are various methods for combining or aggregating the results. This includes the traditional statistics of mean, median and total, which can be calculated across specific sets of games or opponents. It can also be misleading if results from a single metric are used as the sole basis for comparing algorithms.

2.5 Conclusion

The first section of this chapter described the Q-learning algorithm for single agent environments. A brief introduction to game theory and the concepts that are important to this work was also given.

In a multiagent environment, an agent's reward is affected by the actions of all agents. Thus, there is no clear definition for an optimal action. Some of the multiagent algorithms use two phases to learning. They attempt to estimate what strategy their opponent is playing and then learn what actions to play in response to this. Algorithms that fall into this category include fictitious play and the gradient-based algorithms. Alternatively, algorithms can select actions based on beliefs about their opponent. Algorithms in this category include Q-learning which ignores the existence of other agents, minimax-Q which assumes her opponents are trying to hurt her and the determined and Pareto agents which have assumptions about how their opponent will respond to their actions.

This chapter also described previous work in testing methodologies. This included software platforms that are either directly related to game theoretic research (Gambit, GAMUT) or to other areas of Artificial Intelligence (BNT, CLS²). In the multiagent literature, there has not been a standard experimental methodology. This variance in testing technique makes it difficult to compare results between different papers. This has resulted in a body of work containing knowledge on individual algorithms, but very little knowledge on how all of the algorithms compare against each other.

Different metrics for judging algorithm performance were described. No metric is perfect at measuring algorithm performance and thus we suggest that multiple metrics are used to compare algorithms.
Chapter 3

A Platform for Multiagent Learning

One common approach to running experiments is to write one-off code tailored to the particular experiment. Often the code is not used again. While this approach is appropriate for quick and dirty experiments meant to test specific features of a new algorithm, it can make replication of the experiment difficult, and can discourage the sort of exploratory experimentation that is needed to gain understanding of the complex interactions between multiagent reinforcement learning algorithms. This section describes an open and reusable platform that we have implemented for conducting experiments with multiagent reinforcement learning algorithms in repeated games. Interspersed throughout this chapter are descriptions of the graphical user interface (GUI) from the platform.

3.1 Aims and Objectives

The aim of this work is to create a testing platform for multiagent learning algorithms. Often, when an experiment needs to be run, code is written for that one experiment and not used again. To avoid being used only once, a testing platform should meet the following criteria:

- Easy to use, hiding complexity from the user
- Ability to run large-scale experiments
- Wide selection of algorithms and metrics
- Variety of visualization methods available
- Be easy to extend
- Allow experiments to be reproduced with minimal effort
- Cover the full pipeline of an experiment:
  1. Setting up the experiment
  2. Running the experiment and generating metrics/measurements
3.2. The Platform Architecture

3. Combining the metrics over different runs/trials and combinations of experiment features
4. Visualizing the combined metrics
5. Based on the analysis, perform new experiments if required

A user's interaction with the platform is through a Graphical User Interface. We wanted to hide the underlying complexity of the programming from the user and allow the user to start running experiments as quickly as possible.

3.1.1 Setting

The platform is used for running experiments on two-player repeated games. A game can be understood as having a set of game-theoretic properties including the set of Nash equilibria, the set of strategies that survive iterated removal of dominated strategies, Pareto-optimal outcomes and safety levels. Though not all agents require these properties, we assume that all agents are able to calculate them. Furthermore, we assume that each agent is aware of both her own and her opponent's payoff matrix.

In a given stage game agents are aware of the past history (actions that were played in previous stage games) but are not aware of which strategy their opponent will play in the current stage game or which algorithm the opponent is using to determine this strategy. This assumption is based on the principle that an agent’s strategy should be based on observing the opponent’s play, rather than on knowledge of the opponent’s internal state.

3.2 The Platform Architecture

Creating a testing platform is an arduous and iterative task. The first step is to decide what type of experiments should be runnable. This suggests various architecture designs and methods that allow such experiments. These methods may then suggest additional ideas for new types of experiments. This process is repeated until the set of experiments and architecture to support them is finalized.

Our multiagent learning platform was written in object oriented Matlab. We chose Matlab for its rapid prototyping ability, excellent collection of numerical algorithms and built-in data visualization routines. The user does not need to interact with Matlab's command line, as all parameter setting, metric specification and visualization routines are conducted through GUls.

Experiments are run in a tournament fashion, with each algorithm running against all other algorithms, including in self play. Each pairing of algorithms is run twice, with each agent taking turns to play as the row and the column player against the other agent. While in zero-sum and symmetric games it makes no strategic difference to an agent whether she plays as the row or column player, in general games the role each agent plays is important. Allowing each algorithm to play both as the row and the column player prevents bias. Each pairing of algorithms plays on a set of instances of game generators, with each pairing playing on the same instances.
3.2. The Platform Architecture

If there are \( n \) agents, \( g \) games and \( i \) instances of each game, then the total number of runs is \( 2 \times \left[ \sum_{j=1}^{n} j \right] \times g \times i \). This shows the degree of parallelism available, as each of these runs can occur in parallel. Individual runs cannot be split up as every iteration in the run relies on the previous iteration.

Each run can be identified by the following characteristics: the pair of algorithms involved and which is the row and column player, which game generator is being used and which instance of that generator is being played. Each run produces two data types, stage game data, which contains the data from that run, for example rewards received and actions played. The second data type, metric game data contains the results of the metrics for that run. The stage game data for a pairing of agents and game generator is combined to form a data object. The metric game data types are combined into a statistics object, which is similarly representative of a generator and single combination of agents. These objects have the structure shown in Figure 3.1.

![Figure 3.1: Format of a data object, combining the results of agent 1 against agent 2 on instances from a game generator. The first row corresponds to agent 1 playing as the row player and agent 2 playing as the column player in the game. The agent's roles are reversed in the second row.](image)

The visualization process loads data from the statistics objects. The data object is used as a backup of the experiment data. A user can delete the data objects and still be able to run all analysis on the statistics objects.

3.2.1 Algorithms/Agents

Figure 3.2 gives the class structure for the algorithms described in Sections 2.1 and 2.3.

Fictitious play is an implementation of the algorithm described in Section 2.3.1.1. Pareto agent and determined agent are the algorithms discussed in Section 2.3.1. The gradient agent is an implementation of the Infinitesimal Gradient Ascent (IGA) algorithm from Singh et al. [2000]. GIGA-WoLF [Bowling, 2004] and WoLF-IGA [Bowling and Veloso, 2001a] are two of the other gradient-based algorithms and were discussed in Section 2.3.3. The global stochastic approximation agent is an implementation of the stochastic approximation algorithm of Spall [2003], which was discussed in Section 2.3.4.2.

The random agent is an algorithm that selects actions with uniform probability. Q-learning and minimax-Q are implemented versions of the algorithms that were previously described in Sections 2.1.1 and 2.3.2.1. The Q-learning algorithm applies a single agent algorithm to the multiagent setting. The minimax-Q algorithm differs slightly from the implementation in Littman [1994] by randomising over actions that return the safety level of the game. The minimax-Q-IDR plays a minimax strategy on the reduced game after iteratively removing dominated strategies. In games where...
3.2. The Platform Architecture

Figure 3.2: Class hierarchy for the agent/algorithm classes. The root node of agent contains the basic information relevant to all agents, including the algorithm name, number of actions and strategy.

there are no dominated actions, then the strategies of minimax-Q-IDR and minimax-Q will be equivalent. Simulated annealing is an implementation of the simulated annealing algorithm described in Section 2.3.4.1.

3.2.2 Games

The testing platform obtains all of its normal form games from the game generator GAMUT [Nudelman et al., 2004]. A user inputs specific parameter values for the generators in GAMUT and these values are checked to ensure they are valid. If invalid, the platform returns the GAMUT error message. Figure 3.3 shows the game setup window for the Location Game as well as an error message that is displayed when an invalid parameter value is set.

Each game has its payoffs normalized to the range \([-1, 1]\). This procedure is done automatically by GAMUT when the games are generated. Normalization is done for two reasons. Firstly, it allows easier comparisons to be made across games. All games have different default payoff bounds and it is difficult to obtain average statistics across such varied bounds. Normalizing means each game has the same bounds. The second reason for normalizing is because of the gradient-based algorithms (Section 2.3.3). These algorithms use the slope of the payoffs as part of their update function. However, due to the space being defined by probabilities on the range \([0,1]\), a large difference in payoff would lead to an extreme slope and thus a large step. Normalizing the payoff space ensures the steps produced by the update step are well-behaved within the payoff space.

3.2.3 Performance Metrics

Section 2.4.3 provided background on some of the metrics that have been proposed for single agent and multiagent learning. All of those metrics have been implemented in the platform and these are discussed here in more detail.

The most basic metric, reward, is the payoff that each agent receives from a single stage game. This also becomes the basis for the number of wins metric, which is calculated each stage game.
3.2. The Platform Architecture

Game parameters

- **actions** 2 - a C - b D - I E - c F - price j G

- **description**
  - Creates an instance of the two-person Location Game based on Hotelling's original model.
  - In this game there is a street of length L. Player one has a shop set up distance a from one end of the street and player 2 has a shop set up distance b from the other end. Customers are uniformly distributed along the street and the cost of getting a good from a shop to a home on the street is c times the distance. The players must pick a price at which to sell their goods in order to maximize their profit assuming that production is free and customers will always choose the shop for which the combined good price and transportation cost is smaller.
  - Profits may be scaled if normalization is used, but relations between the parameters remain the same and are thus important.

**Actions:** symmetric version (see Section 3.2)

- a: distance between the location of player 1's store and his end of the street.
- b: distance between the location of player 2's store and his end of the street.
- L: length of the entire street. Must be > a + b, but < 1000
- c: cost per unit of transporting the goods. Must fall between 1 and 100.
- price: lowest price each player can choose. Must be > 0 and 1000. The higher the price, the less revenue each player can choose. Must be > 0 and 1000.

**Global Parameters:**

- random_seed: random seed, uses current time by default.
- output_file: the name of the file to generate, or a list of classes from intersection.
- random_params: randomize unset parameters in default ranges
- output: the name of the outputter to use. (Default: SimpleOutput)
- max_payoff: maximum payoff in matrix, set if normalization is desired.
- int_payoffs: generate integral payoffs.
- int_mult: a multiplier used before converting payoffs to integers. Defaults to 1.
- min_payoff: minimum payoff in matrix, set if normalization is desired.
- help_game: Print help info for a game.
- help_graph: Print help info for a graph.
- help_func: Print help info for a function.

