Satisficing Multi-Agent Learning: A Simple But Powerful Algorithm

Jacob W. Crandall and Michael A. Goodrich*

October 10, 2008

Abstract

Learning in the presence of adaptive, possibly antagonistic, agents presents special challenges to algorithm designers, especially in environments with limited information. We consider situations in which an agent knows its own set of actions and observes its own payoffs, but does not know or observe the actions and payoffs of the other agents. Despite this limited information, a robust learning algorithm must have two properties: *security*, which requires the algorithm to avoid exploitation by antagonistic agents, and *efficiency*, which requires the algorithm to find nearly pareto efficient solutions when associating with agents who are inclined to cooperate. However, no learning algorithm in the literature has both of these properties when playing repeated general-sum games in these limited-information environments. In this paper, we present and analyze a variation of Karandikar *et al.*'s learning algorithm [19]. The algorithm is conceptually very simple, but has surprising power given this simplicity. It is *provably secure* in all matrix games, regardless of the play of its associates, and it is *efficient* in self play in a very large set of matrix games. Additionally, the algorithm performs well when associating with representative, state-of-the-art learning algorithms with similar representational capabilities in general-sum games. These properties make the algorithm highly robust, more so than representative best-response and regret-minimizing algorithms with similar reasoning capabilities.

1 Introduction

Since intelligent agents must often repeatedly interact with other intelligent agents in uncertain, previously unknown, environments, there is a need to develop algorithms that can learn successfully and robustly in the presence of other learning agents of various types. To be successful, Axelrod's work [2] suggests that a learning should have at least two properties: (1) *security*, which requires an agent to avoid being exploited by antagonistic agents, and (2) *efficiency*, which requires an agent to find nearly pareto efficient solutions when in the presence of associates who are inclined to cooperate.

Finding an algorithm simultaneously possessing both of these properties in repeated general-sum matrix games is difficult for a number of reasons. First, the presence of multiple learning agents who asynchronously adapt to each other effectually produces a non-stationary environment that prevents the straightforward application of most existing optimization and learning algorithms. Second, the goals of efficiency and security are often competing objectives that require completely different behaviors. A secure response is often not an efficient response and vice versa. Third, many environments afford little or no information about an agent's associates, including their existence, their actions and payoffs, and the learning algorithms they employ. We refer to such environments throughout this paper as *minimal information environments*.

We know of no artificial learning algorithm described in the literature that is simultaneously efficient and secure in repeated general-sum matrix games in minimal information environments. In this paper, we

*J. W. Crandall is an Assistant Professor at the Masdar Institute of Science and Technology, Abu Dhabi, UAE. M. A. Goodrich is an Associate Professor in the Computer Science Department, Brigham Young University, Provo, UT, USA.

present an algorithm, called SALT, that is both efficient in self play and secure. Moreover, rather than requiring ever-increasing representational and reasoning capacity, SALT requires limited representational and reasoning capacity.

SALT is a modification of a satisficing learning algorithm [19, 36], which we call the *S-algorithm*. As is common in both the game theoretic and multi-agent learning literature, the S-algorithm establishes an equilibrium concept as the attractor for the learning process. Instead of using the best-response equilibrium identified by Nash [27], the S-algorithm uses Stirling's notion of a *satisficing equilibrium* [39]. However, unlike Stirling, who introduces substantial representational and computational infrastructure, the S-algorithm uses the minimalist representation of satisficing used by Simon [33, 34] in which an agent seeks to identify an appropriate target payoff threshold (i.e., an *aspiration level*) and a behavior to sustain it.

In this paper, we show that the S-algorithm is efficient in self play in repeated general-sum matrix games when certain initial conditions are satisfied. These conditions include properties of the game itself as well as the players' initial aspiration levels. We also show that the S-algorithm cannot guarantee security in all general-sum matrix games against all associates. We then introduce SALT, which is a trembling-hand version of the S-algorithm. We demonstrate that SALT has three properties in general-sum matrix games played in minimal information environments:

1. It is provably secure in all general-sum matrix games, regardless of its associates.
2. It is efficient in a vast majority of general-sum games, as it eliminates dependence on initial aspiration levels.
3. It performs well in head-to-head associations with representative learning algorithms with similar reasoning capacity.

These three properties make SALT significantly more successful in many contexts than other learning algorithms with similar reasoning capabilities. In fact, with respect to a representative set of such learning algorithms, SALT is, on average, evolutionarily stable [2] in general-sum matrix games, thus demonstrating the powerful learning capabilities of satisficing learning. These results show that, rather than focus solely on myopic optimization techniques such as the best-response and regret minimization, designers of multi-agent learning algorithms should also consider the powerful learning principles implemented in satisficing learning in order to maximize long-term payoffs in repeated general-sum games.

The remainder of this paper proceeds as follows. In the next section, we provide necessary background information and discuss related work. In Section 3, we review the S-algorithm. In Section 4, we analyze the S-algorithm in repeated general-sum matrix games in minimal information environments with respect to the security and efficiency properties. We describe SALT in Section 5, and then analyze its performance properties in Section 6. We state conclusions and discuss future work in Section 7.

2 Background and Related Literature

In this section, we first define terms and definitions related to multi-agent learning in repeated general-sum matrix games. We then formally define *secure* and *efficient* learning and review past work found in the literature.

2.1 Terms and Definitions

Let I be a set of N agents, each of which selects actions from a finite set A_i. Let $\mathbf{a} = (a_1, a_2, \ldots, a_N)$, where $a_i \in A_i$, be a *joint action* for N agents, and let $A = A_1 \times \cdots \times A_N$ be the agents' set of joint actions. Let $a_i^t \in A_i$ denote the action played by agent i at time t, and let $\mathbf{a}^t = (a_1^t, a_2^t, \ldots, a_N^t)$ denote the joint action played by the N agents at time t. Also, let $\mathbf{a}^t(J)$, where $J \subseteq I$, be the joint actions of the $|J|$ agents in J. Let $\mathbf{a}_{-i}$ be the joint actions of all agents except agent i; the agents differing from agent i are referred to as agent i's *associates*.

Central to multi-agent learning is the matrix game, which consists of payoff matrices $R = \{R_1, \cdots, R_N\}$, where R_i is the payoff matrix of agent i. Let $r_i(\mathbf{a})$ be the payoff to agent i when the joint action $\mathbf{a}$ is played, and let r_i^t be the payoff to agent i at time t. Also, let $r_i(a_i, \mathbf{a}_{-i})$ be the payoff to player i when it plays a_i and the rest of the agents play $\mathbf{a}_{-i}$. In this paper, we assume deterministic payoffs for each joint action $\mathbf{a} \in A$.

A *strategy* for agent i is a distribution π_i over its action set A_i. A strategy may be a *pure strategy* (where all probability is placed on a single action) or a *mixed strategy* (otherwise). The joint strategy played by the n agents is $\boldsymbol{\pi} = (\pi_1, \pi_2, \ldots, \pi_N)$ and, thus, $r_i(\boldsymbol{\pi})$ is the expected payoff for agent i when the joint strategy $\boldsymbol{\pi}$ is played. A *solution* is a particular joint strategy.

Definition 1. *(Matrix Game) A matrix game is the tuple (I, A, R), where I, A, and R are defined above.*

A *repeated matrix game* is when the same set of agents repeatedly play the same matrix game (i.e., a series of *episodes*). In this paper, we assume that play repeats indefinitely, or at least for a large number of episodes.

2.2 Measures of Solution Quality

To maximize individual payoffs, it has often been assumed that a learning algorithm should learn to play a *best-response* to the strategies of associates (i.e., $\pi_i^* = \text{argmax}_{\pi_i} R_i(\pi_i, \pi_{-i})$). When all agents play a best-response to the strategies of their associates, the result is a *Nash equilibrium* (NE). However, in infinitely repeated games, the folk theorem shows that there are an infinite number of NE [15]. As such, the NE solution concept does not indicate how an agent should play a repeated game played with learning associates, unless the agent can know exactly how associates will adapt and react to each of its possible strategies. Given the minimal information environments considered in this paper, such an expectation is impractical.

Axelrod identified successful behavior as a combination of *security* and *efficiency*. In his study of the prisoner's dilemma, Axelrod showed that tit-for-tat is a robust strategy since it avoids being exploited (i.e., security) and it cooperates with other agents that are inclined to cooperate (i.e., efficiency) [2]. Furthermore, he identified efficiency in self play as a crucial characteristic for algorithmic robustness, especially in the context of evolutionary stability [2].

Thus, in this paper, we focus primarily on being secure against all associates, and efficient in self play. However, while we emphasize efficiency in self play, we also acknowledge and consider the importance of efficiency when playing other kinds of learners. In so doing, we consider an algorithm's ability to avoid invasion by other learning algorithms.

We now formally define the properties of security and efficiency, beginning with *pareto efficiency*.

Definition 2. *(Pareto Dominated) A solution $\boldsymbol{\pi}$ is strictly pareto dominated if there exists a joint action $\mathbf{a} \in A$ for which $r_i(\mathbf{a}) > r_i(\boldsymbol{\pi})$ for all i and weakly pareto dominated if there exists a joint action $\mathbf{a} \in A$, $\mathbf{a} \neq \boldsymbol{\pi}$ for which $r_i(\mathbf{a}) \geq r_i(\boldsymbol{\pi})$ for all i.*

Definition 3. *(Pareto Efficient) A solution $\boldsymbol{\pi}$ is weakly pareto efficient (PE) if it is not strictly pareto dominated and strictly PE if it is not weakly pareto dominated. Unless specified, the former (i.e. weakly PE) terminology is implied.*

While pareto efficiency can be a goal of an agent who is involved in repeated play, it is sometimes difficult and impractical to guarantee. We relax this goal slightly to near-pareto efficiency. Let ε be a small positive constant.

