# Just Add Pepper: Extending Learning Algorithms for Repeated Matrix Games to Repeated Markov Games

Jacob W. Crandall
Computing and Information Science Program
Masdar Institute of Science and Technology
Abu Dhabi, United Arab Emirates
jcrandall@masdar.ac.ae

## ABSTRACT

Learning in multi-agent settings has recently garnered much interest, the result of which has been the development of somewhat effective multi-agent learning (MAL) algorithms for repeated normal-form games. However, general-purpose MAL algorithms for richer environments, such as general-sum repeated stochastic (Markov) games (RSGs), are less advanced. Indeed, previously created MAL algorithms for RSGs are typically successful only when the behavior of associates meets specific game theoretic assumptions and when the game is of a particular class (such as zero-sum games). In this paper, we present a new algorithm, called Pepper, that can be used to extend MAL algorithms designed for repeated normal-form games to RSGs. We demonstrate that Pepper creates a family of new algorithms, each of whose asymptotic performance in RSGs is reminiscent of its asymptotic performance in related repeated normal-form games. We also show that some algorithms formed with Pepper outperform existing algorithms in an interesting RSG.

## Categories and Subject Descriptors

I.2.6 [**Computing Methodologies**]: Artificial Intelligence-Learning

## General Terms

Algorithms

## Keywords

Multi-agent learning, stochastic games, game theory

## 1. INTRODUCTION

Much research in multi-agent learning (MAL) continues to focus on learning in repeated normal-form (or *matrix*) games. This research has resulted in many *matrix learning algorithms* (MLAs – algorithms for learning repeated matrix games), some of which are able to learn effectively in general-sum matrix games. For example, M-Qubed has been shown to perform robustly in several empirical studies involving many different MLAs in many repeated matrix games [4, 10]. Exp3 [2], GIGA-WoLF [6], and UCB [1] have also performed well in certain games in these studies.

However, many situations in which MAL is necessary are better modeled as repeated stochastic games (RSGs) than repeated matrix games. In RSGs, rewards are received incrementally. This requires a learning agent to solve two problems simultaneously. First, it must determine its delayed rewards, or the rewards it will receive in future states of the world. These delayed rewards are contingent on the strategies of the agent and its associates in future states. Second, the agent (and its associates) must learn an effective strategy in each state. But learning such strategies requires accurate estimates of delayed rewards.

Whereas these chicken and egg problems can be solved simultaneously in single-agent domains via standard reinforcement learning, they are not so easily solved in multi-agent settings since an agent cannot easily predict or control its associates' current and future strategies. As a result, many MAL algorithms have been investigated for RSGs. These algorithms typically assume that associates will conform to a particular game-theoretic behavior, and that the corresponding game theoretic solution concept will define an effective strategy for the agent. While sometimes effective, such algorithms often do not perform well when these assumptions fail. For example, algorithms that seek to learn a Nash equilibrium (NE) sometimes perform well in competitive stochastic games in which the NE corresponds to the minimax strategy (e.g., [17]), but often fail in repeated general-sum stochastic games when there is not a unique NE [5] (see also the folk theorem [12]). Additionally, many algorithms require perfect knowledge of associates' payoffs, which are often unavailable.

In this paper, we introduce a new algorithm, called *Pepper*, for learning in RSGs for situations in which the actions of associates are observable, but where the payoffs of associates and (initially) the state transitions of the game are unknown. This new algorithm uses a separate instance of a MLA to learn a strategy in each unique state of the game using payoffs (future rewards) computed by Pepper. To demonstrate the effectiveness of Pepper, we use it to derive four new algorithms for RSGs. We then evaluate these algorithms in an interesting RSG played against a variety of learning algorithms and hand-coded (static) strategies.

## 2. BACKGROUND AND AN EXAMPLE

We begin by defining important terms and notation. We then review the performance of existing algorithms in an example RSG.

**Appears in:** *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, Conitzer, Winikoff, Padgham, and van der Hoek (eds.), 4-8 June 2012, Valencia, Spain.

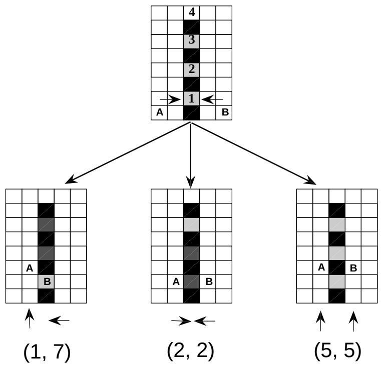

**Figure 1: The SGPD game.**

## 2.1 Definitions and Notation

We consider repeated stochastic games (RSGs) played by players[1] (or agents) $i$ and $-i$. An RSG consists of a set of *stage games* $S$, also known as world states. In each stage $s \in S$, both players choose an action from a finite set. Let $A(s) = A_i(s) \times A_{-i}(s)$ be the set of joint actions available in $s$, where $A_i(s)$ and $A_{-i}(s)$ are the action sets of players $i$ and $-i$, respectively. Each *episode* of the game begins in some beginning stage $s_b \in B \subseteq S$ and terminates when some goal stage $s_g \in G \subseteq S$ is reached. Once a goal stage is encountered, a new episode begins in some stage $s_b \in B$.

When joint action $\mathbf{a} = (a_i, a_{-i})$ is played in stage $s$, player $i$ receives a finite reward $r_i(s, \mathbf{a})$. Here, $a_i \in A_i(s)$ and $a_{-i} \in A_{-i}(s)$ are the actions of players $i$ and $-i$, respectively. Once the joint action $\mathbf{a}$ is played in $s$, the world transitions to some new stage game $s'$ according to the probabilistic stage transition model $P_M(s, s', \mathbf{a})$.

We assume that the transition model $P_M$ is unknown to the players initially. However, we assume that the players can learn the transition model from experience by observing joint actions and stage transitions. Additionally, we assume that players can observe their own immediate rewards, but not those of associates. We also assume that player $i$'s maximum possible reward $R_i^{max}$ for an episode is known *a priori*. This assumption can be replaced by a process through which $R_i^{max}$ is learned, but this is not essential to our discussion.

## 2.2 Motivating Example

Figure 1 depicts a stochastic game prisoner's dilemma (SGPD) [13]. In this game, two players, labeled `A` and `B`, begin each episode in opposite corners of the world. The goal of the game is to enter one of the gates (labeled 1, 2, 3, and 4 in the figure) in as few moves as possible. If both agents try to enter gate 1 at the same time, then gates 1 and 2 close and the agents must enter gate 3. If only one agent enters gate 1, gates 1–3 close and the other agent must enter gate 4. If either agent enters gate 2, then gate 1 closes. If both agents try to enter gate 2 they are both allowed passage. An episode ends once both agents have entered a gate.