Figures 3.3: The GUI windows associated with setting up a game. Part (a) is the window for setting up an instance of the Location Game. Part (b) is the error message GUI that is returned when incorrect game parameters are used, in this case the letter 'C' is being incorrectly used as the value of a parameter.

If agent A receives a higher reward than agent B, then A is said to have won that stage game. This allows calculation of the percentage of stage games that an agent won in a repeated game.

*Regret* considers how much better off the agent would have been if she had played the best among some family of candidate strategies, pretending that the opponent's sequence of actions would not have been affected. Here we consider regret with respect to stationary pure strategies, which was used by Bowling [2004] for the GIGA-WoLF algorithm. This is a fairly weak form of regret, because it is with respect to such a restricted class of strategies. An agent that receives negative regret played a strategy that was superior to any pure strategy. Positive regret means an agent would have been better off playing some pure strategy. Equation (3.1) gives the formula for calculating total regret. \( a_i \) is a pure strategy for agent i and \( u_i(\sigma - i, a_i) \) is the payoff received by
3.2. The Platform Architecture

agent $i$ from playing the pure strategy against the opponent’s strategy $\sigma_{-i}$.

$$R = \max_{a_i \in A_i} \sum_{t=1}^{T} \left( u_i^{(t)}(\sigma_{-i}, a_i) - u_i^{(t)}(\sigma_{-i}, \sigma_i) \right)$$

(3.1)

Incentive to deviate is based on the ideas of Walsh et al. [2003]. This value is the difference between the reward that an agent would have received if she had played a best response every stage game and the reward she actually received. This can be seen as being more of an extreme version of regret, because it is with respect to any possible strategy that an agent could play. Equation (3.2) gives the formula for calculating agent $i$’s best response strategy to an opponent’s mixed strategy $\sigma_{-i}$. $E[u_i(\sigma_{-i}, \sigma_i)]$ is the expected payoff to agent $i$, with $\sigma_i$ being agent $i$’s strategy and $\sigma_{-i}$ being the opponent’s strategy. Equation (3.3) gives the formulas for calculating the incentive to deviate. $br_i(a_{-i})$ is agent $i$’s best response to her opponent’s action $a_{-i}$.

$$br_i(\sigma_{-i}) = \arg\max_{\sigma_i} E[u_i(\sigma_{-i}, \sigma_i)]$$

(3.2)

$$ID_{\sigma_i} = \max(0, u_i(a_{-i}, br_i(a_{-i})) - u_i(a_{-i}, a_i))$$

(3.3)

The $K$-competitiveness metric measures what percentage of the reward from the best response to the opponent’s action the agent received. This metric provides another measure for comparing the reward that an agent received to the reward that she could have received. The formula for this metric is given in Equation (3.4).

$$ke_{\sigma_i} = \frac{u_i(a_{-i}, a_i)}{u_i(a_{-i}, br_i(a_{-i}))}$$

(3.4)

Equation (3.5) is the formula for using the $l_2$ norm as a measure of convergence to a Nash equilibrium [Walsh et al., 2003]. This was discussed in Section 2.4.3.3. The result is the minimum of the distances between the agent’s current strategy and each equilibrium from the set of equilibria $E$. A second measure of convergence was introduced in Section 2.4.3.3. This uses the $l_1$ norm and takes into account how far both players are from all equilibria. The formula is given in Equation (3.6). $e_i$ denotes player $i$’s strategy in the Nash equilibrium $e$.

$$l_{2}(t) = \min_{e \in E} \sum_{a=1}^{N_a} \sqrt{(\sigma_i(a) - e_i(a))^2}$$

(3.5)

$$l_{1}(t) = \min_{e \in E} \left( \sum_{a=1}^{N_a} |\sigma_1(a) - e_1(a)| + \sum_{a=1}^{N_a} |\sigma_2(a) - e_2(a)| \right)$$

(3.6)

The expected utility metric is the same as that described in Section 2.2. This takes into account the current strategy of each agent and their payoff matrices. The reward vs. safety level metric was described in Section 2.4.3.2. This compares the agent’s reward to her safety level payoff.
The metrics used to judge an agent's ability to estimate its opponent's strategy are based on norm measurements. This views an agent's strategy as a point in a high dimensional space. Two such points can then be compared with norms. The formula for the $\ell_1$ norm is given in Equation (3.7). $q_t$ is the vector of differences between the estimate and actual probability distribution for the opponent's actions. $N_o$ is the total number of actions and $q_t(a)$ is the difference for the $a^{th}$ action at time $t$.

$$\ell_1 \text{ norm} : \sum_{a=1}^{N_o} |q_t(a)| \quad (3.7)$$

The metrics are grouped into four categories, depending on what they measure. These categories as well as a summary of each metric is given in Table 3.1.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Metric</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best response</td>
<td>K-competitiveness</td>
<td>Percent of best response payoff achieved</td>
</tr>
<tr>
<td></td>
<td>Sum of incentives to deviate</td>
<td>Extra reward available if an agent were to play a best response</td>
</tr>
<tr>
<td>Nash equilibrium</td>
<td>$\ell_2$ norm convergence</td>
<td>Euclidean distance between agent's strategy and the closest Nash equilibrium strategy</td>
</tr>
<tr>
<td></td>
<td>Joint $\ell_1$ convergence</td>
<td>$\ell_1$ measure of how far the agents are jointly from the closest Nash equilibrium</td>
</tr>
<tr>
<td>Rewards obtained</td>
<td>Expected utility</td>
<td>Expected utility of a game</td>
</tr>
<tr>
<td></td>
<td>Number of wins</td>
<td>Number of stage games in which agent's reward is higher than the opponent's reward</td>
</tr>
<tr>
<td></td>
<td>Regret</td>
<td>Extra reward available if an agent had played a pure strategy</td>
</tr>
<tr>
<td></td>
<td>Reward obtained</td>
<td>Actual reward obtained each stage game</td>
</tr>
<tr>
<td></td>
<td>Reward vs safety level</td>
<td>How often an agent achieves a reward equal or greater than the safety level of a game</td>
</tr>
<tr>
<td>Strategy tracking</td>
<td>$\ell_1$ norm</td>
<td>Sum of absolute differences between estimated and actual strategy of opponent</td>
</tr>
<tr>
<td></td>
<td>$\ell_2$ norm</td>
<td>Square root of the sum of squared differences between estimated and actual strategy of opponent</td>
</tr>
<tr>
<td></td>
<td>$\ell_\infty$ norm</td>
<td>Largest absolute value difference between the estimated and actual probabilities of an opponent taking a specific action</td>
</tr>
<tr>
<td></td>
<td>Action causing $\ell_\infty$ norm</td>
<td>Action that causes the $\ell_\infty$ norm</td>
</tr>
</tbody>
</table>

Table 3.1: Summary of implemented metrics
3.3 Setting Up and Running an Experiment

The pipeline for running an experiment is split into three phases. The first is the setup phase: selecting the agents, games, number of runs or instances, number of iterations and the metrics to be recorded. The second phase is the actual running of the experiment. This phase can be done on a single machine in a batch process, or in parallel on a cluster. The final phase is to visualize the metrics and to analyze the resulting data. The entire pipeline is shown in Figure 3.4.

Figure 3.4: The pipeline for running experiments on the platform. The illustration is shown for a single combination of two agents and a single game generator.

The GUI for setting up an experiment, phase 1 of the pipeline, is shown in Figure 3.5. The user selects the agents, games, metrics and parameters for the experiment. A configuration file describing this setup can be saved prior to running the experiment. This file contains all of the information necessary to rerun the experiment. This file can be made available for download, allowing researchers to reproduce one another’s experiments.

3.4 Visualization

Once an experiment has been run, we need to visualize the results. Due to the volume of data collected in a typical experiment, we would not want to see a statistic for each run, but instead want to be able to visualize statistics over multiple runs. Although visualization could be performed by the user at the command line, it is nontrivial to specify to the system which data to vary in a visualization, which data to aggregate, and which data to omit. For this reason we built a GUI-driven visualization system.

Figure 3.4 presented a pipeline diagram for running experiments on the platform. The visualization phase was not shown in that diagram, but the process for visualizing results is now given in Figure 3.6. This process allows the user to visualize the results of metrics over a set of agents, generators, instances and iterations. The user is able to make these selections through the GUI.
Figure 3.5: The graphical user interface for phase one of an experiment, where the parameters are set. This allows users to select the agents, games, metrics and parameter values for an experiment.

3.4.1 Step 1: Select the Agents, Runs and Iterations

The first step in the visualization process is to select which set of agents we want to visualize. When we select agents A and B, we are pulling the data from A as the row player and B as the column player and vice versa. We can also choose to limit some of the agents to be row players and others to be column players. In the upper left block of Figure 3.7, the fictitious play and minimax-Q-IDR
3.4. Visualization

Figure 3.6: The pipeline for visualizing the results of an experiment

agents have been selected. When a user clicks to select agents, a popup window such as the one shown in Figure 3.8(a) is displayed. This lists all of the agents that were used in the experiment and
3.4. Visualization

those agents that have been selected for visualization. Users are able to move agents between the
two listboxes. Once users are satisfied with their choices, the new selection is shown in the relevant
box in the interface shown in Figure 3.7.

The next step in the process is to decide from which runs or instances of the game generators we
want to view metrics. A user can thus decide to use all instances, or perhaps only one instance per
generator. Each instance or run is composed of a number of iterations. We can similarly chose from
which iterations of the selected instances we want to view metrics. This provides control down to
single stage game interactions between agents. When a user selects the runs or iterations they want
to use, separate popup windows with the selection options are displayed. These two windows are
illustrated in Figures 3.8(b) and 3.8(c).

Figure 3.8: GUIs used in step 1 of the visualization process, where the agents, runs and iterations
to be used in the visualization process are selected

3.4.2 Step 2: Select the Generators

Once the user has selected the agents, runs and iterations that will be used in the visualization
process, the next step is to select the game generators. The generators that are currently selected
are displayed in the top middle listbox in the GUI of Figure 3.7. When a user wants to select the
generators, the window in Figure 3.9 pops up. Similar to the agent popup window, a user is shown
the games currently selected for the visualization process and games that are still available. In the
figure, three games are currently selected.