Definition 4. *(ε-Pareto Dominated) A solution $\boldsymbol{\pi}$ is strictly ε-pareto dominated if there exists a joint action $\mathbf{a} \in A$ for which $r_i(\boldsymbol{a}) > r_i(\boldsymbol{\pi}) + \varepsilon$ for all i and weakly ε-pareto dominated if there exists a joint action $\mathbf{a} \in A$, $\mathbf{a} \neq \boldsymbol{\pi}$ for which $r_i(\mathbf{a}) \geq r_i(\boldsymbol{\pi}) + \varepsilon$ for all i.*

Definition 5. *(ε-Pareto Efficient) A solution $\boldsymbol{\pi}$ is weakly ε-pareto efficient (ε-PE) if it is not strictly ε-pareto dominated and strictly ε-PE if it is not weakly ε-pareto dominated. The former (i.e. weakly ε-PE) terminology is implied unless specified otherwise.*

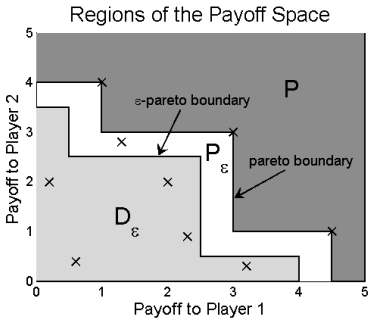


Figure 1: Diagram of the regions $\mathbb{P}$, $\mathbb{P}_\varepsilon$, and $\mathbb{D}_\varepsilon$ in the payoff space $\mathbb{R}^2$ for an arbitrary 3×3 matrix game.

We now give names to various regions of the payoff space, some of which are illustrated in Figure 1 for an arbitrary 2-player 3-action matrix game. The $\times$'s depict the joint payoffs of the pure strategy solutions of the game. Let $\mathbb{R}^n$ denote the real-valued reward space. Let $\mathbb{D}$ be the subset of $\mathbb{R}^n$ that is weakly pareto dominated. Let $\mathbb{P} = \mathbb{R}^n - \mathbb{D}$. That is, $\mathbb{P}$ is the subset of $\mathbb{R}^n$ that is strictly PE. Also, let $\mathbb{D}_\varepsilon$ be the subset of $\mathbb{R}^n$ that is strictly ε-pareto dominated. Finally, let $\mathbb{P}_\varepsilon = \mathbb{D} - \mathbb{D}_\varepsilon$.

The *pareto boundary* divides set $\mathbb{D}$ from set $\mathbb{P}$. If $\mathbf{v} \in \mathbb{D}$, then $\mathbf{v}$ is said to be below the pareto boundary. If $\mathbf{v} \in \mathbb{P}$, $\mathbf{v}$ is said to be above the pareto boundary. Likewise, the ε-pareto boundary divides set $\mathbb{D}_\varepsilon$ from set $\mathbb{P}_\varepsilon$. If $\mathbf{v} \in \mathbb{D}_\varepsilon$ or $\mathbf{v} \in \mathbb{P}_\varepsilon$, then $\mathbf{v}$ is said to be below or above the ε-pareto boundary, respectively.

In a slight abuse of notation, we sometimes say that the pure strategy solution $\mathbf{a} \in \mathbb{X}$ for $\mathbb{X} \subseteq \mathbb{R}^n$ if $(r_1(\mathbf{a}), \ldots, r_N(\mathbf{a})) \in \mathbb{X}$.

We now formally define two desirable properties of a successful multi-agent learning algorithm for general-sum matrix games: *security* and *efficiency*. We begin with the *security* property, which concerns maximizing the minimum average payoffs one receives.

Let the *security level* of action $a \in A_i$ (denoted $r_i^{\text{sec}}(a)$) be the minimum possible payoff agent i can receive when it plays action a. Formally, $r_i^{\text{sec}}(a) = \min_{\mathbf{a}_{-i}} r_i(a, \mathbf{a}_{-i})$. The pure-strategy *maximin value* for agent i (denoted r_i^{mm}) is its best case security level, given by

$$r_i^{\text{mm}} = \max_{a \in A_i} r_i^{\text{sec}}(a). \tag{1}$$

Similarly, the pure-strategy *maximin action* is given by

$$a_i^{\text{mm}} = \arg\max_{a \in A_i} r_i^{\text{sec}}(a). \tag{2}$$

The maximin value over the mixed strategy space also exists and often provides a higher security level. However, throughout this paper, we use the term *maximin* to mean the pure-strategy maximin unless specified otherwise.

Definition 6. *(Secure) We say that a learning algorithm is* secure *if the limit of its average payoffs is nearly as high as its maximin value with high probability, regardless of the behavior of its associates. Formally, a learning algorithm is secure if, with high probability,*

$$\lim_{t \to \infty} \frac{1}{t} \sum_{\tau=0}^{t} r_i^\tau \geq r_i^{\text{mm}} - \varepsilon,$$

where ε is some small positive constant.

We make two observations about this notion of security. First, any algorithm that learns to play its maximin action is secure. Second, this notion of security is not as strong as other notions of security.

Alternatives such as the maximin solution over mixed strategies yield higher levels of security in some games. However, direct computation of the maximin solution over the mixed strategy space requires knowledge of the payoff structure of the game. This information is not available in the minimal information environments we consider in this paper. Nevertheless, it is possible to learn to play strategies that guarantee a payoff close to the maximin value in such environments (e.g. Exp3 [1]), but doing so appears to negate an agent's ability to be *efficient* (see Section 4.2.4).

Definition 7. *(Efficient) We say that a learning algorithm is* efficient *if it learns to play an ε-PE solution in self-play with high probability.*

As with the definition of security, there are refinements of the notion of efficiency that are more sophisticated and strict [9, 29]. However, despite these limitations, we know of no existing learning algorithm in the literature that is both secure and efficient in a large set of interesting general-sum matrix games played in minimal information environments. We now give a brief overview of multi-agent learning algorithms and discuss their capabilities with respect to efficiency and security in general-sum games played in minimal information environments.

2.3 Related Literature

We restrict attention to five categories: belief-based learning, reinforcement learning, no-regret algorithms, principle-based multi-agent learning, and satisficing learning.

Belief-based agents construct models of the strategies employed by their associates. They then use these beliefs to calculate an optimal policy with respect to those beliefs. Many such algorithms exist, the most studied of which are fictitious play [14] and, more generally, Bayesian learning. Fictitious play algorithms model the strategies of associates by recording the empirical distribution of their actions and play a best response with respect to this model. Fictitious play provably converges in self play to an NE in zero-sum games [30], iterative dominance solvable games [26], 2xN games [3], and potential games [25]. However, it is known to not converge in other games such as Shapley's game [14]. Since minimal information environments do not allow an agent to see the actions of associates, belief-based agents of this form cannot be used in these environments.

In the past decade, *reinforcement learning* [40] has been studied extensively in multi-agent domains. In particular, Q-learning [42] has been studied in these contexts [31], as well as several extensions and variations on this algorithm [4, 6, 16, 18, 21, 22]. While effective in many situations, these algorithms often learn strategies that produce low (inefficient) payoffs in many important games. Games in which these algorithms tend to perform poorly are games in which agents must learn to compromise in order to be successful. Such compromises typically require agents to learn to play non-myopic solutions. Recently, several multi-agent reinforcement learning algorithms have been developed that focus on learning pareto efficient and fair solutions in general-sum games. Some of these efforts include [9, 32]. While basic Q-learning satisfies the restrictions of minimal information environments, adaptations to multi-agent environments typically do not (with the exception of WoLF-PHC [6]). As such, they cannot be used successfully in minimal information environments.

No-regret algorithms (also known as universally consistent algorithms [14]) are becoming increasingly popular in the literature. Loosely, *regret* compares an algorithm's performance with that of its best strategy (or expert [20]). An algorithm is said to have *no regret* if it performs at least as well (in the limit) as its best strategy. A formal description is given by Foster and Vohra [13].

Most no-regret algorithms found in the literature assume knowledge of the payoff matrix and actions of associates. Thus, they are not suitable for minimal information environments. There are some no-regret algorithms, however, that do not require these assumptions (e.g. Exp3 [1]). Regardless, like many other multi-agent learning approaches, no-regret algorithms tend to learn myopic strategies in important repeated games since they tend to ignore how the other agents will react to their actions [11]. As a result, they learn inefficient (and unsuccessful) strategies in many important games, though they are secure (e.g. GIGA-WoLF

1. Set initial aspirations (α_i^0)

2. Repeat
   (a) Select an action a_i^t

$$a_i^t \leftarrow \begin{cases} a_i^{t-1} & \text{if } (r_i^{t-1} \geq \alpha_i^{t-1}) \\ \text{rand}(A_i) & \text{otherwise} \end{cases}$$

   where rand(A_i) is a random selection from A_i.

   (b) Receive reward r_i^t and update aspiration level:

$$\alpha_i^{t+1} \leftarrow \lambda_i \alpha_i^t + (1 - \lambda_i) r_i^t$$

Algorithm 1: The S-algorithm for agent i.

[5]). Recently, several less-myopic no-regret algorithms have been developed (such as work by Chang and Kaelbling [7]), but these approaches require additional information about the game structure and the agent's associates; thus, they cannot be used in minimal information environments.

Another recent trend in multi-agent learning has been to learn and develop *principles* that agents should follow to be successful. These algorithms focus intently on establishing good reputations, teaching other agents, and sending informative signals [10, 23, 29, 32]. These algorithms require knowledge of both the payoffs and actions of other agents. As such, they cannot be used in minimal information environments.

Thus, none of these algorithms are both efficient and secure in repeated general-sum games played minimal information environments. In the next section, we describe the S-algorithm, which, while conceptually very simple, shows great potential for achieving both efficient and secure behavior in general-sum games.

3 The S-Algorithm

The original S-algorithm was formalized by Karandikar *et al.* [19] for a small set of 2-player, 2-action repeated matrix games. It was extended by Stimpson and Goodrich to n-player, m-action games in [36]. It is this version of the S-algorithm, shown in Algorithm 1, that we consider in this paper.

An S-algorithm agent (or S-agent) encodes an *aspiration level* α_i^t, which represents its target payoff in episode t. Based on this aspiration level, the S-agent selects its next action using a satisficing rule. If the agent's payoff in the previous episode met its aspiration level (i.e., if $r_i^{t-1} \geq \alpha_i^{t-1}$), then it repeats its previous action (i.e., $a_i^t = a_i^{t-1}$). Otherwise, it chooses its next action randomly (according to a uniform probability distribution over its actions). An important dynamic of the S-algorithm is that once it finds an action that always meets or exceeds its current aspiration level, it will continue to play that action from then on. When this happens, we say that the S-agent has *converged*.