The set of stages of the SGPD is defined by the positions of the players and the status of the four gates. Each configuration of these elements is a unique stage game in which both players' can choose to move *up*, *down*, or *toward the gates* (*right* for player `A`, and *left* for player `B`). Moves into a wall (boundary, black space, or closed gate) result in no movement. Each player receives 10 points for entering any gate, and is penalized 1 point for each move it takes.

[1]For ease of exposition, we assume the game has two players; Pepper can easily be extended to $(n > 2)$-player games.

**Table 1: High-level payoff matrix of the SGPD.**

| | **Defect** (Gate 1) | **Cooperate** (Gates 2–4) |
|---|---|---|
| **Defect** (Gate 1) | 2, 2 | 7, 1 |
| **Cooperate** (Gates 2–4) | 1, 7 | 5, 5 |

**Table 2: Performance in the SGPD. Bold indicates nearly ideal performance. Algorithms marked *VI* use model-based, rather than model free, methods.**

| Algorithm | Associate | | | | |
|---|---|---|---|---|---|
| | Self Play | Defector | Cooperator | Random | TFT |
| **Q-learning** | 2.2 | 1.3 | **7.0** | **4.3** | 3.4 |
| **WoLF-PHC** | 1.9 | 1.2 | **6.9** | **4.1** | 2.1 |
| **Minimax-VI** | 2.0 | **1.9** | **7.0** | **4.4** | 2.0 |
| **Friend-VI** | 1.9 | -0.9 | 3.0 | 1.0 | 2.0 |
| **Nash-VI*** | 2 | **2** | **7** | **4.5** | 2 |
| **uCE-VI*** | **5** | 1 | 5 | 3 | **5** |
| **FolkEgal*†** | **5** | **2** | 5 | 3.75 | **5** |

** Requires knowledge of associate's payoffs; performance estimated
† Requires knowledge of stage transition probabilities $P_M(s, s', \mathbf{a})$

A player that tries to enter gate 1 is said to have *defected.* Otherwise, the player is said to have *cooperated.* Thus, the *high-level* game is the prisoner's dilemma (PD) matrix game shown in Table 1; each cell lists the sum of rewards received in an episode by the row and column players, respectively.

Table 2 shows the average asymptotic per episode payoffs of existing algorithms in the SGPD in self play and when associating with four static associates. Defector always defects, Cooperator always cooperates, Random chooses a gate randomly, and TFT chooses a gate according to the tit-for-tat strategy. Bold values indicate (nearly) ideal performance. The payoffs of Q-learning [24], WoLF-PHC [7], Minimax-VI [19], and Friend-VI [20] are based on 200,000-episode games run using the parameter settings given in Table 5. The payoffs of Nash-VI [16], utilitarian correlated reinforcement learning (uCE-VI) [14], and FolkEgal [9] were deduced from the descriptions of the algorithms.

Table 2 shows that none of the algorithms performs ideally against this limited set of algorithms. Only uCE and FolkEgal cooperate in self play, which results in an expected per-episode payoff of 5. However, uCE and FolkEgal do not behave ideally against Cooperator or Random. Additionally, they must know their associate's payoffs.

We seek to identify MAL algorithms that learn effectively against many kinds of associates in the SGPD and other general-sum RSGs when associate's payoffs are unknown. Since several matrix learning algorithms (MLAs) have, to various degrees, achieved these objectives in matrix games (such as Table 1), we explore the extension of these algorithms to RSGs using Pepper.

## 3. PEPPER

Pepper (potential exploration with pseudo stationary restarts) is defined by three design choices. First, Pepper utilizes the R-max [8] algorithmic framework, which incorporates the *optimism-in-uncertainty* principle. Second, Pepper defines a new mechanism for estimating future rewards, which are used to estimate a payoff matrix for each stage of the RSG. This mechanism is also designed to adhere to the optimism-in-uncertainty principle, while eventually reflecting the agent's actual rewards in an episode. Third, Pepper uses a separate instance of an MLA in each stage of the game to learn the agent's strategy in that stage. The MLA employed in stage $s \in S$ learns the agent's policy in $s$ as it would learn a policy in a repeated matrix game, only it learns from a payoff matrix defined by Pepper.

## 3.1 Algorithmic Framework

To determine how to act in stage $s$, player $i$ must determine how its actions will affect it rewards in the remainder of the episode (i.e., its future rewards). To do this, Pepper computes the payoff matrix $R_i(s)$, which estimates player $i$'s future rewards for each joint action $\mathbf{a} \in A(s)$. Let $R_i(s, \mathbf{a})$ denote the expected future rewards obtained once the joint action $\mathbf{a}$ is played in stage $s$. Also, let $s_k \in S$ be the $k^{\text{th}}$ stage visited in an episode in which $T$ stages are visited. Then, $R_i(s, \mathbf{a})$ is given by

$$R_i(s_k, \mathbf{a}) = \sum_{\tau=k}^{T} r_i^t(s_\tau, \mathbf{a}_\tau), \quad (1)$$

where $\mathbf{a}_\tau$ is the joint action taken in stage $s_\tau$. Bellman [3] showed that $R_i(s_k, \mathbf{a})$ can be equivalently expressed as

$$R_i(s_k, \mathbf{a}) = r_i(s_j, \mathbf{a}) + \sum_{s' \in S} P_M(s_k, s', \mathbf{a})\ V_i(s'), \quad (2)$$

where $V_i(s')$ is the expected future rewards for being in stage $s'$. Thus, given $r_i(s, \mathbf{a})$, $P_M(s, s', \mathbf{a})$, and $V_i(s')$, $R_i(s, \mathbf{a})$, for each $s \in S$, can be computed using value iteration.

When $r_i(s, \mathbf{a})$, $P_M(s, s', \mathbf{a})$ and $V_i(s')$ are unknown, an agent must learn them via exploration. In constant-sum RSGs, the R-max algorithm [8] does this using the optimism-in-uncertainty principle. R-max initially assumes that each joint action from each stage results in maximal reward. It removes this assumption once the joint action has been attempted $K$ (predetermined) times. This draws the agent toward stages that have not been adequately explored.