Figure 3.9: Step 2 of the visualization process, selecting the generators.

3.4.3 Step 3: Select the Free Variables

The third step is to select free variables. These are the variables whose values will be varied in the
visualization. The possible free variables are agent, agent, games, iterations and runs/instances.
Agent is repeated twice because it allows metrics to be visualized over either pairs of agents or
individual agents.

The third step is to select free variables. These are the variables whose values will be varied in
the visualization. The possible free variables are agent, agent, games, iterations and runs/instances.
Agent is repeated twice because it allows metrics to be visualized over either pairs of agents or
individual agents. The free variables form a four dimensional table. The dimensions are defined by
(algorithm pairing, game, run/instance, iteration), which correspond to the possible free variables.
Each cell in the table contains the results of all metrics recorded for that pairing of agents playing
on a specific instance of a generator in one iteration/stage game. Selecting one or two free variables
means that we want to project the table down to one or two dimensions. The dimensions of the table
that are kept correspond to the free variables that are selected. The dimensions corresponding to the
free variables that are not selected are the dimensions that are aggregated across in the projection.
For example, if we wanted to view the performance of specific agents on specific games, then the
free variables would be agent and games. The graph would show results for each agent on each
game, but aggregated over all opponents, instances and iterations. If we want to obtain a statistic
of how every agent did, but want an aggregated result against all opponents, on all games and all
instances and iterations, the free variable would then just be agent. To view the performance of each
agent against each opponent, the free variables would be agent and agent and the graph would show
results for each combination of agents, aggregated over all the games, instances and iterations.

The process for selecting the free variables is the same as agents and games. A popup window, shown in Figure 3.10, displays the list of choices for free variables and any free variables that have already been selected.

![Variable control](image)

Figure 3.10: The GUI for step 3 of the visualization process, where the free variables are selected. The available variables are listed in the left-hand box and the selected variables in the right-hand box.

### 3.4.4 Step 4: Select the Metric

The fourth step in the process is to select the dependent variable. This corresponds to the metric that we want to visualize in a plot. The metrics that are available are those that were recorded during the experiment that we are visualizing. When we want to select the metric, the popup window in Figure 3.11 is displayed. This provides a list of the available metrics and shows which metric is currently selected. Metrics can be easily moved between the boxes, though only one can be placed in the right-hand box, because we currently allow visualization of a single metric on a graph.

### 3.4.5 Step 5: Select the Visualization Routine

The fifth step in the visualization process is to select the visualization routine that will be used to present the metric. Currently, there is a standard line plot, bar plot, box plot and scatter plot implemented. The line plot allows use of error bars to display the first and third quartile of the data. The popup window for selecting the graph is shown in Figure 3.12. The graph currently selected in that window is the box plot.
3.4. Visualization

STEP 4: Select a dependent variable (metric) to visualise

- Dependent variables currently available:
  - Percent of best response achieved
  - Best response measure (smaller is better)
  - 1-norm measure for Nash equilibrium
  - Measure of regret (lower is better)
  - Reward obtained (larger is better)
  - 1-norm on estimating opponent's strategy

- Dependent variable currently selected:

Figure 3.11: The popup GUI for step 4 of the visualization process, where the dependent variable is selected. The metrics that were recorded during the experiment and are not selected are shown in the left-hand box. The currently selected metric is shown in the right-hand box.

STEP 5: Select and configure visualisation method

- Visualisation methods currently available:
  - Bar plot
  - Line plot
  - Scatter plot

- Visualisation method currently selected:

Figure 3.12: The GUI for step 5 of the visualization process, where the graph type is selected. The currently selected graph is shown in the right-hand window, with the remaining graph types shown in the left-hand window.

3.4.6 Step 6: Select the Aggregation Method

Once free variables have been selected, the 4D table containing our data is reduced in dimensionality. The dimensions which correspond to the free variables that were not selected are eliminated through an aggregation method. The final step in the visualization process is to select this aggregation method. Our platform currently supports three methods for achieving this aggregation, no
3.5. Limitations of the Platform

aggregation, averaging the data and totalling/summing the data. The no aggregation method is used in the box and scatter plots, where the plot itself provides a view of the entire data set. The line and bar plot use the averaging and totalling methods. The popup window for selecting the aggregation method is shown in Figure 3.13.

![Figure 3.13](image)

Figure 3.13: The GUI for step 6 of the visualization process, where the aggregation method to be applied to the non-selected free variables is chosen.

Putting this all together, if the selected free variable is agent, the dependent variable is reward and the aggregation method is averaging, then a single reward statistic is calculated for each agent. This corresponds to the average reward that each agent achieved against all opponents, on all of the selected game generators. If a user selected agent and agent as the free variables, regret as the dependent variable and total as the aggregation method, then the graph would display the total regret that each agent achieved against each opponent, over all game generators.

3.5 Limitations of the Platform

The current version of the platform is limited to two agents playing each other in a repeated game. Repeated games are arguably the simplest form of repetitive play on normal form games. There is also a large amount of literature which introduces new algorithms for playing repeated games. This provides a potential group of users for this platform.

How to extend the framework to handle repeated games with more than two players remains an open problem. There are both technical and conceptual problems. Technically, a different method would need to be found for combining the results from all the combinations of agents. Conceptually, it is not clear how some of the algorithms would need to be changed in order to handle more than two opponents.
3.6 Conclusion

This chapter presents the architecture of a platform for testing multiagent learning algorithms. The platform is designed to facilitate large scale reproducible experiments and to visualize the results.

A large portion of this chapter describes how the platform was built and the various algorithms and metrics that have been implemented. User interaction is through customized graphical user interfaces, hiding the complexity of the platform from the user. Visualization of the experimental results is through a process that allows different combinations of metrics and variables to be selected.
Chapter 4

Empirical Testing

In order to demonstrate the usefulness of the testing platform, a large experiment was run and the results were analyzed. The first section of this chapter lists the algorithms, parameters and metrics that were used in this experiment. The remainder of the chapter is devoted to analyzing the data that was collected.

4.1 Experimental Setup

The experiment was set up using a subset of the available algorithms, metrics and games. In total, six algorithms, seven metrics and thirteen game generators were chosen. The experiment was conducted in a tournament fashion, with each algorithm playing each other algorithm. The algorithms and their parameters are listed in Table 4.1. The parameters for each algorithm were either set to values from the paper in which they were introduced or to decay below a certain value after 100,000 iterations. Each agent's beliefs about its opponent's strategy was initialized to a uniform distribution over the opponent's actions. The initial belief update rate was set to 0.9 for all agents that used a discounted stochastic approximation technique to estimate an opponent's strategy. The decay rate for this parameter was set to decay the initial belief update rate to $10^{-6}$ after 100,000 steps (i.e., a decay rate of 0.999886). The update values for GIGA-WoLF and GSA that are displayed in Table 4.1 are based on information supplied in Spall [2003, p. 190].

The algorithms were tested on thirteen game generators, which are split into two groups. The first twelve generators were used to produce games with ten actions; these generators are listed in Table 4.2. We generated 100 instances from each of these generators. The thirteenth generator was the TwoByTwo game generator, which generates from all of the 85 distinct 2x2 games (See Rapoport et al. [1976] for a list). The goal with the 2x2 games was to obtain results over a random distribution of 2x2 games. The ability to sample from the set of possible 2x2 games comes at the expense of being unable to view results for a specific 2x2 game, for example Prisoner's Dilemma. Such experiments would be a worthwhile avenue for future work. We generated twelve times as much data with the 2x2 generator, giving us a total of 1200 10x10 and 1200 2x2 game instances.

A subset of the possible metrics was recorded during the experiment. The metrics were selected based on which metrics measured different aspects of an algorithm's performance. The metrics are
4.1. Experimental Setup

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>GIGA-WoLF</td>
<td>$\gamma_t : \frac{10}{(A+t)^{0.30}}; A : 5000 = 5% \text{ of } 100000$</td>
</tr>
<tr>
<td></td>
<td>This ensures that the update is large at the start and gradually decays over time to $&lt; 10^{-3}$ after 100000 iterations.</td>
</tr>
<tr>
<td>GSA-Agent</td>
<td>$\alpha_t : \frac{10}{(t+1+\frac{1}{A})^{0.00}}; A : 5000; \beta_t : \frac{0.3}{(t+1)^{0.33}\log(t+1)}$</td>
</tr>
<tr>
<td></td>
<td>This provides the same decrease rate for $\alpha$ as in GIGA-WoLF. We want $\beta_t$ to decrease at a faster rate and this setting causes it to be $&lt; 10^{-4}$ after 100000 iterations.</td>
</tr>
<tr>
<td>minimax-Q</td>
<td>Initial exploration rate: 0.2; Exploration rate decay: $10^{\log(0.01)/100000}$</td>
</tr>
<tr>
<td>minimax-Q-IDR</td>
<td>$\alpha : 1; \alpha_{\text{decay}} : 10^{\log(0.01)/100000}; \gamma : 0.5$</td>
</tr>
<tr>
<td>Q-learning</td>
<td>Values taken from the minimax-Q paper [Littman, 1994]. The decay rates were adapted from the paper, which used one million iterations.</td>
</tr>
<tr>
<td>Fictitious play</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 4.1: The six algorithms used in the experiment and the values of their associated parameters. $t$ reflects the value of the current iteration.

<table>
<thead>
<tr>
<th>Generators of games with ten actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arms Race</td>
</tr>
<tr>
<td>Cournot Duopoly</td>
</tr>
<tr>
<td>Dispersion Game</td>
</tr>
<tr>
<td>Grab The Dollar</td>
</tr>
<tr>
<td>Minimum Effort Game</td>
</tr>
<tr>
<td>Travellers Dilemma</td>
</tr>
<tr>
<td>Bertrand Oligopoly</td>
</tr>
<tr>
<td>Covariant Game</td>
</tr>
<tr>
<td>Guess Two Thirds Average</td>
</tr>
<tr>
<td>Location Game</td>
</tr>
<tr>
<td>Majority Voting</td>
</tr>
<tr>
<td>War Of Attrition</td>
</tr>
</tbody>
</table>

Table 4.2: The twelve game generators that were used to generate the games with ten actions.

listed in Table 4.3.