Once the S-agent receives its payoff in episode t (denoted r_i^t), it updates its aspiration level using the following update rule:

$$\alpha_i^{t+1} \leftarrow \lambda_i \alpha_i^t + (1 - \lambda_i) r_i^t, \tag{3}$$

where $\lambda_i \in (0, 1)$ is the agent's learning rate. Thus, the agent's aspiration level α_i^t is really just a weighted average of the agent's past payoffs which implicitly encodes a level of sustainable average reward. In short, the S-algorithm is a simple procedure for learning a sustainable average reward and an action for obtaining that reward.

Since the property of efficiency involves self play, it is useful to talk about the joint behavior resulting from interactions between multiple S-agents. To do this, we define several relevant terms. Let $\boldsymbol{\alpha}^t = (\alpha_1^t, \cdots, \alpha_N^t)$ be the agents' joint aspirations at time t. A solution $\boldsymbol{\pi}$ is said to be *mutually satisficing* at time t if $r_i(\boldsymbol{\pi}) \geq \alpha_i^t$ for all i. Let $S_i(\alpha_i)$ be the set of pure strategy solutions that are satisficing to agent i given its aspiration

level α_i. The *mutually satisficing set* is $S(\boldsymbol{\alpha}) = S_1(\alpha_1) \cap S_2(\alpha_2) \cap \cdots \cap S_N(\alpha_N)$, which may be empty for a given vector of aspirations.

The behavior of the S-algorithm in self play is governed by the trajectory of the aspiration vector through the game's payoff space and this vector's relationship to the various payoff vectors of the game. A steady state is reached when a solution enters the mutually satisficing set and is played by the agents, since such an event causes the agents to converge to that solution (this phenomena is known as a satisficing equilibrium [38]). Thus, in self play, the joint behavior of S-agents can be described as a search through the joint action space for a mutually desirable solution.

The S-algorithm has been analyzed in a public goods game called the multi-agent social dilemma (MASD), an n-player, m-action version of the prisoner's dilemma [36]. In this game, the S-algorithm is efficient if each agent's initial aspiration level (α_i^0) is sufficiently high. Furthermore, the S-algorithm is secure in this game when playing with (static) defecting agents provided that its initial aspirations are as high as its security level (i.e., $\alpha_i^0 \geq r_i^{\text{mm}}$). In the next section, we address the efficiency and security of the S-algorithm in repeated general-sum games.

Additionally, in Section 5, we introduce a trembling-hand version of the S-algorithm, which we call SALT (*S-a*lgorithm with *L*earned *T*rembles). Karandikar *et al.* introduced aspiration trembles in their original version of the S-algorithm, which they applied to a limited set of 2-player, 2-action games [19]. Unlike Karandikar's algorithm, SALT learns to tremble its aspirations to regions of the reward space that produce high payoffs. SALT is provable secure in all games against all associates, and it is efficient in a most general-sum matrix games.

4 Properties of the S-Algorithm

We begin by analyzing the S-algorithm with respect to the security property in repeated general-sum games. We then address the efficiency property in Section 4.2.

4.1 Security

Recall that security refers to an agent's ability to ensure that its average payoffs, in the limit, are not substantially less than its pure-strategy maximin value, regardless of the game the agent is playing or with whom it associates. We first show that the S-algorithm is secure in all repeated general-sum games when its associates employ static pure strategies. We then address security in the general case.

4.1.1 Security Against Pure Strategies?

Stimpson and Goodrich showed that the S-algorithm learns a secure solution in the MASD when it plays with agents that play the single-shot NE of the game [36]. The reason that the S-algorithm cannot be exploited by those specific agents in the MASD is that such attempts cause the S-algorithm's average payoff to be low. Since the aspiration level is a weighted average of previous payoffs, low payoffs imply that the aspiration level must also decrease. When the aspiration level is less than the maximin value, the S-algorithm will likely become satisfied with the maximin solution if it learns slowly enough. Thus, if an agent's associates are playing any pure strategy solution and if the agent begins with high aspiration levels, it is likely that the agent will learn to play a best response pure strategy solution against such associates.

A critical factor in determining the probability that the S-algorithm will select a particular action is the amount of time it takes for the aspiration level to drop a certain amount. Let $T(\lambda_i, \varepsilon)$ be the minimum number of episodes before agent i's aspiration level has deviated by ε units (i.e., $|\alpha_i^{T(\lambda_i,\varepsilon)+t} - \alpha_i^t| \leq \varepsilon$). A variation of the following lemma was first proved in [19].

Lemma 1. *As* $\lambda_i \to 1$, $T(\lambda_i, \varepsilon) \to \infty$ *for all* $\varepsilon > 0$.

Proof: Let t_0 denote the time when the algorithm begins. The aspiration level of the S-algorithm after T episodes depends on the sequence of rewards received by the agent during those episodes. Formally,

$$\alpha_i^{t_0+T} = \lambda_i^T \alpha_i^{t_0} + (1-\lambda_i) \sum_{\tau=0}^{T-1} \lambda_i^\tau r_i^{t_0+T-\tau-1}. \tag{4}$$

Since the set of possible rewards, $\{r_i\}$, is bounded, the difference between $\alpha_i^{t_0}$ and $\alpha_i^{t_0+T}$ approaches zero as λ_i approaches unity. This implies that as $\lambda_i \to 1$, $T(\lambda_i, \varepsilon) \to \infty$. □

If the S-algorithm begins with high initial aspirations and if the S-algorithm converges, then it is easy to show that (a) the algorithm will likely converge on a pure-strategy best-response solution that is at least as good as the maximin action (a_i^{mm}) and (b) this likelihood can be made arbitrarily high by exploiting Lemma 1. Convergence, however, can only be guaranteed for an arbitrary matrix game if the behavior of the S-agent's associate(s) is at a fixed pure-strategy. We formalize this observation in the following lemma. The proof of the lemma follows the pattern given in [36], which was restricted to the MASD.

Lemma 2. *If the S-algorithm converges to a solution, there is a finite probability that it will converge to a solution that yields a payoff no less than its maximin value provided that*

(a) associates play a fixed pure-strategy and
(b) $\alpha_i^0 \geq r_i^{\text{mm}}$.

This probability can be made arbitrarily high by increasing the learning rate λ_i.

Proof. Assume that agent i's associates play the fixed strategy $\mathbf{a}_{-i}$, and assume that $\alpha_i^0 \geq r_i^{\text{mm}}$. The S-algorithm converges if, at some time t, it plays an action a_i^* for which $r_i(a_i^*, \mathbf{a}_{-i}) \geq \alpha_i^t$. When α_i^t is greater than r_i^{mm}, the only actions to which the S-algorithm will converge against all associate behaviors must satisfy $r_i(a_i^*, \mathbf{a}_{-i}) \geq \alpha_i^t > r_i^{\text{mm}}$, meaning that the S-agent would be secure. When α_i^t is less than r_i^{mm}, there is a finite probability that the unsatisfied agent will play a_i^{mm}; this probability is equal to $\frac{1}{|A_i|}$. Since initial aspirations begin above the maximin value, if aspirations have dropped so that α_i^t is less than r_i^{mm}, there must have been a period of time when a_i^{mm} is the only satisficing action[1]. The amount of time that the maximin action remains the only satisficing solution can be made arbitrarily large by increasing the learning rate; see Lemma 1. Thus, the probability that the S-algorithm will converge to a_i^{mm} can be made arbitrarily high since the S-algorithm plays randomly when it is not satisfied. □

The S-algorithm is secure if it converges to the maximin solution. Thus, if the S-algorithm converges and if its associates play fixed pure-strategies, then the S-algorithm is secure. However, these assumptions are strong; they do not apply when the agent associates with non-static agents. We now consider this more general case.

4.1.2 Security Against All Agents?

An S-agent can be exploited in two ways: (1) if it converges to an action that gives it a payoff significantly less than its maximin value (r_i^{mm}) or (2) if the S-agent can be kept from (permanently) converging while its average payoffs are significantly less than r_i^{mm}. Since Lemma 2 shows that the S-agent converges with high probability (controllable by λ_i) to its maximin action when its aspirations fall below its maximin value (and, thus, is secure), we focus attention on the latter possibility.

To exploit an S-agent by keeping it from (permanently) converging to a secure action, an opponent can employ the *pumping* method. In the pumping method, the opponent exploits the S-agent until the S-agent becomes satisfied with its maximin action (i.e., the S-agent's aspirations fall below its maximin value, after which it plays its maximin action). The opponent then allows the S-agent to receive payoffs higher than its

[1]Note that the maximin action may actually represent an equivalence class of actions that have a security level equal to the maximin value.

	a	b
A	$(4, -4)$	$(-9, 9)$
B	$(3, -3)$	$(2, -2)$

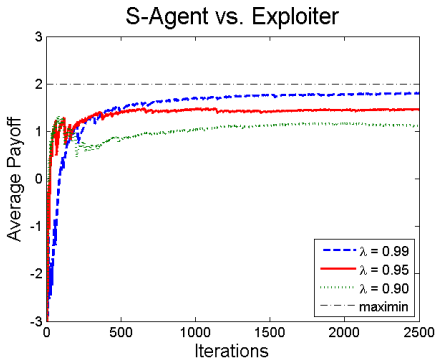


Figure 2: (Left) A zero-sum 2×2 payoff matrix. The payoff to the row player (player 1) is listed first, followed by the payoff to the column player (player 2). (Right) Average payoffs for the row player (an S-agent) against an agent designed to exploit it.

maximin value, which will result in the S-agent's aspirations rising again above its maximin value. At this point, the S-agent will again become dissatisfied with its maximin action, and it will again begin to search for another satisfying action. This, again, makes the S-agent vulnerable to exploitation, and the process repeats itself.

Exploiting the S-agent by *pumping* is made possible by the fact that an S-agent's aspiration levels are a weighted average of its payoffs, with more recent payoffs given higher weight. Thus, an S-agent can be led to believe that it can sustain a payoff as high as its recent payoffs rather than its long-term average payoffs. However, the amount that the S-agent can be exploited using pumping is reduced if the S-agent uses a slower learning rate (i.e., higher λ_i). This is because a slower learning rate causes the aspiration level to be more indicative of average payoffs.