Algorithm 1 embodies principles of the R-max algorithm for RSGs minus rules specifying how $\pi_i(s)$ or $V_i(s)$ are computed. This algorithmic framework can be used to implement many MAL algorithms, including R-max and the algorithms listed in Table 3, each of which computes $\pi_i(s)$ and $V_i(s)$ differently. The algorithms in Table 3 each assume that all players will collectively conform to some game theoretic behavior, such as minimax (Minimax-Q), NE (Nash-Q), correlated equilibria (CE-Q), or the Nash bargaining solution (NBS-Q) [21]. When appropriate assumptions are met, $R_i(s, \mathbf{a})$ will converge to its "true" value, as it does in single-agent reinforcement learning. However, when these assumptions are not met as is often the case, these algorithms often achieve low payoffs.

As an alternative, Pepper proposes a method for creating a new family of algorithms for RSGs. This method also utilizes Algorithm 1, but it proposes a new mechanism for how $\pi_i(s)$ and $V_i(s)$ should be computed.

## 3.2 Computing a Strategy $(\pi_i(s))$

There exist MLAs that define strategy selection rules that are successful in many repeated matrix games played with many different associates. Pepper seeks to leverage these learning rules by extending MLAs to RSGs. In particular, Pepper uses an MLA to learn a strategy $\pi_i(s)$ in each stage $s \in S$. The MLA used in stage $s$ seeks to learn a strategy that, given the strategy played by its associate in $s$, maximizes the payoffs defined by the payoff matrix $R_i(s)$.

Given that the MLA in each stage $s$ seeks to maximize the agent's payoffs as defined by $R_i(s)$, the ability of the MLA employed in stage $s \in S$ to learn a successful strategy is contingent on accurate estimates of $R_i(s)$. Accurately modeling $R_i(s)$ depends, in turn, on effective estimates of future reward, which are encoded by $V_i(s')$.

**Algorithm 1** Algorithmic framework

```
Input:
  Let $S' = \{s_0, S\}$ be a set of stage games, where $s_0 \in G$
  Let $R_i^{max}$ be player $i$'s maximum reward for an episode
Initialize:
  $\forall s \in S', \mathbf{a} \in A(s), r_i(s, \mathbf{a}) = 0$, $P_M(s, s_0, \mathbf{a}) = 1$, $\kappa(s, \mathbf{a}) \leftarrow 0$
  $\forall s \in S', V_i(s) = R_i^{max}$ and $\pi_i(s) =$ random
  $t \leftarrow 1$
repeat
  $\forall s \in S, \mathbf{a} \in A(s)$, update $R_i(s, \mathbf{a})$ by value iteration; Eq. (2)
  $\forall s \in S$, update $\pi_i(s)$ and $V_i(s)$
  Observe starting state $s \in B$
  repeat
    Select action $a_i$ according to $\pi_i(s)$ and execute
    Observe $\mathbf{a}^t = (a_i, a_{-i})$, $s'$, and $r_i$
    $\kappa(s, \mathbf{a}) \leftarrow \kappa(s, \mathbf{a}) + 1$
    if $\kappa(s, \mathbf{a}) \geq K$ then
      Update $r_i(s, \mathbf{a})$ according to observations
      Update $V_i(s)$
      Update $P_M(s, \cdot, \mathbf{a})$ according to observed frequencies
      Update $R_i(s, \mathbf{a})$ according to Eq. (2)
      Update $\pi_i(s)$
    end if
    $s \leftarrow s'$
  until $s \in G$
until Game Over
```

**Table 3: Algorithms generalized by Algorithm 1.**

| Algorithm | Computing $\pi(s)$ | Computing $V_i(s')$ |
|---|---|---|
| Minimax-Q | maximin strat. of $R_i(s)$ | maximin value of $R_i(s)$ |
| Nash-Q | A NE of $R_i(s)$ | value of a NE of $R(s)$ |
| Friend-Q | $\arg\max_{\mathbf{a} \in A(s)} R_i(s, \mathbf{a})$ | $\max_{\mathbf{a} \in A(s)} R_i(s, \mathbf{a})$ |
| CE-Q | A CE of $R_i(s)$ | value of a CE of $R(s)$ |
| NBS-Q | NBS of $R(s)$ | NBS value of $R(s)$ |

## 3.3 Determining Future Rewards $(V_i(s))$

We advocate that estimates of $V_i(s)$ should satisfy two objectives, or properties:

**Realism Property:** $V_i(s)$ must eventually reflect the actual payoffs received by the agent in an episode after stage $s$ is reached. That is, $\lim_{t \to \infty} V_i^t(s) = \bar{V}_i(s)$, where $V_i^t(s)$ is the estimate of the expected rewards received in episode $t$ after stage $s$ is reached, and $\bar{V}_i(s)$ is the actual expected sum of rewards received in an episode after $s$ is reached.

**Optimism Property:** $V_i(s)$ should overestimate, rather than underestimate, the agent's future reward while learning. That is, for all $t$, $V_i^t(s) \geq \bar{V}_i(s)$.

The realism property ensures that the MLA employed in stage $s$ eventually learns from true expected payoffs, while the optimism property, similar to an *admissible heuristic* in search, helps the agent to avoid learning strategies that lead to premature convergence to local (but not global) maxima.

Pepper seeks to satisfy these two properties by combining off-policy and on-policy methods for estimating $V_i(s)$. Off-policy methods estimate $V_i(s)$ using some ideal (often derived from $R_i(s)$) that the agent hopes to eventually reach. This ideal is specific to the MLA that is used. For example, each algorithm in Table 3 employs a different off-policy method for estimating $V_i(s)$. Let $V_i^{\text{off}}(s)$ denote the estimate of $V_i(s)$ computed from the designated off-policy method.

Alternately, on-policy methods estimate $V_i(s)$ from the actual distribution over joint actions induced by the players' joint strategy. Let $V_i^{\text{on}}(s)$ denote this estimate. Formally,

$$V_i^{\text{on}}(s) = \sum_{\mathbf{a}=(a_i,a_{-i})\in A(s)} \pi_i(s,a_i)\ \pi_{-i}(s,a_{-i})\ R_i(s,\mathbf{a}), \quad (3)$$

where $\pi_i(s,a_i)$ and $\pi_{-i}(s,a_{-i})$ are the probabilities that players $i$ and $-i$ play actions $a_i$ and $a_{-i}$, respectively, in stage $s$. However, since player $i$ does not know $\pi_{-i}(s)$ and since its own strategy varies over time, $V_i^{\text{on}}(s)$ can be defined in terms of the observed distribution of joint actions. Let $\kappa(s,\mathbf{a})$ be the number of times that joint action $\mathbf{a}$ has been played in stage $s$, and let $\kappa^t(s)$ be the number of times that stage $s$ has been visited. Then,

$$V_i^{\text{on}}(s) = \sum_{\mathbf{a}\in A(s)} \frac{\kappa(s,\mathbf{a})}{\kappa(s)}\ R_i(s,\mathbf{a}). \quad (4)$$

In practice, $\frac{\kappa(s,\mathbf{a})}{\kappa(s)}$ can be replaced with a probability that places higher weight on more recent observations.