<table>
<thead>
<tr>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-competitiveness</td>
</tr>
<tr>
<td>$l_1$ norm convergence sum</td>
</tr>
<tr>
<td>Regret</td>
</tr>
<tr>
<td>$l_1$ norm strategy estimation</td>
</tr>
<tr>
<td>Sum of incentives to deviate</td>
</tr>
<tr>
<td>Number of wins</td>
</tr>
<tr>
<td>Reward obtained</td>
</tr>
</tbody>
</table>

Table 4.3: List of the seven metrics recorded during the experiment.

A single run on a game instance consisted of 100 000 stage games, with the first 90 000 allowing the agent’s learning parameters to settle and statistics being recorded on the last 10 000 stage games. Each instance of a game was played twice, with the agents taking turns to be the row and column players.

We used the Kolmogorov-Smirnov Z test to test for statistical similarity between distributions.
4.2 Observations from the Empirical Study

The test is performed in the SPSS statistics package, with the data imported from Matlab. We use a p-value \( \leq 0.05 \) to indicate a statistically significant difference between the distributions being compared.

The majority of graphs presented in this article are box plots. The middle line of the box represents the median of the distribution, the leftmost and rightmost edges of the box are the 25\textsuperscript{th} and 75\textsuperscript{th} percentile values. These percentiles correspond to the values in the distribution, below which are 25\% and 75\% of the distribution respectively. The two “whiskers” are \( 1.5 \times IQR \) from the edges of the box, with Inter Quartile Range \( (IQR) = (75\textsuperscript{th} \text{ percentile} - 25\textsuperscript{th} \text{ percentile}) \). Any crosses are outliers that lie outside the \( 1.5 \times IQR \) distance.

The results presented in this dissertation do not select among either the reward or convergence to a Nash equilibrium paradigms. Rather, we want to make claims that are backed up by empirical evidence. Each of the two paradigms are discussed, first reward and then convergence to a Nash equilibrium and then reward. Outside of the metrics that judge performance, we also look at one of the factors in the inner workings of some of the algorithms: the estimation of an opponent’s strategies, as previously discussed in Section 2.3.5.

4.2 Observations from the Empirical Study

We first present a series of high level observations that have emerged during this empirical study. We list these before going into detail on individual results in order to provide the reader with an overview of our results and to single out noteworthy points.

Observation 1 No algorithm dominates.

In the set of 2x2 and 10x10 generators used for this experiment, there is no single algorithm which beats any other algorithm on all games. While an algorithm might perform well on one metric, for example GIGA-WoLF on regret, there is no algorithm which dominates on both the reward and convergence to a Nash equilibrium metrics.

Observation 2 An algorithm needs to be tested on a variety of generators in order to obtain accurate performance information.

This is one of the more important observations resulting from this work. There is a large amount of variance in the results, based on different game generators and their payoff structures. It is dangerous to claim anything about an algorithm’s general performance based on an experiment involving a single instance of one game. Testing needs to be done over a wide variety of games.

Observation 3 There is no relationship between algorithm performance and the number of actions in a game and thus it is important to test on games with different numbers of actions.

We ran a small experiment to determine whether there is a trend between the reward that algorithms obtain and the number of actions in a game. Two generators, the Covariant game and Grab the dollar were used in the test, generating twenty random instances of each game size from
two to ten actions. We calculated the average reward achieved by each algorithm in each different game size, averaged over all opponents (Figure 4.1). A linear regression test\(^1\) showed that there was no linear relationship between the reward that an agent obtained and the number of actions in these game generators (Table 4.4). When claims are made about algorithm performance it should be with reference to the specific game size tested, since the performance of an algorithm on other game sizes cannot always be predicted. While it may be possible that a trend exists for other generators, at the least our experiment shows that such a trend does not exist for all generators.

Figure 4.1: Average reward received by each agent in different sized instances of the the Covariant and Grab the dollar games.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Range of the gradient with 95% confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Covariant game</td>
</tr>
<tr>
<td>Fictitious play</td>
<td>[-0.0170, 0.0291]</td>
</tr>
<tr>
<td>GIGA-WoLF</td>
<td>[-0.0124, 0.0311]</td>
</tr>
<tr>
<td>GSA</td>
<td>[-0.0134, 0.0311]</td>
</tr>
<tr>
<td>minimax-Q-IDR</td>
<td>[-0.0411, 0.0028]</td>
</tr>
<tr>
<td>Q-learning</td>
<td>[-0.0113, 0.0288]</td>
</tr>
</tbody>
</table>

Table 4.4: The range of the gradients of the trend lines through average reward. The ranges are the 95% confidence interval for the slope of the line.

\(^1\)The results of the tests are the slope of the line that would fit each set of data with 95% confidence. If the entire range of the slope is positive (negative), then there is an increasing (decreasing) trend.
two to ten actions. We calculated the average reward achieved by each algorithm in each different game size, averaged over all opponents (Figure 4.1). A linear regression test\(^1\) showed that there was no linear relationship between the reward that an agent obtained and the number of actions in these game generators (Table 4.4). When claims are made about algorithm performance it should be with reference to the specific game size tested, since the performance of an algorithm on other game sizes cannot always be predicted. While it may be possible that a trend exists for other generators, at the least our experiment shows that such a trend does not exist for all generators.

![Figure 4.1: Average reward received by each agent in different sized instances of the Covariant and Grab the dollar games.](image)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Range of the gradient with 95% confidence</th>
<th>Covariant game</th>
<th>Grab the Dollar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fictitious play</td>
<td>[-0.0170, 0.0291]</td>
<td></td>
<td>[-0.0143, 0.0707]</td>
</tr>
<tr>
<td>GIGA-WoLF</td>
<td>[-0.0124, 0.0311]</td>
<td></td>
<td>[-0.0210, 0.0623]</td>
</tr>
<tr>
<td>GSA</td>
<td>[-0.0134, 0.0311]</td>
<td></td>
<td>[-0.0594, 0.0661]</td>
</tr>
<tr>
<td>minimax-Q-IDR</td>
<td>[-0.0411, 0.0028]</td>
<td></td>
<td>[-0.0179, 0.0849]</td>
</tr>
<tr>
<td>Q-learning</td>
<td>[-0.0113, 0.0288]</td>
<td></td>
<td>[-0.0156, 0.0745]</td>
</tr>
</tbody>
</table>

Table 4.4: The range of the gradients of the trend lines through average reward. The ranges are the 95% confidence interval for the slope of the line.

\(^1\)The results of the tests are the slope of the line that would fit each set of data with 95% confidence. If the entire range of the slope is positive (negative), then there is an increasing (decreasing) trend.
4.2. Observations from the Empirical Study

**Observation 4** Having information about the games and opponents being played can provide insight into which algorithm to use in that environment.

This is because algorithms perform differently on different generators and against different opponents. This observation is consistent with arguments made by Shoham et al. [2003]: “categorizing strategic environments, that is, populations of agent types with which the agent being designed might interact”. However, it is not a simple task to cluster agents into classes. Thus, an environment may be best described by a distribution over possible agents, rather than over groups of similar agents. The aim would be to choose an algorithm that does relatively well against all of the algorithms.

**Observation 5** The choice of which algorithm to use in an environment should also depend on the paradigm being used to evaluate the algorithms.

Measuring convergence to a Nash equilibrium and obtaining reward are shown to not be equivalent in Claim 12. In deciding what algorithm to use in a scenario or in designing a new algorithm, it choose a specific metric which will be used for evaluation. It is possible to create an algorithm that does extremely well on a specific metric; e.g., GIGA-WoLF is very effective at minimizing regret.

**Observation 6** If convergence to a Nash equilibrium is important, then a good algorithm to use is fictitious play.

Fictitious play obtained the highest rates of convergence to a Nash equilibrium, as we will show in Section 4.5. This is especially true for the larger 10x10 set of generators.

**Observation 7** If reward is important, then GSA or Q-learning are good options.

Figure 4.9 showed that GSA and Q-learning obtained the highest reward averaged over all opponents and generators. Q-learning had the highest median reward and GSA had the highest mean reward. These algorithms both scored relatively well against each of the other opponent algorithms.

**Observation 8** Iterated removal of dominated actions may benefit algorithms other than minimax-Q.

As shown in Claim 10, minimax-Q-IDR consistently achieves higher reward than minimax-Q and generally outperforms it. These algorithms play the same strategy except for the iterative removal of dominated strategies. As far as we know, the combination of IDR with the minimax-Q strategy is new to this paper; the approach of removing dominated strategies could be beneficial to other algorithms. We mention this as an open problem in the conclusion.

**Observation 9** Large experiments were easier to conduct using our platform.
Our platform proved extremely useful for this research. It was a straight-forward process for setting up the parameters for the experiment. The platform then automatically split up the experiment into individual jobs for the cluster and these jobs were run in parallel. This system ran for seven weeks with an approximate total CPU time of two years. We were forced to rerun some small portions of the experiment due to small cluster outages; nevertheless, our platform made it easy to combine all of our experimental results once all of our data had been collected.

The platform also served us well in the analysis phase of our experimental work. The GUI greatly speeded the selection of parameters of interest and probably meant that we ran many more iterations of analysis (visualization, formation of a hypothesis, testing of the hypothesis through a new visualization, ...*) than we otherwise would have done. There was rarely a need to run new experiments when testing a new hypothesis as all of the data combinations were already present. When a smaller sub-experiment was necessary to test a hypothesis, this was set up by loading the configuration file from the main experiment, ensuring consistency with the parameters in the main experiment.

4.3 Reducing the Size of the Experiment Space

Earlier, we discussed the idea that the results of an experiment can be thought of as being stored in a four dimensional table indexed by (iteration, instance, game, algorithm pairing). For the experiments we conducted this table contained $21 \times 24 \times 100 \times 10,000 = 504$ million cells. We clearly had too much data to permit us to examine every cell, so a method was required to reduce the table's dimensionality. We consider how (and whether) each dimension can be reduced, examining each in turn. The graphs that are presented in this section are all generated directly from the platform.

4.3.1 Collapsing the Iterations Dimension

Claim 1 The iterations dimension can be reduced by averaging metric values across iterations.