This trend is demonstrated in Figure 2, which shows the average payoffs of an S-agent over time when it plays the particular zero-sum game shown against an exploiter agent. In the game, the S-agent (the row player) can guarantee itself a payoff of at least 2 if it plays B. Thus, $r_i^{\text{mm}} = 2$. Consider now the situation in which the exploiter (the column player) can view everything about the S-agent, including the action the S-agent is going to play and its current aspiration level. In this case, the exploiter can coax the S-agent to play A (using the pumping methodology just discussed), at which point the exploiter plays b, giving the S-agent a payoff of -9. When the exploiter employs the pumping method for exploiting the S-agent in this game, the S-agent's average payoffs are plotted in Figure 2 for various values of λ_i. The figure shows that as $\lambda_i \to 1$, the S-agent's average payoffs approach $r_i^{\text{mm}} = 2$. This is because slower learning rates (i.e., higher λ_i) cause the agent's aspirations to more closely reflect its average rewards due to the nature of Eq. (3). The closer the S-agent's payoffs reflect its actual long-term payoffs, the less susceptible it is to being exploited by pumping. Thus, in addition to controlling the probability to which the S-agent converges to a secure solution (see Lemma 2), λ_i can, in some situations, help control the amount of possible loss incurred by pumping.

However, the S-agent only becomes satisfied with its maximin action with high probability when its aspirations fall below its security level. This means that if the number of cycles of pumping grows high, it is likely that the S-agent will become perpetually satisfied at some point with an action that has a security level less than its maximin value minus ε. When this happens, the exploiter can cease to use the pumping method, and the S-agent will be perpetually exploited.

For example, consider the 3×2 zero-sum matrix game shown in Figure 3 and let the S-agent be the row player. The S-agent's maximin value is 2, as it can guarantee itself a payoff of at least 2 if it always plays B. However, if its aspiration level ever drops to 1.8, it will converge to action C (resulting in a payoff of 1.8) if it plays C and its opponent plays a (and continues to play it thereafter). If the opponent employs pumping, the S-agent will eventually converge to action C, resulting in it being exploited indefinitely (assuming $\varepsilon < 0.2$).

In the previous example, an omniscient opponent could exploit the S-agent in some games. However,

	a	b
A	$(-9, 9)$	$(4, -4)$
B	$(3, -3)$	$(2, -2)$
C	$(1.8, -1.8)$	$(1, -1)$

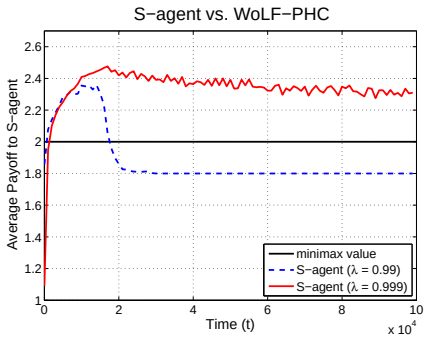


Figure 3: (Left) 3×2 zero-sum matrix game. (Right) Payoffs over time to the S-agent (the row player) against WoLF-PHC [6] in the matrix game shown at left for $\lambda_i = 0.99$ and $\lambda_i = 0.999$. WoLF-PHC was implemented with the parameter values shown given in Table 4.

if we assume that opponents are limited to the same minimal information environments as the S-agent, exploitation of the S-agent becomes more difficult. Nevertheless, simple learning algorithms can learn to exploit an S-agent in some games. For example, when an S-agent plays WoLF-PHC [6] in the game shown in Figure 3, WoLF-PHC learned to exploit the S-agent when $\lambda_i = 0.99$ (Figure 3). In this case, the S-agent does not converge since WoLF-PHC learns to play a mixed strategy.

We note that when the S-agent uses a slower learning rate (i.e., $\lambda_i = 0.999$), Figure 3 shows that it avoids being exploited (in fact, it does better than its maximin value). However, it is possible that WoLF-PHC could learn to exploit the S-agent with the slower learning rate if it adjusted its own learning parameters. In short, while the S-algorithm can often avoid being exploited by lowering its learning rate, doing so does not guarantee secure play. In Section 5, we introduce an extended version of the S-algorithm that is secure. Before doing so, however, we analyze the S-algorithm with respect to efficiency.

4.2 Efficiency

We now shift focus from security to efficiency by considering how the S-algorithm behaves in self play in repeated general-sum games. The key principles that govern the behavior of the S-algorithm in self play are the same as when the S-algorithm is playing other agents:

- Slower learning rates (for all agents) cause the joint aspirations of the agents to remain in a particular region of the reward space for long periods of time; see Lemma 1.
- When the joint aspirations stay in a particular region for a long period of time, it is likely that all possible joint actions will be attempted by the agents before the joint aspirations leave the region.
- If aspirations enter a desirable region of the aspiration space, the probability that the S-agents learn an efficient solution is controllable by the agents' joint learning rates $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_n)$.

These observations have been tested empirically [37] and analyzed theoretically [35] in the prisoner's dilemma and in the MASD. It is useful to extend this previous work to repeated general-sum games.

4.2.1 Conditions of Efficiency in General-Sum Matrix Games

In this subsection, we show that the S-algorithm can be efficient in any general-sum matrix game for which certain technical conditions, called *the conditions of pareto efficiency*, are met. Even though these conditions are defined in terms of aspiration levels, it is important to note that these conditions are characteristics of the game itself, not of the learning algorithm *per se*. We begin by defining these conditions and then discuss the games in which these conditions can be satisfied. In Section 5, we present a trembling-hand version of the S-algorithm that allows the agents to learn how to satisfy these conditions.

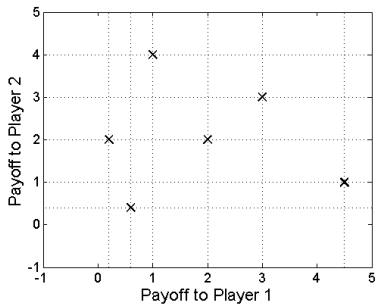


Figure 4: The payoff space $\mathbb{R}^n$ is partitioned into subregions defined by the satisficing sets.

The choices and learning behavior of S-agents are dictated by the relationship between the agents' aspiration levels and the payoff structure of the game. As joint aspirations vary through their range of possible values, there are regions of the aspiration (or reward) space for which the resulting sets of satisficing actions are invariant. Thus, the mutually satisficing sets effectually divide the payoff space $\mathbb{R}^n$ into contiguous disjoint *subregions* in which $\mathbb{W}$ is a subregion of $\mathbb{R}^n$ if, for all $i \in I$, $S_i(\alpha_i)$ is constant for all $\boldsymbol{\alpha} \in \mathbb{W}$. Figure 4 illustrates how the satisficing sets partition $\mathbb{R}^n$ for an arbitrary 2-agent game. In the figure, each $\mathbf{x}$ (or joint payoff) represents a solution of the game. Each rectangle in the figure is a separate subregion of $\mathbb{R}^n$.

When a game has at least one subregion that satisfies a set of desirable properties, then it is possible for S-agents to converge on an *efficient* solution in these games with high probability controllable by $\boldsymbol{\lambda}$. Formally, we say that a game satisfies the *conditions of pareto efficiency* if there exists a subregion $\mathbb{W}$ that has the following three properties:

Property 1. $\forall \boldsymbol{\alpha} \in \mathbb{W}$, $S(\boldsymbol{\alpha}) \neq \emptyset$. *In words, the set of mutually satisficing solutions (or joint actions) is nonempty for all joint aspirations in the subregion.*

Property 2. $\forall \mathbf{a} \in S(\boldsymbol{\alpha})$, $\mathbf{a} \in \mathbb{P} \cup \mathbb{P}_\varepsilon$. *In words, all mutually satisficing solutions are ε-PE for all joint aspirations in the subregion.*

Property 3. $\forall \boldsymbol{\alpha} \in \mathbb{W}$, $\exists \mathbf{a} \in S(\boldsymbol{\alpha})$ *for which* $Pr(\mathbf{a}) > 0$. *In words, there is a nonzero probability that at least one mutually satisficing solution will be played for all joint aspirations in the subregion.*

In summary, the S-algorithm and payoff matrix implicitly define subregions of the aspiration space. If there exists a subregion that satisfies the above properties (or conditions) and if the joint aspirations enter that subregion, then the S-algorithm (in self play) will behave efficiently. In the remainder of this section, we identify what kinds of games allow these conditions to be satisfied.

Let $\mathcal{R}_{P_1}$ and $\mathcal{R}_{P_2}$ denote the set of subregions that satisfy properties 1 and 2, respectively. The intersection of these two sets consists of joint aspirations for which the set of mutually satisficing solutions is both non-empty and ε-PE. Importantly, there always exists a region within this intersection.

Lemma 3. $\mathcal{R}_{P_1} \cap \mathcal{R}_{P_2} \neq \emptyset$.

Proof: Every subregion that contains a solution of the matrix game is in $\mathcal{R}_{P_1}$. Thus, $\mathcal{R}_{P_1} \neq \emptyset$. Additionally, all games have at least one solution in $\mathbb{P}$, so the subregion containing this solution is also in $\mathcal{R}_{P_2}$. □

Thus, the first two properties are true for at least one subregion in every matrix game. To understand when the third property holds for this subregion or some other subregion that also satisfies properties 1 and 2, we introduce the notion of a *critical event* (CE).

	a	b
A	$(3,2)$	$(3,1)$
B	$(4,3)$	$(1,4)$

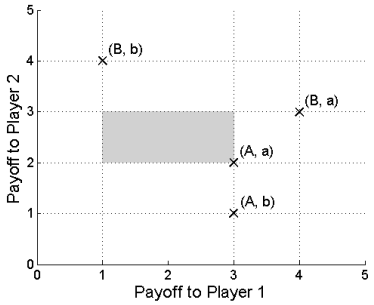


Figure 5: (Left) A 2×2 payoff matrix. The payoff to the row player (player 1) is listed first, followed by the payoff to the column player (player 2). Thus, if the row player plays A and the column player plays b, then player 1's payoff is 3 and player 2's payoff is 1. (Right) The game's payoff space.

4.2.2 Critical Events

A payoff matrix and a group of learning agents form a dynamic system. When the learning agents use the S-algorithm, then the choices made by the agents follow a stochastic, nonlinear evolution depending on the internal parameters (aspirations) of the agents. We use the term *critical event* (CE) to denote the nonlinearities in the choice and update dynamics. For the learning and choice dynamics of the S-algorithm, a CE occurs when a choice made by some subset of agents ensures that one or more solutions cannot be played while aspirations remain in the current subregion.

Formally and without loss of generality, let $t = 0$ denote the time when joint aspirations $\boldsymbol{\alpha}$ enter some subregion $\mathbb{W}$. At time $t = T$, the joint aspirations leave the subregion $\mathbb{W}$.