$V_i^{\text{on}}(s)$, as computed in Eq. (4), satisfies the realism property when $R_i(s,\mathbf{a})$ approaches true expected payoffs for each joint action $\mathbf{a}\in A(s)$. Given that $R_i(s)$ converges with high probability when each joint action in each stage is played sufficiently [8], $V_i(s)$ will converge to the average rewards received from stage $s$, since joint actions not played sufficiently will not substantially impact it.

However, $V_i^{\text{on}}(s)$ may not satisfy the optimism property. Rather, an agent hoping to compute $V_i(s)$ to satisfy the optimism property could estimate $V_i(s)$ as the maximum of $V_i^{\text{on}}(s)$ and $V_i^{\text{off}}(s)$, particularly since both $V_i^{\text{on}}(s)$ and $V_i^{\text{off}}(s)$ are based on initially optimistic assessments of $R_i(s)$. We denote $\hat{V}_i(s)$ as this optimistic assessment, given by

$$\hat{V}_i(s) = \max\left(V_i^{\text{off}}(s), V_i^{\text{on}}(s)\right). \quad (5)$$

Thus, $\hat{V}_i(s)$ has the best chance of satisfying the optimism property, and $V_i^{\text{on}}(s)$ satisfies the realism property. Pepper seeks to obtain the best of both worlds by computing $V_i(s)$ as a convex combination of $\hat{V}_i(s)$ and $V_i^{\text{on}}(s)$. That is,

$$V_i(s) = \lambda_i(s)\ \hat{V}_i(s) + (1-\lambda_i(s))\ V_i^{\text{on}}(s) \quad (6)$$

where $\lambda_i(s)\in[0,1]$ is set to one initially, but approaches zero as the agent obtains more experience in stage $s$. However, it is not clear how quickly $\lambda_i(s)$ should be decreased. If $\lambda_i(s)$ decreases too quickly, then Eq. (6) is unlikely to satisfy the optimism property. On the other hand, if $\lambda_i(s)$ decreases too slowly, then the algorithm will learn too slowly.

In attempt to avoid either extreme, Pepper regulates $\lambda_i(s)$ using a concept that we refer to as *pseudo-stationarity*. We say that payoff matrix $R_i(s)$ is pseudo-stationary if each entry of $R_i(s)$ has stopped decreasing.

**Pseudo-Stationary:** Let $\underline{R}_i^T(s,\mathbf{a})$ be the lowest estimate of $R_i(s,\mathbf{a})$ observed up to time $T$, and let $R_i^t(s,\mathbf{a})$ be the estimate of $R_i(s,\mathbf{a})$ at time $t$. $R_i(s)$ is *pseudo-stationary* after time $T$ if $\forall t\geq T, \mathbf{a}\in A(s)$, $R_i^t(s,\mathbf{a})\geq \underline{R}_i^T(s,\mathbf{a})+\delta$, where $\delta>0$ is some small positive constant.

Since $R_i(s,\mathbf{a})$ is initially set to $R_i^{max}$, it will likely decrease in early episodes. Thus, $R_i(s)$ will not likely be pseudo-stationary in early episodes of the game, but will eventually become pseudo-stationary given episodes of finite length.

Pepper uses the concept of *non-pseudo-stationary re-starts* to regulate $\lambda_i(s)$. That is, when $R_i(s)$ is observed to not be pseudo-stationary, $\lambda_i(s)$ is reset to one. This restarts the transition from $\hat{V}_i(s)$ to $V_i^{\text{on}}(s)$ defined by Eq. (6). Let $\kappa'(s)$ be the number of visits to stage $s$ since $R_i(s)$ was last observed to not be pseudo-stationary. Then, $\lambda_i(s)$ is given by:

$$\lambda_i(s) = \max\left(0,\ \frac{C-\kappa'(s)}{C}\right), \quad (7)$$

where $C$ is some positive integer. $\lambda_i(s)$ decreases more slowly for higher values of $C$ than for lower values of $C$ in the absence of non-pseudo-stationary restarts.

Since the number of restarts in each stage is finite given bounded rewards, $V_i(s)$ as defined by Eqs. (6) and (7) will converge to actual rewards if each $(s,\mathbf{a})$-pair is visited sufficiently. Since Algorithm 1 ensures with high probability that all $(s,\mathbf{a})$-pairs will be explored sufficiently given that the optimism property is met [8], $V_i(s)$ will converge to actual rewards in stages encountered in the learned solution when the optimism property is met.

## 4. ALGORITHMS FORMED BY PEPPER

We now describe four new algorithms for RSGs formed by extending four different MLAs with Pepper. Algorithm 1 paired with Eq. (6) requires that we must only define how $\pi_i(s)$ and $V_i^{\text{off}}(s)$ are computed. We refer the reader to the literature for specifications of how $\pi_i(s)$ is computed by each MLA, as Pepper uses these rules as they have been defined. We now specify how each algorithm computes $V_i^{\text{off}}(s)$.

### 4.1 M-Qubed with Pepper

M-Qubed [10] is a reinforcement learning algorithm that balances cautious, optimistic, and best-response attitudes. It encodes the previous $\omega$ joint actions taken by the agents as state (called *recurrent state*). It then learns a Q-value for each recurrent state-action pair, each of which is initialized to its highest possible value given its discount factor $\gamma$. M-Qubed typically selects actions based on its Q-values in the current (recurrent) state, but triggers to its maximin strategy when its total loss exceeds a pre-determine threshold.

M-Qubed learns to play the Nash bargaining solution in self play in many repeated matrix games. It also avoids being exploited, meaning that its long-term payoffs meet or exceed its maximin value regardless of the behavior of its associates. In the PD matrix game, it learns to cooperate in self play and to defect against associates that defect.

M-Qubed's mechanics define a relaxation search for a strategy that sustains a future discounted reward $\frac{r}{1-\gamma}$ that meets or exceed its highest current Q-value over all of its recurrent states (denoted $Q^*(s)$). Thus, basing $V_i^{\text{off}}(s)$ on $Q^*(s)$ is a natural choice. Formally, let $\Omega(s)$ be the set of M-Qubed's recurrent states in stage $s$, and let $\Omega'(s)\subseteq\Omega(s)$ be the set of recurrent states visited in the last $\tau$ visits to $s$. Also, let

$Q(\sigma, a_i)$ be the Q-value for taking action $a_i$ in $\sigma \in \Omega$. Then,

$$V_i^{\text{off}}(s) = (1-\gamma) \cdot \left( \max_{\sigma \in \Omega'(s), a_i \in A_i(s)} Q(\sigma, a_i) \right) \quad (8)$$

In RSGs, M-Qubed's recurrent state $\sigma \in \Omega(s)$ in stage $s$ is determined by the previous $\omega$ joint actions taken in $s$.