We have observed that metric values often differ from one iteration to the next, which means that it would be inappropriate to keep only a single iteration as representative of the whole sequence. It is nevertheless desirable to keep only a single value for each metric and each run; therefore, we average over the iterations. We observed no cases where metric values are not either constant or drawn from a small set of values.

As an example, Fictitious play, GIGA-WoLF and GSA obtain constant values in Figure 4.2. In Figure 4.2, Q-learning and minimax-Q-IDR do not converge, but receive payoffs alternating among a small set of values, sixteen values for Q-learning and nine values for minimax-Q-IDR. Minimax-Q acts similarly, but is not shown in this figure in an attempt to simplify the graph.

For the instances where the metric values converged, averaging over these values is clearly suitable technique: it does not hide or remove any of the underlying information. For the cases where the metric values do not converge, but rather come from a small set of values, averaging over these values does hide some of the underlying detail, but nevertheless preserves information about all the values. Clearly we could choose other aggregation approaches: we could take both
4.3. Reducing the Size of the Experiment Space

4.3.1 Fictitious play, GIGA-WoLF, and GSA converge. Q-learning and minimax-Q-IDR cycle between a small set of rewards. Minimax-Q was removed from the graph to make the figure clearer.

Figure 4.2: Average reward obtained by all agents over the iterations of a single instance of the Majority voting game. Fictitious play, GIGA-WoLF and GSA converge. Q-learning and minimax-Q-IDR cycle between a small set of rewards. Minimax-Q was removed from the graph to make the figure clearer.

the average and the standard deviation, or we could take the median. This point notwithstanding, our choice agrees with choices made by others in the literature: averaging has also been used in the same way by e.g., Powers and Shoham [2004] and Nudelman et al. [2004].

4.3.2 Collapsing the Instances Dimension

Claim 2 The instances dimension can be reduced by averaging metric values across instances.

Instances from the same generator that are qualitatively the same can produce different behaviour because of payoff differences. This means that we should not judge performance on a single instance, since reporting statistics from one instance would not be representative of an agent’s general performance on that generator. Reporting results by aggregating over different instances reduces the effect of the payoffs from any single instance. As above, the choice of averaging rather than aggregating in another way seems reasonable although other aggregation methods could also be defended.

For example, the Arms Race game has a single equilibrium in which both agents play their first action. If algorithms play to this equilibrium, they can still obtain different payoffs because of the different underlying payoff matrices for the different instances. Figures 4.3(a) and 4.3(b) are histograms showing the occurrence of values that each algorithm receives from the metric being visualized. Figure 4.3(a) displays the average reward obtained over 10 000 instances for 31 of the 100 instances. It can be seen that the algorithms obtain different rewards in the different instances. However, looking at Figure 4.3(b), it can also be seen that algorithms consistently converge to
4.3. Reducing the Size of the Experiment Space

Figure 4.3: Reward and $\ell_1$ distance to a Nash equilibrium for instances 20-50 of the Arms race generator. Each point is averaged over all 10,000 iterations of that instance and against all opponents.

the Nash equilibrium in the instances. Thus the differences in reward are attributable to different underlying payoff matrices, rather than different strategies.

There are also some generators for which there does not appear to be a link with reward between different instances. An example of this is shown for a subset of the instances of the Majority Voting generator in Figure 4.4. The plots for minimax-Q and GSA are omitted for clarity. This figure demonstrates how reporting results on any single instance would give a distorted view of the results.
4.3 Reducing the Size of the Experiment Space

There is no consistent pattern in algorithm performance amongst the different instances.

The method used to run the experiment supports the aggregation approach. Even though there are differences amongst the game generators and instances, every agent plays every other agent on all instances. No agent plays a more difficult game, but rather all agents are on a level playing field.

4.3.3 Collapsing the Generators Dimension

We have now discussed aggregating over iterations and instances of generators. We now look at whether this approach can be applied to the size of the generators dimension. If similar games led to similar metric values, we would have been able to reduce the games dimension by discarding some of our games.

Claim 3 Generators from the same game theoretic group do not exhibit consistently similar properties according to any of the metrics we measured.

One way of attempting to cluster games is according to their game theoretical properties. Nudelman et al. [2004] taxonomized the games GAMUT can generate according to categories such as dominance solvability and the absence of a pure strategy equilibrium. In our experiment we observed that games from the same category do not cause consistently similar dynamics among agents, but rather produce different behaviour across the different generators. This was also true for other clustering approaches that we considered.
For example, two generators that are in the same game theoretic class are Cournot Duopoly and Bertrand Oligopoly, which are both supermodular games\(^2\) and therefore both have pure strategy equilibrium. In both generators, agents converge in the median to the Nash equilibrium during play (Figures 4.5(a) and 4.5(b)). In both graphs, only the distributions for Q-learning and minimax-Q-IDR against their opponents are shown. The distribution across convergence values for the same algorithm and opponent on the two generators is however different. There is high variance in Bertrand Oligopoly, but almost no variance in the Cournot Duopoly generator. The corresponding reward distribution is also very different (Figures 4.6(a) and 4.6(b)). Based on these results it is difficult to place the two generators in a similar cluster with respect to the dynamics that occur during agent play.

### 4.3.4 Collapsing the Agents Dimension

**Claim 4** No two algorithms are similar in performance against all opponents on all games.

If two algorithms performed similarly against all opponents on all games, we would have no need to use both algorithms in an experiment. We could use just one of the algorithms and take its performance as being representative of the other algorithm. Unfortunately, as will be evident from the results presented in the rest of this section, we did not find this to be the case.

### 4.4 Experimental Results: Reward-based Metrics

Having reduced our experiment space by aggregating over iterations and instances, we now move on to considering different algorithms' performance according to specific metrics. The first group of metrics we consider are based on reward.

In a sense the average amount of reward obtained by an agent is the most fundamental metric, as agents' explicit goals are to maximize this quantity.

#### 4.4.1 Raw Reward

**Claim 5** No algorithm obtains the highest average reward against every opponent in either the 2x2 or 10x10 sets of generators.

Figure 4.7 displays the average reward obtained by each agent against each other agent in the set of 2x2 (top) and 10x10 (bottom) game generators. In the 2x2 set of games, GSA and minimax-Q-IDR each obtain the highest average reward against an opponent once and GIGA-WoLF and Q-learning each obtain the highest average reward twice.

\(^2\)The website for GAMUT describes these games as: “Supermodular games are games in which each player's strategy set is partially ordered, the marginal returns to increasing one's strategy rise with increases in the competitors' strategies (so that the game exhibits 'strategic complementarity') and, if a player's strategies are multidimensional, the marginal returns to any one component of the player's strategy rise with increases in the other components”. [http://gamut.stanford.edu](http://gamut.stanford.edu)
4.4. Experimental Results: Reward-based Metrics

In the 10x10 set of games, the three algorithms that estimate their opponent’s strategy—fictitious play, GIGA-WoLF and GSA—obtain the highest average reward. Fictitious play obtains the highest average reward once, GIGA-WoLF twice and GSA three times. Q-learning now obtains a higher average reward than minimax-Q-IDR against all opponents, which was not the case in the 2x2 set of games.

An interesting point to note is that in both the 2x2 and 10x10 sets of generators, it is never the case that the algorithm that achieves the highest average reward against a given algorithm is that same algorithm. In the 10x10 set of generators, no estimation algorithm clearly outperforms the other estimation algorithms, in the sense that each one of the estimation algorithms obtains the highest average reward against one of the other estimation algorithms.

Although we cannot identify a best algorithm, we do observe from Figure 4.7 that the Minimax-Q algorithm performs consistently badly across opponents. We will examine the behaviour of this
4.4. Experimental Results: Reward-based Metrics

Claim 6 Q-learning achieves the highest mean and median reward against all opponents in the 2x2 set of generators.

Though we cannot say that a single algorithm beats all other algorithms or that there is a consistent pattern to the ranking of all of the algorithms, we can identify a "best" algorithm based on average reward over all opponents. Q-learning obtains the highest median and mean reward in the 2x2 set of game generators when we average over all of the opponents. Figure 4.8(a) displays the reward distributions that each agent received against all opponents on the 2x2 set of games. Recall that the middle line in a box plot is the median of the distribution. Table 4.5 provides the results of a Kolmogorov-Smirnov Z test performed on the reward distributions obtained by Q-learning and the algorithm that achieves the next highest median reward, GSA. There is a statistically significant difference between the Q-learning and GSA reward distributions in the set of 2x2 game generators,
4.4. Experimental Results: Reward-based Metrics

Figure 4.7: Average reward obtained by each agent against each other agent in the set of 2x2 (top) and 10x10 (bottom) generators.

Figure 4.8(b) displays the reward distribution obtained by each agent against all opponents in the 10x10 set of generators. Fictitious play obtains the highest median reward, with the other strategy estimating algorithms, GIGA-WoLF and GSA performing similarly. Q-learning does not perform as well on the 10x10 set of generators as in the 2x2 set of games. The three estimation algorithms all obtain higher median, 25th and 75th percentile reward values than Q-learning. The estimation
4.4. Experimental Results: Reward-based Metrics

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Q-learning</th>
<th>GSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>0.355</td>
<td>0.305</td>
</tr>
<tr>
<td>median</td>
<td>0.490</td>
<td>0.356</td>
</tr>
<tr>
<td>25\textsuperscript{th} percentile</td>
<td>-0.040</td>
<td>-0.139</td>
</tr>
<tr>
<td>75\textsuperscript{th} percentile</td>
<td>0.804</td>
<td>0.086</td>
</tr>
</tbody>
</table>

Table 4.5: Kolmogorov-Smirnov test comparing the distribution of reward obtained by Q-learning and GSA in the set of 2x2 game generators.

algorithms also obtain the three highest mean reward positions in the 10x10 set of generators. The larger game size would appear to benefit those algorithms which estimate their opponent’s strategy. However, we also know that there is no trend in performance as the number of actions in the game increases (see Observation 3). This suggests that it is the structure of the games rather than simply their size which affects the performance of the algorithms.

Figure 4.9 displays the reward distribution obtained by each agent against all opponents on all generators. (That is, the game generators are not divided into 2x2 and 10x10 games as in Figure 4.8.) Q-learning has the highest median reward, due to its performance in the 2x2 set. GSA obtains a higher 3\textsuperscript{rd} percentile value, which causes GSA to have the highest mean reward.