Definition 8. *(Critical Event) The joint action $\mathbf{a}^\tau(J)$ played by some subset of agents, $J \subseteq I$ and $J \neq \emptyset$, is a CE in episode τ if there is a joint action $\mathbf{b}$ for which the probability of choosing $\mathbf{b}$ is greater than zero for $t < \tau$ but the probability of choosing $\mathbf{b}$ is zero for all $t \in (\tau, T)$.*

Thus, a CE is a choice made by some subset of agents that *excludes the solution* $\mathbf{b}$ for all time unless the aspirations leave the subregion $\mathbb{W}$. The set of CEs in some subregion $\mathbb{W}$ is the set of all $a^t(J)$ that can exclude any solution while $\boldsymbol{\alpha} \in \mathbb{W}$. The *set of possible CEs is invariant across a subregion*, meaning that the same set of CEs can occur for each vector in a subregion. We can now reinterpret property 3 using the definition of a CE to mean that no CE or set of CEs can exclude all mutually satisficing solutions.

There are two different categories of CEs: *satisficing* and *preclusion* events. We discuss each category in turn.

Satisficing CEs. The set of satisficing CEs include all joint actions that cause a subset of agents to "lock-onto" their portion of this choice until the aspirations leave the subregion. Satisficing CEs include *selection*, *security*, and *collusion* CEs. A selection CE involves all agents, a security CE involves just one agent, and a collusion CE involves a proper, non-singleton subset of agents.

Selection CEs. A selection CE occurs when the agents play a mutually satisficing solution. Since the agents will continue to play a mutually satisficing solution indefinitely once it is played, a selection CE excludes all other solutions.

Security CEs. Security CEs are illustrated by the game shown in Figure 5. In this game, if joint aspirations are in the shaded subregion (or if the row player's aspiration level is 3 or less), a security CE occurs when the row player plays A. If the row player plays A, it will continue to do so from then on regardless of the column player's actions. Thus, the joint actions (B, a) and (B, b) will be excluded.

If joint aspirations are in a subregion for which $\alpha_i^t \leq r_i^{\text{sec}}(a_i)$ (i.e., agent i's aspirations are greater than action a_i's security level), then a_i is a security CE since, once it is played, agent i will continue to play it until joint aspirations leave the current subregion. This CE excludes every solution in which agent i does

	a	b	c
A	$(0, 3)$	$(3, 0.5)$	$(3, 0)$
B	$(0.5, 3)$	$(4, 4)$	$(1.5, 3)$
C	$(3, 1.5)$	$(3, 1)$	$(1, 3)$

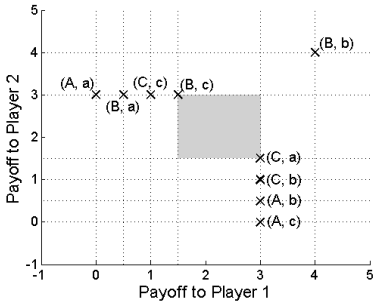


Figure 6: Situation in which a preclusion CE can occur.

not play a_i. This is equivalent to stating that the other agents are now using a payoff matrix which is one dimension smaller than before agent i became satisfied.

Importantly, when initial aspirations are set high, the first possible security CE that each agent can play is its *maximin action*. If agent i's maximin action is not a security CE in a given subregion, then neither are any of its other actions (provided, of course, that no other CEs have occurred).

Collusion CEs. Collusion CEs span the space between security and selection CEs. They differ from security and selection CEs only in the cardinality of the set J. In collusion CEs, J is a proper subset of I, but it is not a singleton (i.e., $1 < |J| < |I|$).

Preclusion CEs. Satisficing CEs (a) reduce the cardinality of the set of joint actions that will be played and (b) reduce the dimensionality of the payoff matrix that will be played. On the contrary, preclusion CEs only reduce the cardinality of the set of joint actions that will be played; they do not reduce the effectual dimensionality of the payoff matrix.

For example, consider the 2-player, 3-action matrix game shown in Figure 6. Now suppose that the joint aspiration vector is in the shaded subregion of the figure. In this case, all joint actions except (B, b) are preclusion CEs since they exclude the joint action (B, b). For example, if the joint action (C, a) is played, then the row player will continue to play action C until the column player plays action c, at which point the column player will be satisfied (receiving a payoff of 3). Thus, the column player will continue to play c until the row player plays action A. At this point, the row player (who, again, gets a payoff of 3) will be satisfied until the column player plays action a, meaning that the column player will be satisfied until the row player plays action C. This cycle will repeat while the joint aspirations remain in the shaded subregion. At no time during the cycle will the action (B, b) be played.

Thus, the number of playable joint actions decreases, but the dimensionality of the playable space of the game matrix remains the same. This is because at least one agent is satisfied with each non-excluded joint action, but none of the agents is satisfied with all non-excluded joint actions. Additionally, no agent is satisfied with all of its payoffs associated with one of its (non-excluded) actions.

There are two kinds of preclusion CEs: *cyclic* and *acyclic*. Cyclic preclusion CEs require that a cycle emerge in which one subset of agents is temporarily satisfied, followed by a different subset of agents being satisfied, etc., until the initial subset of agents is again temporarily satisfied. The game shown in Figure 6 is a cyclic preclusion CE.

Acyclic preclusion CEs differ from cyclic preclusion CEs in that they do not create a cycle in agent play. Rather, the cardinality of the search space is reduced by an event that essentially "funnels" the agents' play into a second CE (of any kind). For example, consider the matrix game shown in Table 1, which differs from the game shown in Figure 6 only in the payoffs in cell (C, c). In this game, suppose that $\boldsymbol{\alpha}^t = (2.5, 2.5)$, and that the joint action played at time t is (A, c). In this case, the row player would be satisfied, whereas the column player would not be. Thus, the row player would repeat action A until the column player played a. The column player would then play a until the row player played C, which is a security CE, resulting in the row player playing C thereafter. Thus, the acyclic preclusion CE (A, c) excludes the solutions (B, B)

	a	b	c
A	$(0, 3)$	$(3, 0.5)$	$(3, 0)$
B	$(0.5, 3)$	$(4, 4)$	$(1.5, 3)$
C	$(3, 1.5)$	$(3, 1)$	$(3, 1)$

Table 1: Game containing an acyclic preclusion CE.

and (B, c) due to the imminent security CE that will follow. Thus, acyclic preclusion CEs can only occur if another CE is possible in the current subregion of the aspiration space.

We note that satisficing and preclusion CEs cover all possible nonlinearities in the dynamical system. These nonlinearities occur when the payoffs of the game are restricted to a sub-portion of the payoff matrix.

4.2.3 Theoretical Results

We first state and prove sufficient properties for games that satisfy the conditions of pareto efficiency. We then discuss the frequency and relevance of those games that do not possess those properties.

Theorem 1. *All general-sum matrix games in which collusion and cyclic preclusion CEs do not occur in all $\mathbb{W} \in \mathcal{R}_{P_1} \cap \mathcal{R}_{P_2}$ have a subregion that satisfies the conditions of pareto efficiency.*

Proof. By definition, the absence of CEs means that all solutions in the payoff matrix have a non-zero probability of being played (i.e., in which case Property 3 from Section 4.2.1 holds). We must now consider whether a satisficing or security CE can exclude all mutually satisficing ε-PE solutions. Note that we need not consider acyclic preclusion CEs since the existence of an acyclic preclusion CE implies that a satisficing or security CE could exclude the same solutions without the acyclic preclusion CE.

Consider first whether a selection CE can exclude every mutually satisficing solution within all subregions $\mathbb{W} \in \mathcal{R}_{P_1} \cap \mathcal{R}_{P_2}$. Selection CEs only occur when all agents are satisfied with an action. According to Property 2, all mutually satisficing actions are ε-PE when $\boldsymbol{\alpha} \in \mathbb{W}$, so a selection CE cannot exclude all mutually satisficing solutions.

Consider now whether a security CE can exclude every mutually satisficing solution within all subregions $\mathbb{W} \in \mathcal{R}_{P_1} \cap \mathcal{R}_{P_2}$. More precisely, we are interested in knowing if it is possible for a security CE to occur while aspirations are in $\mathbb{W}$ that prevents property 3 from holding. A security CE occurs when an agent selects an action that has a security level above its aspiration level. Since the maximin action, a_i^{mm}, is the action that has maximum security, if any other action can trigger a security CE within a region, then so can the maximin action. Thus, we will restrict attention to the security CE associated with an agent playing its maximin action.

Without loss of generality, suppose that agent i is capable of playing a security CE within some region $\mathbb{W} \in \mathcal{R}_{P_1} \cap \mathcal{R}_{P_2}$. By definition of a subregion, it follows that $\alpha_i \leq r_i^{\text{mm}}$ for all $\boldsymbol{\alpha} \in \mathbb{W}$. Since the security CE occurred in $\mathbb{W} \in \mathcal{R}_{P_1} \cap \mathcal{R}_{P_2}$, this implies that a_i^{mm} must be part of some joint action $\mathbf{a}$ for which the joint payoff vector $(r_1(\mathbf{a}), \ldots, r_N(\mathbf{a})) \in \mathbb{W}$ for $\mathbb{W} \in \mathcal{R}_{P_1} \cap \mathcal{R}_{P_2}$. Otherwise, there is a solution that ε-pareto dominates $(a_i^{\text{mm}}, \mathbf{a}_{-i})$ for all $\mathbf{a}_{-i}$, so $\mathbb{W} \notin \mathcal{R}_{P_1} \cap \mathcal{R}_{P_2}$, which is a contradiction.