## 4.2 Salt and Pepper

The satisficing learning technique (Salt) is a simple MLA proposed by Karandikar et al. [18]. We use the version of the algorithm defined and analyzed by Stimpson and Goodrich [22]. Salt converges with high probability in self play to pareto efficient solutions. In the repeated PD matrix game, it learns with high probability to cooperate in self play and to defect against agents that *always* defect [22].

Salt and Pepper encodes an aspiration level $\alpha_i(s)$ in each stage $s \in S$, which is initialized to $\max_{\mathbf{a}} R_i(s, \mathbf{a})$. $\alpha_i(s)$ is then incremented toward $R_i(s, \mathbf{a})$ when $\mathbf{a}$ is played in $s$. When $\alpha_i^t \geq R_i(s, \mathbf{a})$, Salt and Pepper repeats its action the next time $s$ is visited. Otherwise, it randomly selects a new action. Salt and Pepper sets $V_i^{\text{off}}(s)$ to $\alpha_i^t$, which typically provides an optimistic estimate of $V_i(s)$ in early episodes.

## 4.3 Fictitious Play with Pepper

Fictitious play (FP) [11] is one of the oldest MLAs. It forms a simple model of its opponent by observing the empirical distribution of its opponent's actions. In each time step, it selects the action that maximizes its expected payoff given this opponent model and its payoff matrix. Formally, let $\gamma(a_{-i})$ be the percentage of time that its opponent (player $-i$) has played action $a_{-i}$ in the past. Then, FP's utility for playing action $a_i$ is:

$$u_i(a_i) = \sum_{a_{-i} \in A_{-i}(s)} \gamma(a_{-i}) \; R_i(s, (a_i, a_{-i})). \quad (9)$$

We implemented weighted FP, in which the assessment $\gamma(a_{-i})$ gives more weight to recent observations,

FP converges to a NE in self play in matrix games that are iterative dominance solvable. In the PD matrix game, it learns to defect against all associates regardless of their propensity to cooperate or retaliate.

FP with Pepper sets $V_i^{\text{off}}(s)$ to its max utility. Formally,

$$V_i^{\text{off}}(s) = \max_{a_i \in A_i(s)} u_i(a_i). \quad (10)$$

This valuation is not optimistic when $R_i(s)$ has converged to actual payoffs. However, since $R_i(s)$ is initialized optimistically, it is optimistic initially.

## 4.4 GIGA-WoLF with Pepper

GIGA-WoLF [6] is a gradient ascent MLA that uses multiple learning rates to achieve no regret. Unlike the other three learning algorithms, GIGA-WoLF selects a strategy from the mixed strategy space. In the PD matrix game, GIGA-WoLF quickly learns to defect against all associates.

GIGA-WoLF with Pepper sets $V_i^{\text{off}}(s)$ to its weighted average reward (given by $R_i(s)$); newer samples receive more weight. Thus, like FP with Pepper, this valuation is optimistic initially, and falls to true values as $R_i(s)$ converges.

## 5. RESULTS

In this section, we evaluate the behavior and performance of M-Qubed with Pepper, Salt and Pepper, FP with Pepper,

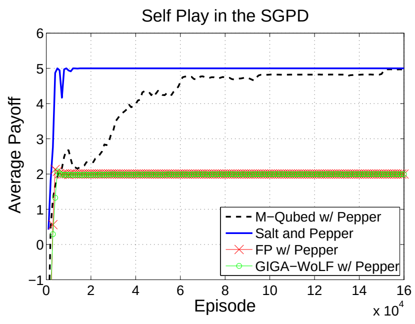

**Figure 2: Average payoffs in self play over 20 trials.**

and GIGA-WoLF with Pepper in the SGPD. First, we analyze their behavior in self play and against WoLF-PHC [7], an MAL algorithm that typically learns to defect in this game. We then provide a more in-depth analysis of the robustness of these algorithms when associating with many different kinds of associates in the SGPD, including association among the various Pepper algorithms, other existing learning algorithms, and hand-coded (static) strategies. We do so by conducting a round-robin tournament and an evolutionary tournament, each involving the same 12 algorithms. Parameter values for all algorithms are provided in Table 5.

## 5.1 Pepper in the SGPD

Figure 2 shows the average performance over time of M-Qubed with Pepper, Salt and Pepper, FP with Pepper, and GIGA-WoLF with Pepper in the SGPD in self play. The figure shows that both M-Qubed with Pepper and Salt and Pepper learn to cooperate in self play. This results in an average asymptotic payoff of 5 points per episode. However, while both of these algorithms learn to cooperate in self play, Salt and Pepper converges much faster than M-Qubed with Pepper. These results are consistent with the behavior of these algorithm in the PD matrix game, in which both Salt and M-Qubed learn to cooperate in self play, with Salt converging faster than M-Qubed.

Alternately, Figure 2 shows that both FP with Pepper and GIGA-WoLF with Pepper quickly learn to defect in self play in the SGPD. This results in an asymptotic payoff of 2 points per episode. This is substantially less than if both agents had learned to cooperate, but it is consistent with how GIGA-WoLF and FP perform in the PD matrix game.

Against WoLF-PHC in the SGPD, the average asymptotic payoff of each algorithm is near 2 points per episode (Figure 3). All four algorithms learn to defect against WoLF-PHC, though Salt and Pepper's average payoffs are slightly lower than that of mutual defection. This degraded performance is due to occasional exploration by WoLF-PHC, which causes Salt and Pepper to become dissatisfied in some stages. This requires it to re-learn how to defect, which typically takes several episodes. Again, M-Qubed with Pepper learns much slower than the other algorithms.

Despite M-Qubed with Pepper's slow learning rate, we are not aware of another learning algorithm from the literature that, without knowing its associate's payoffs, learns to both cooperate in self play and to always defect against WoLF-PHC in the SGPD. This demonstrates the effectiveness of Pepper for creating algorithms that outperform existing MAL algorithms in RSGs.