4.4.2 Regret

We now consider different metrics which depend on reward, asking in different ways whether the agent could have obtained a higher reward by playing a different action. First, we consider regret, which asks whether an agent would have gained more reward by playing a stationary pure strategy.

The GIGA-WoLF algorithm Bowling [2004] is motivated by the theoretical guarantee that it will obtain at most zero average regret. However, it has not been shown experimentally how the regret achieved by this algorithm compares to the regrets achieved by other algorithms; nor has it been demonstrated whether GIGA-WOLF often achieves better than zero regret in practice.

Claim 8 GIGA-WoLF achieves lower average regret than other algorithms and often achieves negative regret.

Figure 4.10 shows a box plot of the regret distribution for each algorithm against all other algorithms on the set of 2x2 (top) and 10x10 (bottom) game generators. GIGA-WoLF has a substantially smaller variance than any other algorithm and consistently achieves negative regret. Averaging over all of the opponents, GIGA-WoLF achieves negative average regret in the 2x2 set of games and an average regret of \( \approx 0 \) in the 10x10 set of generators. The other algorithms do tend to achieve
4.4. Experimental Results: Reward-based Metrics

Figure 4.8: Distribution of reward obtained by each agent against all algorithms in each of the sets of game generators.

zero median regret but have higher average regret. Q-learning is sometimes able to achieve negative regret on both sets of generators and fictitious play does so on the 10x10 set of generators. Averaging over all of the opponents, GIGA-WoLF achieves negative average regret in the 2x2 set of games and average regret of \( \approx 0 \) in the 10x10 set of generators.

To confirm that there is a statistical difference between the regret distributions obtained by GIGA-WoLF and Q-learning, we performed a Kolmogorov-Smirnov Z test. Table 4.6 provides the results of this test. In both the 2x2 and 10x10 generator sets there is a statistically significant
### 4.4. Experimental Results: Reward-based Metrics

![Graph](image)

**Figure 4.9:** Distribution of reward obtained by each agent against all algorithms on both sets of game generators.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>GIGA-WoLF</th>
<th>Q-learning</th>
<th>GIGA-WoLF</th>
<th>Q-learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>-0.0004</td>
<td>0.0052</td>
<td>0.0003</td>
<td>0.0263</td>
</tr>
<tr>
<td>median</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>25\text{th} percentile</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>75\text{th} percentile</td>
<td>0</td>
<td>0</td>
<td>0.0001</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 4.6:** Kolmogorov-Smirnov test comparing the distribution of regret obtained by GIGA-WoLF and Q-learning.

The difference between the regret distributions achieved by GIGA-WoLF and Q-learning.

#### 4.4.3 Incentive to Deviate

We now consider agents’ incentive to deviate, which can be understood as their regret with respect to arbitrary nonstationary strategies. In other words, incentive to deviate measures how much extra reward an agent would have obtained had she managed to play a best response action to her opponent’s action in each stage game.

**Claim 9** GIGA-WoLF consistently achieves a lower incentive to deviate score against its opponents.
4.4. Experimental Results: Reward-based Metrics

Figure 4.10: Distribution of regret obtained by each agent against all other agents in the 2x2 (top) and 10x10 (bottom) sets of game generators.

Figure 4.11 displays the incentive to deviate score achieved by each agent against each other agent averaged over the set of 2x2 games. We have sorted the bars from the smallest to largest incentive to deviate obtained across all games (i.e., the averages of the values in Figure 4.11 and Figure 4.12.) Against all agents, GIGA-WoLF achieves the lowest score, with fictitious play always a close second. The graph also shows a fairly regular order of incentive to deviate: GIGA-WoLF, fictitious play, GSA, minimax-Q-IDR, Q-learning and minimax-Q. This order only differs when
4.4. Experimental Results: Reward-based Metrics

**minimax-Q-IDR** is an opponent, in which case Q-learning obtains a lower score than minimax-Q-IDR.

Figure 4.12 displays the results for the 10x10 set of generators. Results are generally the same as in the previous figure, but now the order of the algorithms' scores is the same for all opponents. Q-learning now always achieves a better score than Minimax-Q-IDR, which is the opposite of what occurred in the 2x2 games. Again, GIGA-WoLF consistently achieves the lowest average best response score against all opponents. These results are interesting since incentive to deviate is a kind of regret and GIGA-WoLF is designed to minimize a different form of regret.

### 4.4.4 Minimax-Q and Domination

Although for the most part we cannot make broad claims about how algorithms behave in terms of the reward they attain, we can say more about one algorithm: Minimax-Q.

**Claim 10** Minimax-Q-IDR consistently outperformed minimax-Q in terms of reward.

Minimax-Q-IDR obtained higher median reward than minimax-Q against all opponents in all calls of the 2x2 generator. This was also true in the majority of 10x10 generators (62.50% of the time). Due to this performance difference, minimax-Q will sometimes be dropped from the results in the next section.
The 10x10 games in which minimax-Q obtained higher reward are characterized generally by having a single equilibrium\(^3\). The generators are the supermodular games (Cournot Duopoly, Bertrand Oligopoly, Arms Race), Location game, Traveller’s Dilemma, Covariant game, Dispersion game and War of Attrition. Minimax-Q-IDR assumes that its opponents will not play dominated actions, which of course is not always true. For example, in the Traveller’s Dilemma game both players can achieve rewards by playing dominated strategies than they can by playing non-dominated strategies. This suggests that while minimax-Q-IDR plays the Nash equilibrium strategy, minimax-Q does not. On average across all opponents, this behaviour leads to higher rewards for minimax-Q than for minimax-Q-IDR.

Figures 4.13(a) and 4.13(b) display the reward distribution for minimax-Q and minimax-Q-IDR against all other algorithms in the 2x2 and 10x10 game sets respectively. In both cases, there is a statistically significant difference in the reward distribution, with minimax-Q-IDR obtaining higher reward. The results of running a Kolmogorov-Smirnov statistical test on the distributions are shown in Table 4.7.

**Claim 11** Minimax-Q-IDR dominates GIGA-WoLF in Traveller’s Dilemma when judged by percentage of wins.

---

\(^3\)Note: When Gambit [McKelvey et al., 2004] is used to calculate equilibria, there are cases where a generator instance is found to have two equilibria. One of these is the correct pure strategy equilibrium, while the other is a mixed strategy equilibrium with a very large probability on the action from the pure strategy equilibrium and a very small probability on another action. It is assumed that this is due to numerical inaccuracy between the algorithms calculating the equilibria and the values of the payoff matrix.
4.4. Experimental Results: Reward-based Metrics

Figure 4.13: Reward distribution obtained by minimax-Q and minimax-Q-IDR against all opponents on all instances of each set of generators.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>minimax-Q</th>
<th>minimax-Q-IDR</th>
<th>minimax-Q-IDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>0.1636</td>
<td>0.3161</td>
<td>0.1081</td>
</tr>
<tr>
<td>median</td>
<td>0.1412</td>
<td>0.3705</td>
<td>0.0740</td>
</tr>
<tr>
<td>25th percentile</td>
<td>-0.2509</td>
<td>-0.1471</td>
<td>-0.2653</td>
</tr>
<tr>
<td>75th percentile</td>
<td>0.6784</td>
<td>0.8789</td>
<td>-0.6767</td>
</tr>
</tbody>
</table>

Table 4.7: Results of the Kolmogorov-Smirnov Z test on the reward obtained by minimax-Q and minimax-Q-IDR against all algorithms on the two sets of generators.

Nudelman et al. [2004] compared minimax-Q, Q-learning and one of the earlier variants of WoLF across a distribution of games. They found that for the percentage of wins metric, minimax-Q dominated WoLF in the Traveller's Dilemma generator: that is, that Minimax-Q always achieved the higher reward in interactions between the two algorithms. This was shown for a two-action version of the game.

We were interested in whether this claim would hold true for different sizes of the game and for the updated version of WoLF. We ran a small subtest on 20 instances of a 2x2 and 10x10 version of Traveller's Dilemma. The configuration file for the main experiment was used to ensure consistency among all parameters with the values used in the main experiment. The results showed that the claim that minimax-Q dominates WoLF does not extend to GIGA-WoLF. However, we saw that minimax-Q-IDR did substantially outperform both minimax-Q and GIGA-WoLF in both the 2x2 and 10x10 versions of Traveller’s Dilemma.
4.5. Experimental Results: Convergence to a Nash Equilibrium

We now examine whether algorithms converge to a Nash equilibrium. Note that we cannot ask this question about a single algorithm because the concept of equilibrium depends on both players' strategies.

There has been some debate in the literature about whether it is reasonable to evaluate algorithms in terms of their ability to converge to an equilibrium. For example, Shoham et al. [2003] argue that an unconditional focus on Nash equilibrium is misguided. They suggest that the focus should be...
4.5. Experimental Results: Convergence to a Nash Equilibrium

on meeting an objective (e.g. achieving high reward) given information about the types of agents who inhabit the environment. In this scenario, “the equilibrium concept [is not considered] central or even necessarily relevant at all” [Shoham et al., 2003]. However, the Nash equilibrium retains broad appeal as the “solution” to a game. These are at least “focal” strategies in some sense; if two algorithms are playing Nash equilibrium strategies, both are best responding to each other and neither can obtain a higher payoff by unilaterally changing her strategy.

4.5.1 Linking Reward and Convergence

The first experimental question to ask is whether it turns out that good performance under reward- and equilibrium-based metrics are correlated; if so, the philosophical debate about choosing between the metrics would appear less important.

Claim 12 There is no relationship between obtaining large reward and converging to a Nash equilibrium.

Our experiments do not support the idea that there is a link between the reward that an agent receives and its convergence to a Nash equilibrium. Two agents can be equally far from an equilibrium and not obtain similar rewards. Two different pairs of agents could also converge to different equilibria which have different payoffs.

At a high level, Figures 4.16 and 4.17 display the combined \( \hat{e}_1 \) score for convergence to a Nash equilibrium and the reward obtained by each agent in the set of 2x2 and 10x10 game generators respectively. The results for each agent are aggregated against all opponents. Minimax-Q is not included in these tests. In these figures, we compare the reward and convergence scores between
4.5. Experimental Results: Convergence to a Nash Equilibrium

Figure 4.16: Box plot of the distributions of the reward and $\ell_1$ distance to a Nash equilibrium obtained in the 2x2 set of games. FP is fictitious play.

agents, rather than comparing these scores for the same agent. In the top figure, all convergence box plots have their mass at zero, with the pluses being outliers to this model; this means that the vast majority of data points had a value of zero, contrary to the visual impression given by the graph.