When a_i^{mm} is played and the security CE occurs, the payoff matrix is, in effect, restricted to an $N-1$ dimensional sub-matrix that includes the joint action $\mathbf{a} \setminus a_i^{\text{mm}}$. Thus, the pareto efficient and mutually satisficing action $\mathbf{a}$ is not excluded when agent i plays its maximin action, which implies that property 3 holds for this action (provided that no collusion or cyclic preclusion CEs occur). □

Theorem 1 shows that, in the absence of collusion and cyclic preclusion CEs, there is at least one subregion that satisfies the conditions of pareto efficiency in all repeated general-sum matrix games. Two questions related to the technical restrictions imposed by the theorem naturally follow. First, how many games do not have collusion and cyclic preclusion CEs when aspirations are in these subregions? Second, how likely

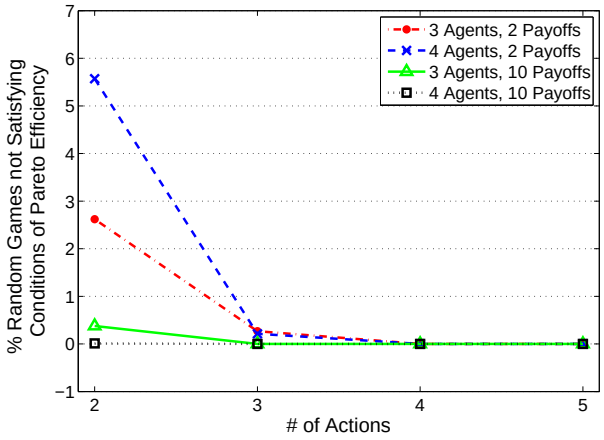


Figure 7: Percentage of randomly generated games for which no initial aspiration vectors satisfy the conditions of pareto efficiency. Percentages are given for games with varying numbers of actions (x-axis), players, and unique possible payoffs. Graphs for 2-player games are not shown since there always exists a set of aspiration vectors that satisfy the conditions of pareto efficiency; see Corollary 1.

is it that S-agents will set their individual aspiration levels so that the joint aspiration vector will find these subregions?

In response to the first question, we show that an overwhelming majority of general-sum matrix games satisfy the technical conditions of the theorem. We begin by providing a corollary to Theorem 1, whose proof is found in [8].

Corollary 1. *All 2-player general-sum matrix games have a subregion that satisfies the conditions of pareto efficiency.*

While the corollary is restricted to 2-player games, we emphasize the significance of this achievement. The majority of theoretical results in the multi-agent learning literature are restricted to subclasses of 2-player games. For example, fictitious play has only been shown to converge in self play to an NE in a limited set of 2-player games (zero-sum games [30], iterative dominance solvable games [26], 2xN games [3], and potential games [25]). This set of games excludes many important 2-player games. Thus, that the technical conditions of Theorem 1 are satisfied in all 2-player general-sum matrix games is a significant result.

Additionally, the technical conditions of Theorem 1 are satisfied in a vast majority of games with more than two players. While it is difficult to mathematically define these games due to the nonlinear dynamics of the satisficing learning algorithm, we can investigate this issue empirically. In so doing, we observe that, indeed, a very small set of general-sum matrix games do not have subregions that satisfy the conditions of pareto efficiency. To see this, consider Figure 7, which shows the results of an empirical study in which we measured the percentage of randomly generated payoff matrices that do not have a subregion that satisfies the conditions of pareto efficiency for 3- and 4-player games.

The figure shows several trends. First, as the number of actions available to each agent increases, the percentage of randomly generated games that do not satisfy the conditions of pareto efficiency decreases. In fact, for 3-action games, only a fraction of a percent of random games do not satisfy the conditions of pareto efficiency (for both 3- and 4-player games). For games with four or more actions, the percentage of games that do not satisfy the conditions of pareto efficiency is very small (indeed, our empirical study found no such games). This is because the average number of ε-PE solutions increases with the number of actions in the game, making it likely that at least one of these solutions cannot be excluded by CEs for some initial

aspiration vector.

Second, for games with only two unique payoffs, a lower percentage of games satisfy the conditions of pareto efficiency for 4-player games than 3-player games. For example, for games with two actions and two unique payoffs, increasing the number of players from three to four makes it more than twice as likely that the game will not satisfy the conditions of pareto efficiency. However, the trend is reversed for games with ten unique payoffs, as 3-player games are less likely to satisfy the conditions of pareto efficiency than 4-player games. These patterns are caused by two interacting principles: (1) more players increase the likelihood that two or more players will combine to form a collusion or preclusion CE, and (2) more unique joint payoffs (determined by numbers of agents and numbers of unique individual payoffs) make it more likely that at least one ε-PE solution cannot be excluded by a collusion or preclusion CE.

Third, Figure 7 shows that a randomly generated game with 10 possible unique payoffs is more likely to satisfy the conditions of pareto efficiency than a randomly generated game with only two unique payoffs. For example, for 3-player, 2-action games, games with two unique payoffs have a 2.62 percent chance of not satisfying the conditions of pareto efficiency, compared to just a 0.38 percent chance for games with ten unique payoffs.

Thus, despite the fact that not all general-sum matrix games satisfy the conditions of pareto efficiency, most do. Additionally, the set of matrix games commonly studied in the literature satisfy the conditions of pareto efficiency. For example, every finite-action matrix game in Stanford's GAMUT of games [28] has a subregion that satisfies the conditions of pareto efficiency. Thus, the conditions of pareto efficiency can be satisfied in a high percentage of games an S-agent is likely to encounter.

This leads to the second question: how likely is it that S-agents will (individually) set their initial aspirations so that the joint aspiration vector will enter an appropriate subregion given that such a subregion exists? We answer this question with an empirical study, in which we analyze the behavior of the S-agent in self play in a variety of general-sum matrix games.

4.2.4 Empirical Study

The empirical study has two parts. First, we explore the behavior of the S-algorithm in a public goods game [17] called the Multi-Agent Social Dilemma (MASD) [36, 37]. We then empirically analyze the performance of the S-algorithm (in self play) in a variety of two-player matrix games. As a point of reference, we compare the performance of the S-algorithm with a variety of learning agents suitable for use in minimal information environments.

Learning Behavior in the MASD. The MASD is a multi-player, multi-action extension of the celebrated prisoner's dilemma. It is of particular interest since successful behavior in this game requires that the algorithm learn to overlook short-term payoffs in favor of long-term payoffs. Additionally, the game allows us to determine how well a learning algorithm scales to increasing numbers of agents and actions.

In the MASD, each agent is faced with a decision of allocating M units of some discrete resource toward (a) some purely self-interested goal and (b) some group goal for all agents. Let a_i be the amount contributed by agent i toward the group goal G; $M - a_i$ is the amount contributed to the selfish goal S_i. Then, for each agent there are $M + 1$ possible values for $a_i \in \{0, 1, 2, ...M\}$. Let each agent's total utility be represented as a linear combination of the total amount contributed to the group goal G and the amount individually contributed to his or her own selfish goal S_i. Assuming that the relative, rather than absolute, utilities are important, we can reduce the number of parameters and describe the game as follows

$$R_i(a_i, a_{-i}) = \frac{(1 - kN)a_i + a_{-i}}{NM(1 - k)}, \tag{5}$$

where

$$a_{-i} = \sum_{j=1, j\neq i}^{N} a_j. \tag{6}$$

Algorithm	Parameters	Results
S-algorithm	$\lambda_i = 0.99$, high $\boldsymbol{\alpha}_0$	efficient not secure (if "pumped", secure otherwise)
Fixed (Non-adaptive)	$a_i = 0$ $a_i = M$ random Tit-for-Tat	Secure but not efficient Efficient but not secure Neither secure nor efficient. $\bar{R}_i = 0.5$. secure and efficient
Belief-based (Fictitious Play)	$\lambda_i = 0$ $\lambda_i \to \infty$	$\bar{a}_i = 0.5$, Random $\bar{a}_i \to 0$, Secure but inefficient.
Q-Learning	ϵ-greedy exploration, $\alpha = 0.2$, $\gamma = 0.9$. $N = 2$ $N = 3$ $N = 10$	 40% cooperation, 60% defection 35% cooperation, 65% defection 20% cooperation, 80% defection
WoLF-PHC	almost all reasonable parameters	Secure but inefficient
M-Qubed	ϵ-greedy expl, $\alpha = 0.1$, $\gamma = 0.95$, $\omega = 1$	Secure and efficient
No-regret (Exp3)	As required in [1]	Secure but inefficient

Table 2: Results and parameters of several algorithms in the MASD. Where appropriate, average rewards are given as $\bar{R}_i$ and the average action is given as $\bar{a}_i$.

It can easily be seen that when $a_i = 0$ (i.e., all resources are committed to the selfish goal) for all i, the payoff to each agent is zero. In addition, the payoff to each agent i when $a_i = M$ (i.e., the Nash bargaining solution [27] in which all resources are committed to the group goal) is always 1.

The key characteristic of the MASD is that it represents a situation similar to a prisoner's dilemma. Clearly, no matter what a_{-i} is chosen, agent i would receive the highest reward by choosing $a_i = 0$. Thus, a_i is strategically dominant. However, if all the other agents also make this choice, then all agents receive a reward of 0 which is the lowest cumulative reward for the group. On the other hand, if agent i chooses $a_i = M$ and the other agents fully cooperate, then all agents receive a reward of 1. However, by choosing $a_i = M$, agent i is exposed to possible exploitation; if the other agents all choose full defection ($a_{-i} = 0$), then agent i receives the minimum reward possible (which is less than 0). In addition to pure cooperation and defection, there are intermediate positions when $a_i \in \{1, \ldots, M-1\}$ for any agent.

We empirically evaluate the behavior of several types of algorithms in the MASD in self play. These algorithms are the S-algorithm, fixed algorithms, belief-based algorithms (Fictitious Play), reinforcement learning algorithms (Q-learning, WoLF-PHC, and M-Qubed), and no-regret algorithms (Exp3). Results and key parameters are summarized in Table 2. We selected these particular algorithms as points of comparison with the S-algorithm since they (a) span the space of typical multi-agent learning algorithms, (b) have been, for the most part, commonly studied in the literature, and (c) can operate in minimal information environments (with the exception of belief-based algorithms and M-Qubed).

The simplest possible strategies are fixed, non-adaptive, strategies. We first considered always defect, always cooperate, and uniform random selection. Each of these strategies suffers from weaknesses. They are either insecure, inefficient, or both since the fixed strategies fail to adapt to other agents playing the game. Note that tit-for-tat is both efficient and secure in this game, but lack of knowledge of the game structure and the actions of other agents would make this strategy difficult to implement.