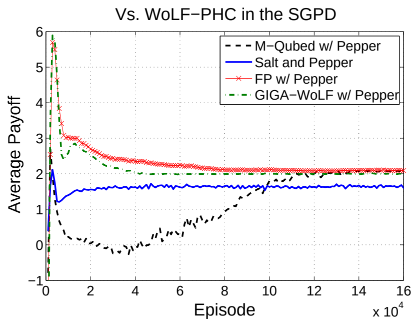

**Figure 3: Average payoffs against WoLF-PHC over 20 trials.**

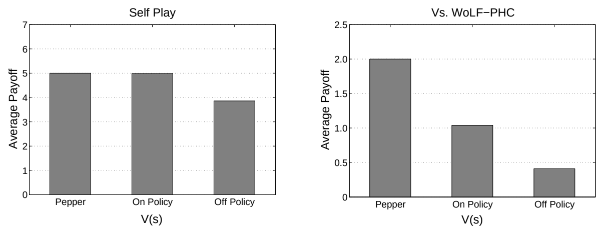

**Figure 4: Average asymptotic payoffs of M-Qubed with Pepper given different estimates of $V_i(s)$ in self play and against WoLF-PHC.**

The success of M-Qubed with Pepper can be traced in large part to how Pepper estimates $V_i(s)$. Figure 4 shows the average payoffs of M-Qubed in self play and against WoLF-PHC given different methods for estimating $V_i(s)$. When $V_i(s)$ is set equal to $V_i^{\text{on}}(s)$ (On Policy), M-Qubed still learns to cooperate in self play, but it gets exploited by WoLF-PHC. When $V_i(s)$ is set equal to $V_i^{\text{off}}(s)$ (Off Policy), its payoffs in self play and against WoLF-PHC are substantially lower than when $V_i(s)$ is set by Pepper. In fact, in one trial (not reflected in Figure 4) in self play, using the off-policy valuation caused both agents to converge to a strategy in which neither agent entered a gate within 200 moves.

To better understand the $V_i(s)$ as it is computed by Pepper, consider Figure 5. This figure shows values of $V_i^{\text{off}}(s)$, $V_i^{\text{on}}(s)$, and $V_i(s)$ over time in the stage game in which each agent is immediately next to an open gate 1. Figures 5(a)–5(d) correspond to valuations made by each of the Pepper algorithms in self play, while Figures 5(e) and 5(f) show valuations of M-Qubed with Pepper and Salt and Pepper against WoLF-PHC. We note that, from this particular stage, mutual defection gives each agent a future reward of 4, and mutual cooperation gives each agent a future reward of 7.

We make several observations about Figure 5. First, all valuations eventually converge to the true value of the stage in question in each scenario. Second, $V_i^{\text{off}}(s)$ is often greater than $V_i^{\text{on}}(s)$ in each scenario. Thus, $V_i(s)$ tends to mirror $V_i^{\text{off}}(s)$ except in the case of M-Qubed with Pepper (Figures 5(a) and 5(e)), particularly against GIGA-WoLF. In this latter scenario, $V_i(s)$ eventually mirrors $V_i^{\text{on}}(s)$, which allows it to defect against WoLF-PHC. Third, $V_i(s)$ typically, but not always, conforms with the optimism property; $V_i(s)$ is usually greater than or equal to the eventual value of the stage. This causes the algorithms to effectively explore using the optimism-in-uncertainty principle.

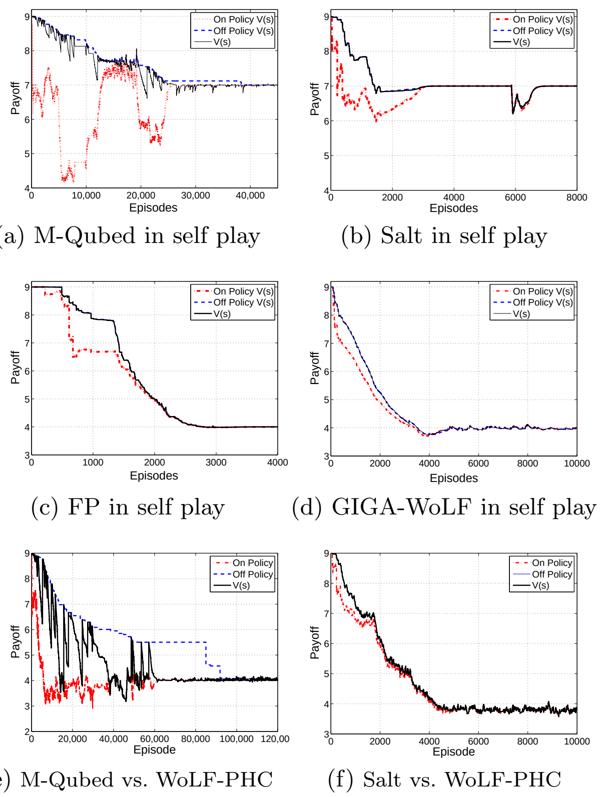

**Figure 5: $V_i^{\text{on}}(s)$, $V_i^{\text{off}}(s)$, and $V_i(s)$ in the SGPD for the stage where both agents are in front of gate 1.**

## 5.2 Tournaments in the SGPD

We conducted a round-robin tournament involving 12 algorithms: the four Pepper algorithms, four other MAL algorithms (Q-learning [24], WoLF-PHC, Friend-VI [20], and Minimax-VI [20]), and four static strategies (Tit-for-Tat, Always Defect, Always Cooperate, and Random). In this tournament, each algorithm was paired with itself and the other algorithms in a 200,000-episode SGPD. Since we are primarily concerned with asymptotic performance in this paper, the performance of the algorithms was taken only in the last 10,000 episodes. Alternate evaluation windows (such as the average of all episodes) could yield different results.

The results of the round-robin tournament are shown Table 4. M-Qubed with Pepper had the highest average performance, followed by Always Defect, Tit-for-Tat, and Salt and Pepper. FP with Pepper and GIGA-WoLF with Pepper placed fifth and sixth, respectively.

In addition to learning to cooperate in self play, M-Qubed with Pepper learns mutual cooperation with Salt and Pepper. On the other hand, it learns to always defect against each of the other algorithms except Tit-for-Tat. This allows it to avoid being exploited by algorithms that are apt to defect, and to exploit algorithms that will cooperate (Always Cooperate, Friend-VI, and, to a lesser degree, Random).