At a high level, Figures 4.16 and 4.17 plot the combined score for convergence to a Nash equilibrium and the reward obtained by each agent. The results for each agent are aggregated against all opponents. Minimax-Q is not included in these tests.

Focusing on the 10x10 plot of Figure 4.17, it can be seen that GSA obtains a higher median reward than Q-learning, but has a worse convergence rate. GIGA-WoLF obtains a similar reward distribution to GSA, but has a very different Nash convergence distribution.

Examining a single generator, we provide the example of Figure 4.18 where we show the rates of convergence to a Nash equilibrium and the reward obtained for Q-learning and fictitious play in self-play (These algorithms are being used as an example and the argument is not specific to self-play.). Both agents converge to an equilibrium, but obtain very different reward distributions.

We did find one connection between reward and convergence to a Nash equilibrium. If algorithm $A$ plays $B$ and $A$ also plays $C$ and in both cases $A$ obtains similar convergence rates, then $A$ should be expected to achieve a similar reward distribution in both cases. An example which follows this rule is shown in Figure 4.19, where the distributions for the reward obtained and distance to the Nash equilibrium are shown for minimax-Q-IDR against fictitious play and GIGA-WoLF. The algorithms play similar strategies leading to the same set of rewards. We performed a Kolmogorov-Smirnov test comparing the resultant Nash and reward distributions, which showed that there was no significant difference between the metric values that minimax-Q-IDR obtained against fictitious play and against GIGA-WoLF ($p \approx 1$ in both cases). The results for the test are given in Table 4.8. Similar examples of this connection can be found for all of the algorithms tested in our experiment.
4.5. Experimental Results: Convergence to a Nash Equilibrium

![Distributions of the reward and $\ell_1$ distance to a Nash equilibrium obtained in the 10x10 set of generators. FP is fictitious play.](image1)

![Distributions of the reward and $\ell_1$ distance to a Nash equilibrium obtained in the Arms race game.](image2)

Interestingly, our experimental evidence does not allow us to generalize the rule described above to say that when algorithm A plays C and algorithm B plays D, if A and B obtain similar Nash distributions, then they should also obtain similar reward distributions.

The results of this experiment suggest that convergence to Nash equilibrium and reward are not necessarily related and one metric cannot be used to judge both. We believe that if possible, future
Experimental Results: Convergence to a Nash Equilibrium

Figure 4.19: minimax-Q-IDR obtains similar distributions of the $\ell_1$ distance to a Nash equilibrium against two opponents. The resulting reward distribution is also similar.

<table>
<thead>
<tr>
<th>Kolmogorov-Smirnov Z test</th>
<th>$\ell_1$ convergence to a Nash equilibrium</th>
<th>Reward obtained</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No statistical difference</td>
<td>No statistical difference</td>
</tr>
<tr>
<td></td>
<td>$p = 1$</td>
<td>$p = 1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Statistic</th>
<th>minimax-Q-IDR vs Fictitious play</th>
<th>minimax-Q-IDR vs GIGA-WoLF</th>
<th>minimax-Q-IDR vs Fictitious play</th>
<th>minimax-Q-IDR vs GIGA-WoLF</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>0.2139</td>
<td>0.2230</td>
<td>0.4135</td>
<td>0.4135</td>
</tr>
<tr>
<td>median</td>
<td>0</td>
<td>0</td>
<td>0.6897</td>
<td>0.6897</td>
</tr>
<tr>
<td>Std. Deviation</td>
<td>0.57431</td>
<td>0.57791</td>
<td>0.64099</td>
<td>0.64088</td>
</tr>
<tr>
<td>25th percentile</td>
<td>0</td>
<td>0</td>
<td>-0.1063</td>
<td>-0.1063</td>
</tr>
<tr>
<td>75th percentile</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.8: Results of the Kolmogorov-Smirnov Z test between the reward and Nash convergence results of minimax-Q-IDR against fictitious play and minimax-Q-IDR against GIGA-WoLF, as shown in Figure 4.19

4.5.2 Exact convergence to a Nash equilibrium

We now investigate the extent to which algorithms converge exactly (within machine precision) to Nash equilibria.

Claim 13 Algorithms often converge to Nash equilibria, but also often fail to converge.
4.5. Experimental Results: Convergence to a Nash Equilibrium

Results are shown for the percentage of interactions in which an algorithm and its opponent converge to a Nash equilibrium. In the 2x2 set of games, the majority of algorithms converge more than 50% of the time. However, in the 10x10 set of generators all algorithms converge in less than 50% of the interactions. In the literature, convergence to an equilibrium strategy is normally shown for self-play on a single game. The evidence for the above claim is presented for each algorithm against six opponents across the sets of 2x2 and 10x10 generators.

Figure 4.20 displays the probability of each of the algorithms converging with their opponent to a Nash equilibrium. The algorithms are sorted according to decreasing overall probability of convergence averaged across both sets of games.

Fictitious play has the highest success in both sets of games, converging more than two thirds of the time in the 2x2 set. Surprisingly, it is followed by minimax-Q-IDR, perhaps an indication of how often the 2x2 game can be iteratively reduced down to a Nash equilibrium. Equally surprisingly is the fact that GIGA-WoLF is not as successful and is in fact beaten by Q-learning in both generators. Q-learning regularly converges, despite the facts that it always plays a pure strategy and ignores its opponent's ability to adapt. GSA almost never converges, possibly because of its proclivity to jump out of local minima.

For all of the algorithms, there is a higher percentage of convergence in the 2x2 set of games, compared to the 10x10 set of generators. One possible reason for this is that pure strategy equilibria are more prevalent in the 2x2 games.
4.5. Experimental Results: Convergence to a Nash Equilibrium

4.5.2.1 Joint $\ell_1$ measure of convergence to a Nash equilibrium

We can now ask the question whether algorithms converge to a strategy which is close to a Nash equilibrium, whether or not they converge exactly. To answer this question, we use the $\ell_1$ measure of convergence to a Nash equilibrium and ask how often the algorithms converged to strategies which were less than 0.005 from a Nash equilibrium.

Claim 14 Algorithms often converge to strategies which are close to a Nash equilibrium.

Figure 4.21 displays the percentage of repeated games in which the average joint $\ell_1$ distance to the closest Nash equilibrium is less than 0.005. The algorithms are in the same order as Figure 4.20. The results show that in the 2x2 and 10x10 sets every algorithm has a substantially higher percentage chance of being "near" to an equilibrium than being exactly at the equilibrium. The percentage increase in the 2x2 set of generators from the percentages shown for exact convergence in Figure 4.20 is a minimum of 12.5%, a maximum of 74.05% and a median of 15.89%. In the 10x10 set there is a minimum of 6.6%, a maximum of 32.85% and a median of 9.5% increase in the probabilities. The biggest increase is for the GSA algorithm, where the values increase by 74% and 33% in the 2x2 and 10x10 set of generators respectively. This supports our claim above that the extra "jump" in the GSA algorithm causes it to get close to the actual Nash strategy, but often not quite reach it.

This indicates that in a large number of cases, algorithms almost converge to the exact equilibrium, but do not quite reach it. In the $\ell_1$ data, the algorithms are still more likely to converge in the 2x2 set than the 10x10 set of generators. In the 2x2 set of games, Q-learning gets close to an equilibrium marginally more often than fictitious play. In the 10x10 set of generators, fictitious play now reaches the window in more than 50% of the cases.
4.5. Experimental Results: Convergence to a Nash Equilibrium

The most important point from this evidence is that all of the algorithms converge to a near-equilibrium strategy in higher percentages than for exact convergence. Any test for convergence should consider reporting convergence to strategies which are near equilibria as well as to exact equilibria.

4.5.3 Convergence in self play

We now differentiate between self and non-self play to see if an algorithm is more likely to converge in either of those cases. Convergence to a Nash equilibrium in self play is arguably more important than convergence to a Nash equilibrium against an arbitrary opponent, for two reasons. First, we pointed out above that convergence to a Nash equilibrium is a metric that measures both algorithms playing the game; however, in self play this metric really does describe only a single algorithm. Second, if an algorithm is successful then it should expect to encounter itself in self play; therefore, its self-play behaviour should be understood.

Claim 15 Algorithms converge more often to an exact Nash equilibrium in self play than in non-self play.

The majority of theoretical results for convergence in the literature are for self play. We are not aware of previous empirical comparisons between self and non-self play convergence rates. Figure 4.22 shows the percentage of repeated games in which an algorithm converged exactly to a Nash equilibrium. The results are shown for self and non-self play in the two generator sets and are sorted according to the self play probabilities averaged across all games.
4.5. Experimental Results: Convergence to a Nash Equilibrium

For all algorithms except GSA, the convergence probabilities drop when moving from self to non-self play; GSA almost never converges in either self or non-self play. Interestingly the difference in percentages for minimax-Q between self play (56.58%) and non-self play (52.30%) is quite small. Q-learning, which ignores what the opponent is doing, is very successful. Since Q-learning only plays pure strategies, this indicates that many of our 2x2 games have pure strategy equilibria.

The values also indicate how much the convergence probability is affected by the opponent. In the 2x2 set of generators, there is more than a 30% difference for GIGA-WoLF in convergence probability between self and non-self play. Fictitious play and Q-learning both exhibit differences greater than 20%. While viewing an algorithm's ability to converge in self play is important, it is not representative of overall convergence. The results suggest that testing for convergence in self play provides an optimistic convergence rate.

Claim 16 The difference in convergence rates to strategies close to a Nash equilibria between self and non-self play is smaller than the difference in convergence rates for self and non-self play for exact convergence rates.

Figure 4.23 presents the rates for algorithms converging to an epsilon window around the equilibrium in self and non-self play in the 2x2 and 10x10 generator sets. In the 2x2 set, all algorithms except for minimax-Q have a difference of less than 5% between the self play and non-self play convergence rates. This is different to the results displayed in Figure 4.22, where minimax-Q was the only algorithm with a small difference between self and non-self play convergence rates for the 2x2 set of games.