The idea of belief-based algorithms is to model the probabilities of opponents taking actions, and then use those probabilities to determine the expected reward for each action. This has been studied in both economics and game theory for a wide variety of games. We evaluated a general form of belief-based learning [12] that subsumes many other forms of belief-based learning, such as fictitious play, as special cases. This form counts the number of actions played by opponents, translates this into a frequency-based probability model of the

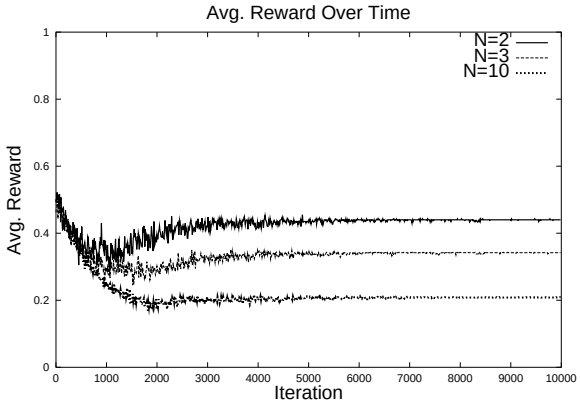


Figure 8: The average reward of Q-learning agents over time in the MASD. The three lines represent three separate experiments with N=2, N=3, and N=10. In all cases, $M = 1$. Each experiment consisted of averaging the rewards of all the agents over 200 trials. The game parameter k was chosen from a uniform random distribution over its legal range given N. The Q-learners were stateless and used a fixed $\alpha = 0.2$, a discount factor $\gamma = 0.9$ and ϵ-greedy exploration.

opponents, computes the expected value for each response to this model, and then plays a mixed-strategy using this expected value as an input to a Boltzmann distribution with parameter λ_i. A key result for this form of belief-based learning is that it degenerates into purely random play in the MASD or any other similar game. The proof of this is found in [35].

The Q-learning algorithm serves as an informative baseline for comparison, as it represents a best-response-style algorithm. We explored various state representations including no state, state consisting of the entire joint action from the previous round of play, and a state consisting of the sum of the opponents' actions on the previous round of play[2]. Results are relatively insensitive to learning parameters and to state representation. Typical results are shown in Figure 8, which displays the average rewards throughout the learning process for three different systems of Q-learning agents. In order to account for the inherent randomness in exploration, the data in Figure 8 is an average of 200 separate simulations. In most cases, the selected joint action was the Nash equilibrium, but occasionally mutual cooperation emerged. The properties of the game (with the exception of the number of actions, M) are important in determining when cooperation will emerge. For example, it is apparent in Figure 8 that N, the number of players, is an important factor in determining the ability of the agents to learn cooperation. Figure 9 displays the relationship between the number of players and the average payoff in more detail.

A variation of Q-learning designed for use in multi-agent games is WoLF-PHC [6]. WoLF-PHC extends Q-learning in two ways: it allows learning of mixed strategies by invoking policy hill climbing (PHC), and it alters learning rates depending on whether an agent thinks it is succeeding or failing. This philosophy of altering learning rates, encoded in Bowling and Veloso's *win or learn fast* (WoLF) principle, is in the spirit of staggered learning wherein agents take turns playing fixed policies and learning best responses. Although PHC allows learning of mixed strategies, it is easily shown that there are interesting games such as Shapley's game [14] where PHC will not converge. WoLF extends the number of games on which PHC can converge, but WoLF-PHC does not converge on every game. As it applies to the MASD, it is not a failure to converge

[2]Note that the use of joint actions or opponent behaviors as states violates the assumptions of minimal information environments.

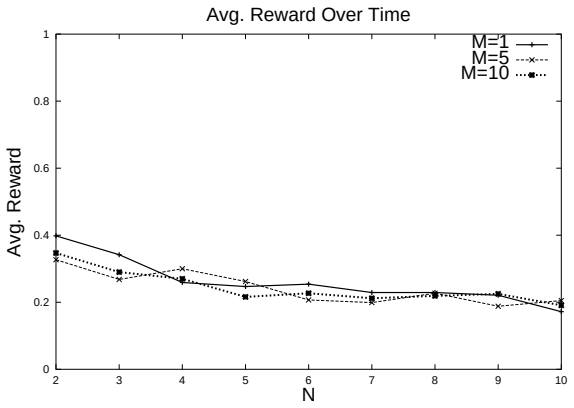


Figure 9: A graph of the average reward as a function of the number of agents N. For each value of N, 200 games were played and the average reward over the entire game was plotted. Each series represents a different value for M, but there appears to be little correlation between M and the ability of the agents to learn cooperation.

that is of interest, but rather the emphasis on the single-shot Nash equilibrium as the target for learning. As suggested by its name, PHC is a best-response learner. Thus, WoLF-PHC seeks to learn a best-response on the stage game. Empirical results indicate that for almost all initial parameter settings over both 2-player and 10-player games, stateless WoLF-PHC converges to defection. The speed of convergence depends on the relative difference between the payoffs for self and group interest, but eventually this best response learner learns the single-shot Nash equilibrium.

While most no-regret learning algorithms require knowledge of the complete payoff matrix as well as the actions of associates, some do not, including Exp3 [1]. Like the previous algorithms we have discussed, Exp3 and most other no-regret learning algorithms learn the single-shot Nash equilibrium in the MASD. As such, they perform poorly in the repeated play of this game.

Unlike the other learning algorithms we have discussed, in self play, the S-algorithm typically learns to allocate all resources to the group goal in the MASD provided that each agents' initial aspirations are high enough. Thus, it is efficient in this game, which results in higher average payoffs. Empirical results for the MASD in self play are given in [35]. We summarize these results for completeness. Figure 10 shows the average rewards of the S-algorithm in the MASD when each agent sets its initial aspiration level higher than unity (which ensures that the joint aspiration vector will enter a subregion that satisfies the conditions of pareto efficiency in this game). The figure shows that while, indeed, the S-algorithm learns cooperation it is affected by both the number of actions ($M + 1$) and the number of agents playing the game (N). The rate at which performance decays with these parameters is dependent on the learning rate λ_i.

In addition to being efficient, the S-algorithm is also secure in the MASD provided that the associate that does not successfully apply the pumping methodology. Given minimal information environments, successful application of pumping to exploit an S-agent in this game is unlikely, though possible. Thus, while likely to achieve security, the S-algorithm is not guaranteed to be secure in this game.

Table 2 also reviews the proficiency of the M-Qubed algorithm [9] in the MASD. M-Qubed is provably secure. It is also efficient in this game. Unlike the S-algorithm, however, M-Qubed explicitly learns the payoff structure of the game and requires state representation. Thus, its knowledge structures are more

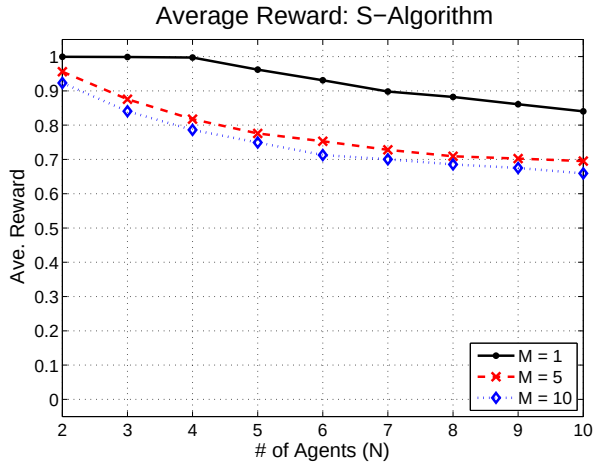


Figure 10: A graph of the average reward to S-agents ($\lambda_i = 0.99$) as a function of the number of agents N. For each value of N, 200 games were played (50,000 iterations each) and the average reward over the entire game was plotted. Each series represents a different value for M.

	c	d
C	$25, 25$	$75, 100$
D	$100, 75$	$50, 50$

(a) Battle of the Sexes

	c	d
C	$86, 86$	$57, 100$
D	$100, 57$	$29, 29$

(b) Chicken

	a	b	c
A	$0, 0$	$0, 100$	$100, 0$
B	$100, 0$	$0, 0$	$0, 100$
C	$0, 100$	$100, 0$	$0, 0$

(c) Shapley's Game

	c	d
C	$100, 100$	$-125, 75$
D	$75, -125$	$50, 50$

(d) Staghunt

	a	b
A	$0, 100$	$100, 67$
B	$33, 0$	$67, 33$

(e) Tricky Game

	a	b	c
A	$0, 91$	$91, 15$	$91, 0$
B	$15, 91$	$100, 100$	$45, 91$
C	$91, 45$	$91, 30$	$30, 91$

(f) Preclusion Game

	a	b
A	$88, 50$	$88, 25$
B	$25, 100$	$100, 75$

(g) Security Game

Table 3: Payoff matrices used to compare the algorithms (rounded to the nearest whole number).

complex than the S-algorithm[3].

Learning Behavior in Other Games. We now discuss the behavior of the S-algorithm in other interesting 2-player matrix games. We selected several games from the set of classical matrix games (Battle of the Sexes, Chicken, Shapley's game, and Staghunt; Table 3a–d). These games test the algorithms' ability to deal with risk, to coordinate actions, and to share profits. We also selected several other matrix games designed to test algorithms on convergence properties (Tricky game [5]; Table 3e), security CEs (Table 3g) and preclusion CEs (Table 3f). We also tested the S-algorithm on randomly generated 2-player, 5-action matrix games.

We studied two different methods for initializing aspirations. In the first method, each S-agent initialized its aspirations to its highest payoff ($r_i^{\max}$), as high initial aspiration levels have been shown to be successful in the prisoner's dilemma [37]. In the second method, each S-agent randomly initialized its aspirations to a value between its maximin value (r_i^{mm}) and its highest payoff ($r_i^{\max}$). Requiring that initial aspirations be at least as great as the maximin value is reasonable since a player knows that it can sustain this payoff.

[3]It is interesting to note that M-Qubed is really just a more complicated version of the S-algorithm. It adds action history, and, thus, additional states, and an informed mechanism for trembling aspirations.

Name	Parameter Values
S-agent (max)	$\lambda_i = 0.99$; $\alpha_i^0 = r_i^{\text{mm}}$
S-agent (random)	$\lambda_i = 0.99$; $\alpha_i^0 \in [r_i^{\text{mm}}, r_i^{\max}]$
Exp3	As defined in [1]
WoLF-PHC	$\alpha = 1/(10 + 0.01\kappa_s^a)$, $\gamma = 0.95$, ϵ-greedy expl. w/ $\epsilon = \max(0, 0.2 - 0.00006t)$, $\delta_w = \frac{1.0}{10+\kappa_s^a}$, $\delta_l = \frac{4.0}{10+\kappa_s^a}$
Q-Learning	$\alpha = 1/(10 + 0.01\kappa_s^a)$, $\gamma = 0.95$, ϵ-greedy expl. w/ $\epsilon = \max(0, 0.2 - 0.00006t)$

Table 4: The agents' parameter values. κ_s^a is the # of times that action a has been played in state s. λ_i is used in place of α to avoid confusion. QL uses a stateless Q-update.