However, M-Qubed with Pepper does not learn to always cooperate with Tit-for-Tat, nor do any of the other learning algorithms. While an ideal algorithm cooperates with Tit-for-Tat, the version of Tit-for-tat we implemented for the SGPD responds to the global behavior of its associate,

Table 4: **Average asymptotic payoffs in the SGPD for each pairing. All results are an average of 20 trials.**

| **Algorithm** | **Associate** | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | M-Qubed w/ Pepper | Always Defect | TFT | Salt and Pepper | FP w/ Pepper | GIGA-WoLF w/ Pepper | Q-learning | Minimax-VI | Random | WoLF-PHC | Always Cooperate | Friend-VI | Ave. |
| 1. M-Qubed w/ Pepper | 5.00 | 2.00 | 3.86 | 5.00 | 1.99 | 2.07 | 2.01 | 2.01 | 4.48 | 2.08 | 7.00 | 6.97 | **3.70** |
| 2. Always Defect | 2.00 | 2.00 | 2.00 | 2.00 | 2.00 | 2.08 | 5.03 | 2.08 | 4.51 | 5.05 | 7.00 | 6.38 | **3.51** |
| 3. TFT | 3.89 | 2.00 | 5.00 | 3.62 | 4.00 | 2.06 | 3.62 | 2.07 | 3.75 | 2.27 | 5.00 | 4.42 | **3.47** |
| 4. Salt and Pepper | 4.89 | 2.00 | 2.69 | 5.00 | 2.00 | 1.63 | 1.63 | 1.62 | 2.81 | 1.64 | 7.00 | 6.96 | **3.32** |
| 5. FP w/ Pepper | 2.02 | 2.00 | 3.00 | 2.00 | 2.00 | 2.08 | 2.11 | 2.08 | 4.50 | 2.08 | 7.00 | 6.39 | **3.11** |
| 6. GIGA-WoLF w/ Pepper | 1.93 | 1.91 | 1.99 | 2.86 | 1.91 | 1.99 | 2.00 | 1.99 | 4.44 | 2.00 | 6.97 | 5.55 | **2.96** |
| 7. Q-learning | 1.98 | 1.31 | 3.37 | 2.71 | 1.87 | 1.95 | 2.15 | 1.95 | 4.32 | 3.12 | 6.96 | 3.80 | **2.96** |
| 8. Minimax-VI | 1.97 | 1.91 | 2.00 | 2.89 | 1.91 | 2.00 | 2.00 | 1.99 | 4.45 | 2.00 | 6.97 | 4.78 | **2.91** |
| 9. Random | 1.52 | 1.50 | 3.75 | 3.49 | 1.50 | 1.54 | 1.71 | 1.55 | 3.75 | 2.40 | 6.00 | 5.07 | **2.81** |
| 10. WoLF-PHC | 1.90 | 1.21 | 2.14 | 2.71 | 1.89 | 1.95 | 1.64 | 1.94 | 4.06 | 1.95 | 6.94 | 1.94 | **2.52** |
| 11. Always Cooperate | 1.00 | 1.00 | 5.00 | 1.00 | 1.00 | 0.99 | 1.00 | 1.00 | 3.00 | 1.02 | 5.00 | 3.76 | **2.06** |
| 12. Friend-VI | -0.85 | -0.94 | 1.99 | -0.89 | -0.87 | -0.63 | -1.20 | -0.96 | 1.01 | 1.96 | 2.98 | 1.93 | **0.29** |

whereas Pepper and the other learning algorithms in our study learn locally. This prohibits these algorithms from observing global effects not connected in the Markov chain.

Like M-Qubed with Pepper, Salt and Pepper learns to defect against Friend-VI, Always Cooperate, Always Defect, and FP with Pepper. However, as against WoLF-PHC (Figure 3), Salt and Pepper is sometimes exploited by GIGA-WoLF with Pepper and Minimax-VI. Likewise, it does not always defect against Random. Meanwhile, both GIGA-WoLF with Pepper and FP with Pepper learn to defect against all associates, with some slight variations caused by GIGA-WoLF's exploration strategy. These results are consistent with the behavior of these algorithms in the PD matrix game, which demonstrates Pepper's ability to extend algorithms designed for repeated matrix games to RSGs.

We also conducted an evolutionary tournament in the SGPD. In this tournament, an arbitrarily large population of agents, each using one of the 12 algorithms, was evolved over a series of generations according to the algorithms' performance in the SGPD against the agents in the population. Initially, each algorithm was equally represented in the population. In each subsequent generation, the population was altered using the replicator dynamic [23].

Figure 6 shows the proportion of the population using each algorithm over time. After about 10 generations, Tit-for-Tat and M-Qubed with Pepper dominated the population, with Salt and Pepper also holding a small but substantial share. However, once these three algorithms dominated the population, M-Qubed with Pepper quickly took over.

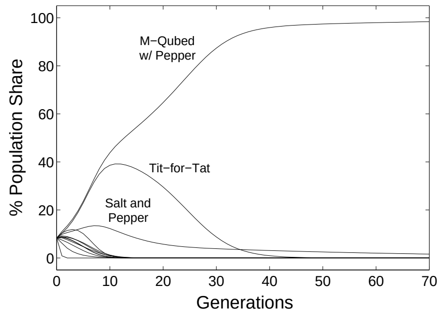

**Figure 6: Results of the evolutionary tournament.**

## 6. CONCLUSIONS AND DISCUSSION

In this paper, we presented a new algorithm, called Pepper, which is designed to extend learning algorithms designed for repeated matrix games to algorithms capable of playing effectively in repeated stochastic games (RSGs). To demonstrate the usefulness of Pepper, we extended four matrix learning algorithms from the literature to algorithms for RSGs using Pepper. We then evaluated their performance in a stochastic game prisoner's dilemma (SGPD). Our results show that the behavior of these algorithms in the SGPD is reminiscent of their behavior in the corresponding prisoner's dilemma matrix game.

As in many other MAL algorithms for RSGs, one drawback of Pepper is that all learning is local (within a stage). Thus, it does not account for some effects that are only visible globally. In this paper, this was demonstrated by the fact that none of the learning algorithms we considered were able to learn to consistently cooperate with Tit-for-Tat.

Our results also found that combining the M-Qubed algorithm with Pepper produces an algorithm that learns in the SGPD to cooperate in self play, while learning to defect against associates that are not apt to cooperate. We are not aware of another algorithm in the literature that is able to achieve this without knowing its associate's payoffs as well as the state transitions of the game. As a result, M-Qubed with Pepper outperformed the other algorithms in both a round-robin and evolutionary tournament.

Despite its robust behavior in the SGPD, M-Qubed with Pepper learns very slowly. While its learning rate can be increased to some degree by simply adjusting M-Qubed's learning rate $\alpha$, perhaps a more effective solution would be to deduce when a conflict between agents is possible [15] or to use different kinds of matrix learning algorithms (MLAs) in each stage of the game. The selection of the MLA used in each stage could be based on that stage's payoff matrix. Faster matrix learning algorithms could be used in stages that do not appear to require sophisticated reasoning, whereas slower matrix learning algorithms (such as M-Qubed) could be used in stages that appear to require more sophistication. Pepper makes this possible.