GSA and GIGA-WoLF are the only algorithms that have any difference in self play percentages between convergence to the exact equilibrium and convergence to a strategy close to an equilib-
4.5.4 Convergence in Cooperative Games

Claus and Boutilier [1997] made the claim that fictitious play converges to the Nash equilibrium in a cooperative game, but this claim was not evaluated empirically. We now look at whether this is true.

**Claim 17** Algorithms converge to a Nash equilibrium in self play in cooperative games.

One of the games used in our experiment is the Minimum effort game, which is a cooperative game. The results for exact convergence to a Nash equilibrium are shown in Figure 4.24. This is a plot of the joint $\ell_1$ distance to the Nash equilibrium for all agents against all opponents. All of the algorithms, including minimax-Q, converge in the median to the Nash equilibrium. This supports the claim of Claus and Boutilier [1997] and also shows convergence in both self and non-self play.

4.6 Experimental Results: Estimating an Opponent's Strategy

Fictitious play, GIGA-WoLF and GSA all estimate their opponent's strategy based on the history of actions played. We now look at whether different estimation techniques have an effect on the
4.6. Experimental Results: Estimating an Opponent’s Strategy

Discounted averaging!

(a) Estimation distribution in the set of 2x2 games. (b) Estimation distribution in the set of 10x10 generators.

Figure 4.25: Distributions of the $\ell_1$ distance between the actual and estimated strategy in the sets of game generators.

performance of an algorithm.

To isolate the estimation technique, we ran a test using two versions of fictitious play. The first version used the standard undiscounted count-based technique to estimate the opponent’s strategy, while the second version used the discounted averaging technique. We used sixty instances of the TwoByTwo generator and twenty instances of three 10x10 generators, Traveller’s dilemma, Location game, and the Dispersion game. The configuration file for the original experiment was used to ensure that parameter values were consistent with those used in the main experiment. This included running the test for 100,000 iterations, with the first 90,000 iterations allowing parameters to settle and results being recorded on the last 10,000.

Claim 18 The discounted-average and the average estimation techniques return statistically different estimates of an opponent’s strategy.

Using the $\ell_1$ estimate of the opponent’s distribution, we show that the two estimation techniques return statistically different estimates, but neither performs consistently better. The discounted-average returns the more accurate estimate for the 2x2 set of games, while the count-based averaging technique returns the more accurate estimate for the 10x10 set of generators.

Figures 4.25(a) and 4.25(b) show the distribution of the $\ell_1$ norm for each technique in estimating its opponent’s strategy, in the 2x2 and 10x10 sets respectively. The algorithms are identified through the name of the estimation technique.

We then ran Kolmogorov-Smirnov Z tests on the distributions to determine if they were statistically different. The results from applying this test to the 2x2 and 10x10 set of generators are given in Table 4.9. The results show significant differences between the obtained distributions in both generator sets. Discounted averaging is more accurate in the 2x2 set of games, while averaging is more accurate in the 10x10 set of generators.

Claim 19 The two different estimation techniques do not obtain reward that is significantly different.
4.6. Experimental Results: Estimating an Opponent’s Strategy

Kolmogorov-Smirnov Z test

<table>
<thead>
<tr>
<th></th>
<th>2x2 set</th>
<th>10x10 set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Results are statistically different</td>
<td>Results are statistically different</td>
<td></td>
</tr>
<tr>
<td><em>p</em> = 0.005</td>
<td><em>p</em> ≈ 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Averaging</th>
<th>Discounted Averaging</th>
<th>Averaging</th>
<th>Discounted Averaging</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>0.5416</td>
<td>0.5</td>
<td>0.8686</td>
<td>0.9</td>
</tr>
<tr>
<td>median</td>
<td>0.9999</td>
<td>0.5</td>
<td>0.8058</td>
<td>0.9</td>
</tr>
<tr>
<td>25&lt;sup&gt;th&lt;/sup&gt; percentile</td>
<td>0</td>
<td>0</td>
<td>0.0001</td>
<td>0</td>
</tr>
<tr>
<td>75&lt;sup&gt;th&lt;/sup&gt; percentile</td>
<td>1</td>
<td>0</td>
<td>1.8</td>
<td>1.8</td>
</tr>
</tbody>
</table>

Table 4.9: Results of the statistical comparison between the \( \ell_1 \) distances scored by each of the estimation techniques.

(a) Reward distribution for the 2x2 games

(b) Reward distribution for the 10x10 generators

Figure 4.26: Distribution of reward that the fictitious play algorithms with different estimation techniques receive. Results for the sets of 2x2 and 10x10 generators are shown separately.

The evidence for the previous claim (Claim 18) showed that there is a statistical difference between the strategy estimates that the two different estimation techniques obtain. However, we now show that these different strategy estimates do not statistically affect the reward received by the different versions of fictitious play.

Figures 4.26(a) and 4.26(b) show the reward distribution that the two fictitious play algorithms obtained. The algorithms are identified by the name of the estimation technique that the algorithm used.

Table 4.10 provides the results of the Kolmogorov-Smirnov Z test, testing for similarity between the different reward distributions obtained by the two estimation techniques. In both sets of games, the reward distribution obtained by the two algorithms is not statistically different with 95% confidence. However, the results of the 10x10 generator with *p* = 0.071 are not far from statistical difference.

These two claims show that although the two estimation techniques produce statistically differ-
4.7 Directions for Future Study

We end this section with some questions that were raised by our work and that could serve as interesting directions for future study:

1. **What sorts of algorithms will dominate Q-learning?**

   Q-learning proved to be surprisingly resilient for an algorithm that ignores the existence of other agents in the environment. Why was it able to obtain such impressive results? It would be useful to characterize the algorithms that dominate Q-learning.

2. **Could other algorithms be improved by iteratively removing dominated actions and then applying the algorithm's strategy to the remaining payoff space?**

   Minimax-Q-IDR produced better results over minimax-Q by following this strategy. Would this technique be useful to other algorithms?

3. **Why were our results on our set of 10x10 games qualitatively different from our results on our set of 2x2 games?**

   We saw that there was no clear relationship between reward performance and the number of actions in the game for any of the agents. However, we obtained consistently different results for the two generator sets. (For example, Figure 4.20 on page 62 showed that algorithms converge to an equilibrium less often in the set of generators with ten actions.) Was this due to different game structures, were the results random, or is there another explanation?

4. **Are other algorithms affected by different estimation techniques?**
Section 4.6 showed that the two different estimation techniques produced statistically different estimates of an opponent’s strategy. This did not cause a statistically different amount of reward to be received by the two fictitious play algorithms. Is this also true for other algorithms which utilize the estimate of the opponent’s strategy differently?
Chapter 5

Conclusion

The purpose of this dissertation was to design a platform to test multiagent reinforcement learning algorithms in a repeated game setting and to visualize the results of such testing. Indeed, this dissertation has described such a platform and the results of an experimental investigation conducted with this platform.

Chapter 1 provided motivation for this research. Chapter 2 described background work. It was found that there was very little consistency in the metrics, game generators or experimental setup across different papers and this made it difficult to compare results between various papers.

Chapter 3 described the architecture of the platform. This included the algorithms, metrics and visualization routines that have been implemented. An important aspect of the platform is the ability to quickly visualize statistics at the end of an experiment. This process was outlined in detail and illustrated through figures showing the various graphical user interfaces that make up the platform.

Chapter 4 presented the experimental results, describing the empirical test and resulting analysis. The games were split into two sets of generators, two-action and ten-action games. We gave a set of nine high level observations on our work:

1. No algorithm dominates.
2. An algorithm needs to be tested on a variety of generators in order to obtain accurate performance information.
3. There is no relationship between algorithm performance and the number of actions in a game and thus it is important to test on games with different numbers of actions.
4. Having information about the games and opponents being played can provide insight into which algorithm to use in that environment.
5. The choice of which algorithm to use in an environment should also depend on the paradigm being used to evaluate the algorithms.
6. If convergence to a Nash equilibrium is important, then a good algorithm to use is fictitious play.
7. If reward is important, then GSA or Q-learning are good options.
8. Iterated removal of dominated actions may benefit algorithms other than minimax-Q.
9. Large experiments were easier to conduct using our platform.

The results from our experiment were given in the form of 19 claims. Each claim was justified with empirical evidence. We have argued that:
1. The iterations dimension can be reduced by averaging metric values across iterations.
2. The instances dimension can be reduced by averaging metric values across instances.
3. Generators from the same game theoretic group do not exhibit consistently similar properties according to any of the metrics we measured.
4. No two algorithms are similar in performance against all opponents on all games.
5. No algorithm obtains the highest average reward against every opponent in either the 2x2 or 10x10 sets of generators.
6. Q-learning achieves the highest mean and median reward against all opponents in the 2x2 set of generators.
7. Fictitious play obtains the highest median and mean reward in the 10x10 set of generators.
8. GIGA-WoLF achieves lower average regret than other algorithms and often achieves negative regret.
9. GIGA-WoLF consistently achieves a lower incentive to deviate score against its opponents than other algorithms.
10. Minimax-Q-IDR consistently outperformed minimax-Q in terms of reward.
12. There is no relationship between obtaining large reward and converging to a Nash equilibrium.
13. Algorithms often converge to Nash equilibria, but also often fail to converge.
14. Algorithms often converge to strategies which are close to a Nash equilibrium.
15. Algorithms converge more often to an exact Nash equilibrium in self play than in non-self play.
16. The difference in convergence rates to strategies close to a Nash equilibria between self and non-self play is smaller than the difference in convergence rates for self and non-self play for exact convergence rates.
17. Algorithms converge to a Nash equilibrium in self play in cooperative games.

18. The discounted-average and the average estimation techniques return statistically different estimates of an opponent's strategy.

19. The two different estimation techniques do not obtain reward that is significantly different.

The platform proved to be extremely helpful in running and analyzing experiments. The process for exploring and evaluating hypotheses using the experimental results worked well and allowed simpler analysis. The testing platform allowed results to be easily generated and visualized. The list of claims included some surprising results and confirmed a number of claims from the literature. The platform helped to gain a better understanding of algorithms and aspects of multiagent learning. There are however many questions still to be answered and this work is only a first step.
Bibliography