## 7. ACKNOWLEDGMENTS

The author would like to thank Michael A. Goodrich of Brigham Young University for his helpful feedback.

## 8. REFERENCES

[1] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multi-armed bandit problem. *Machine Learning*, 47 (2–3):235–256, 2002.

[2] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. Gambling in a rigged casino: the adversarial multi-armed bandit problem. In *Proc. of the 36th Symp. on the Foundations of CS*, pages 322–331. IEEE Computer Society Press, 1995.

[3] R. E. Bellman. *Dynamic Programming.* Princeton University Press, NJ, 1957.

[4] B. Bouzy and M. Metivier. Multi-agent learning experiments in repeated matrix games. In *Proc. of the 27th Intl. Conf. on Machine Learning*, 2010.

[5] M. Bowling. Convergence problems of general-sum multiagent reinforcement learning. In *Proc. of the 17th Intl. Conf. on Machine Learning*, pages 89–94, 2000.

[6] M. Bowling. Convergence and no-regret in multiagent learning. In *Advances in Neural Information Processing Systems 17*, pages 209–216, 2005.

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

[8] R. I. Brafman and M. Tennenholtz. R-max – a general polynomial time algorithm for near-optimal reinforcement learning. *The Journal of Machine Learning Research*, 3:213–231, March 2003.

[9] E. M. De Cote and M. L. Littman. A polynomial-time Nash equilibrium algorithm for repeated stochastic games. In *Proc. of the 24th Conf. on Uncertainty in Artificial Intelligence*, 2008.

[10] J. W. Crandall and M. A. Goodrich. Learning to compete, compromise, and cooperate in repeated general-sum games. *Machine Learning*, 82(3):281–314, 2011.

[11] D. Fudenberg and D. K. Levine. *The Theory of Learning in Games.* The MIT Press, 1998.

[12] Herbert Gintis. *Game Theory Evolving: A Problem-Centered Introduction to Modeling Strategic Behavior.* Princeton University Press, 2000.

[13] M. A. Goodrich, J. W. Crandall, and J. R. Stimpson. Neglect tolerant teaming: Issues and dilemmas. In *AAAI Spring Symp. on Human Interaction with Autonomous Systems in Complex Environments*, 2003.

[14] A. Greenwald and K. Hall. Correlated Q-learning. In *Proc, of the 20th Intl. Conf. on Machine Learning*, pages 242–249, 2003.

[15] Y. M. De Hauwere, P. Vranx, and A. Nowe. Future sparse interactions: A MARL approach. In *Proc. of the 9th European Workshop on Reinforcement Learning*, 2011.

[16] J. Hu and M. P. Wellman. Multiagent reinforcement learning: Theoretical framework and an algorithm. In *Proc. of the 15th Intl. Conf. on Machine Learning*, pages 242–250, 1998.

[17] M. Johanson, N. Bard, M. Lanctot, R. Gibson, and M. Bowling. Efficient Nash equilibrium approximation through Monte Carlo counterfactual regret minimization. In *Proc. of the 11th Intl. Conf. on Autonomous Agents and Multiagent Systems*, 2012.

[18] Rajeeva Karandikar, Dilip Mookherjee, Debraj Ray, and Fernando Vega-Redondo. Evolving aspirations and cooperation. *Journal of Economic Theory*, 80:292–331, 1998.

[19] M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proc. of the 11th Intl. Conf. on Machine Learning*, pages 157–163, 1994.

[20] M. L. Littman. Friend-or-foe: Q-learning in general-sum games. In *Proc. of the 18th Intl. Conf. on Machine Learning*, pages 322–328, 2001.

[21] H. Qiao, J. Rozenblit, F. Szidarovszky, and L. Yang. Multi-agent learning model with bargaining. In *The 2006 Winter Simulation Conf.*, pages 934–940, 2006.

[22] J. R. Stimpson and M. A. Goodrich. Learning to cooperate in a social dilemma: A satisficing approach to bargaining. In *Proc. of the 20th Intl. Conf. on Machine Learning*, pages 728–735, 2003.

[23] P. D. Taylor and L. Jonker. Evolutionarily stable strategies and game dynamics. *Mathematical Biosciences*, 40:145–156, 1978.

[24] C. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

## 9. APPENDIX

**Table 5: Parameter settings and algorithm specifications.**

| | |
|---|---|
| Pepper | $K = 20$ (for FP) and $K = 5$ (for Salt, M-Qubed, and GIGA-WoLF); $C = 1000$ |
| M-Qubed | $\gamma = 0.95$, $\eta(t) = \frac{0.04 \cdot 1000}{1000 + \max_s \kappa_t(s)}$, $\omega = 1$, $\xi \in [0.1, 0.15]$, $\alpha = 0.1$, $\tau = 10,000$, and $L_i^{\text{tol}} = 500 \cdot \lvert A_i(s)\rvert \cdot \lvert A_i(s)\rvert \cdot \lvert A_{-i}(s)\rvert\ \xi$, |
| Salt | $\lambda = 0.99$ |
| Fictitious Play | $\gamma_i^{t+1}(a) = \alpha\gamma_i^t(a) + (1-\alpha)I(a, a_{-i})$, where $I(\cdot)$ is the indicator function and $\alpha = \min\left(0.99, \frac{\kappa_i(s)-1}{\kappa_i(s)}\right)$ |
| GIGA-WoLF | $\eta = 0.01$; explores with probability 0.01; uses $R_i(s)$ to estimate the reward gradient |
| Minimax-VI | Uses Algorithm 1 with $K = 20$; explores with probability 0.01 |
| Friend-VI | Uses Algorithm 1 with $K = 20$; explores with probability 0.01 |
| WoLF-PHC | $\alpha = \frac{1}{100+\kappa_i(s,\mathbf{a})/10000}$, $\delta = \delta_w = \frac{1}{20000+t}$, $\delta_l = 4\delta_w$, $\forall s, a$, $Q^0(s,a) = \text{rand}\left(0, \frac{r_i^{\max}}{1-\gamma}\right)$, explores with probability 0.01, initial policy is a random mixed strategy |
| Q-learning | $\alpha = \frac{1}{10+\kappa_i(s,\mathbf{a})/10000}$, $\gamma = 1$, explores w/ prob. $\max(0.01, 0.2 - \frac{\kappa_i(s)}{100,000})$ |