While neither $r_i^{\max}$ nor r_i^{mm} are immediately known in minimal information environments, an agent can easily estimate these values by observing its payoffs for a number of episodes at the beginning of the game (while playing randomly).

As we did in the case of the MASD, we compare the performance of the S-algorithm with other learning algorithms that (a) are suitable for minimal information environments and (b) have similar representational capabilities as the S-algorithm. We chose Q-learning, WoLF-PHC, and Exp3. Table 4 specifies the parameter values used in study.

General Trends. The average payoffs obtained by the algorithms after learning in each of these games are shown in Figure 11. The figure also plots the payoffs corresponding to the Nash bargaining solution (NBS)[4] of each game as compute by Littman [24]. We note three important general trends before discussing the individual results. First, in most of the games, the average payoffs of the S-agents (both *S-agent (max)* and *S-agent (random)*) are as high or higher than the other three learning algorithms. This is largely due to the S-algorithms' ability to learn efficient solutions in many games. Second, more often than not, the average payoffs of the S-agents match or nearly match the payoffs corresponding to the NBS. Third, *S-agent (max)* is typically more successful than *S-agent (random)*.

We discuss the performance of the algorithms in each individual game separately in order to better understand the properties of the algorithms.

Battle of the Sexes. In this game, each of the learning algorithms always learns to play either (D, c) or (C, d). Both solutions rewards one of the agents a payoff of 100 and the other agent a payoff of 75. These pareto efficient solutions produce the same average payoff as does the NBS. Note, however, that the behavior of each of the algorithms is different than the NBS, as the NBS requires the agents to alternate between the (D, c) and (C, d) solutions.

Chicken. In Chicken, both *S-agent (max)* and *S-agent (random)* learn to play (C, c), resulting in both players receiving a payoff of 86. This solution is pareto efficient and corresponds to the NBS. The other three algorithms usually learn a solution in which one of the agents receives a payoff of 100 while the other receives a payoff of 50, resulting in an average payoff of 75. Thus, the S-algorithm outperforms the other three learning algorithms in this game.

Shapley's Game. The S-algorithm does not converge in Shapley's game. Rather the empirical distribution of its actions is random, which produces an average payoff of approximately 33 to each agent. In the pure-strategy sense, this average joint payoff is pareto efficient. However, in the NBS, the players take turns "winning" (payoff of 100) and losing (payoff of 0), which results in an average payoff of 50 to each agent. Both Q-learning and Exp3 achieve this average payoff, though they do not converge to the NBS. Rather, these other algorithms take turns exploiting each other for long periods of time as their policies slowly change. Like the S-algorithm, neither WoLF-PHC, Q-learning, nor Exp3 converge in this game.

Shapley's game reveals an significant weakness of the S-algorithm, which is that it cannot learn solutions

[4]The NBS is a PE solution that maximizes the product of the agents' advantages [27].

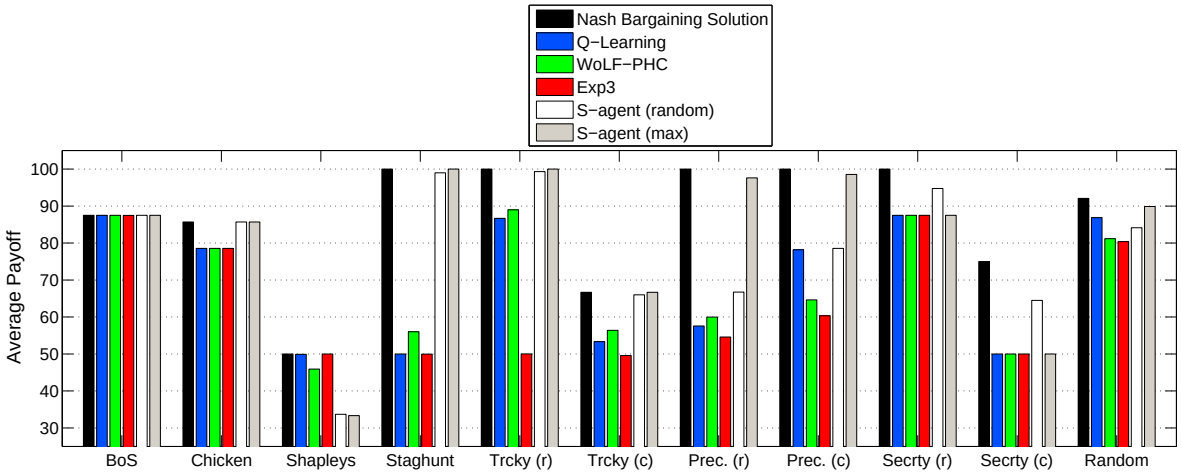


Figure 11: Average learned payoffs to the agents in various 2-player matrix games shown in Table 3 (in self play). The bars show the algorithms' average payoffs in the last 10,000 episodes (out of 200,000), averaged across 50 trials. For symmetric games, average payoffs for both row and column players are lumped together. For asymmetric games, the average payoffs of the row (r) and column (c) players are shown separately (with the exception of the random games).

that are not pure strategies. This means that the S-algorithm cannot learn profitable compromises in which the agents alternate between solutions. Technically, however, the S-agent's play is still efficient in Shapley's game with respect to the definition of pareto efficiency given in Section 2.

Staghunt. The Staghunt is a game in which the S-algorithm substantially outperforms Exp3, Q-learning, and WoLF-PHC. In this game, both players profit the most when the joint action (C, c) is played. However, this solution is risky as a deviation by an associate yields a payoff of -125. Thus, most learning algorithms, including Exp3, Q-learning, and WoLF-PHC, learn to play the less risky NE in which the agents play (D, d), which results in a payoff of 50 to each agent (the maximin value). However, *S-agent (max)* and *S-agent (random)* both learn to play the (more profitable) NBS of (C, c), resulting in a payoff of 100 to each player. Thus, in self play, the S-algorithm's ability to be efficient in this game results in higher payoffs than the other learning algorithms.

Tricky Game. In Tricky game, the NBS is for the agents to play the joint action (A, b), which yields a joint payoff of $(100, 67)$. Both *S-agent (max)* and *S-agent (random)* usually learn to play this solution (though, in one of the 50 trials of our study, *S-agent (random)* converged to (B, b)). Thus, the S-agent exhibits efficient behavior in this game, while the other learning algorithms typically do not. WoLF-PHC and Exp3 generally learn the mixed-strategy NE solution of the game, in which each agent plays randomly. This results in an average payoff of 50 to each agent, which is substantially lower than the payoffs obtained by the S-algorithm.

Preclusion Game. In the preclusion game, which is functionally similar to the game shown in Figure 6, *S-agent (max)* typically learns to play the NBS (B, b), which gives each agent a payoff of 100. *S-agent (max)'s* average payoffs are just below this value since, in three of the 50 trials, the S-agents failed to play (B, b) before the agents' aspirations fell below 91. This caused the agents to become subject to a cyclic preclusion CE if any action other than (B, b) were played. A slower learning rate (i.e., higher λ_i) makes this event less likely.

When the S-agents initialize their aspirations to random values between r_i^{mm} and $r_i^{\max}$ in the preclusion game, the play of the S-agent is typically not efficient, as shown by the performance of *S-agent (random)* in

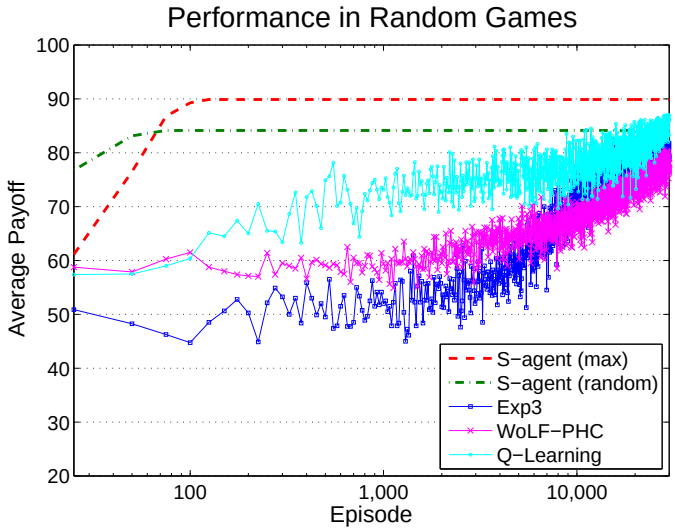


Figure 12: Average payoffs over time to the various agents in random matrix games in the first 15,000 episodes. Payoffs are the average across both agents in 50 random 2-player, 5-action matrix games.

Figure 11. This is because initializing aspirations in this way does not produce joint aspirations that satisfy the conditions of pareto efficiency. Thus, in this game, the choice of initial aspirations is critical for efficient play. We note that WoLF-PHC, Q-learning, and Exp3 are, in effect, also subject to the preclusion CE. As a result, their payoffs are much lower in this game than those of *S-agent (max)* and, even, *S-agent (random)*).

Security Game. The security game also requires careful selection of initial aspirations. In this game, the security critical event in which the row player plays A results in the exclusion of the NBS (which is (B, b)). This event occurs when the row player's aspirations are 88 or less. Thus, when both players initialize their aspiration level to the maximum reward of 100, the S-agent becomes subject to this CE before the joint payoff $(100, 75)$ becomes mutually satisficing. As a result, *S-agent (max)* is not efficient in this game since joint aspirations do not enter a subregion that satisfies the conditions of pareto efficiency. However, other joint initial aspirations do satisfy the conditions of pareto efficiency, as demonstrated by the fact that *S-agent (random)* is sometimes able to avoid this security CE. Thus, the choice of initial aspirations determines whether or not the S-algorithm is efficient in this game. However, unlike the other games we have encountered, the "right" initialization is not to the highest payoff.

We note that Exp3, Q-learning, and WoLF-PHC are also not efficient in this game, as they each become subject to the security CE.

Random Games. As a final test, we analyze the performance of the algorithms in randomly generated 2-player, 5-action matrix games. In these games, *S-agent (max)* outperforms all the other agents. In fact, its average payoffs are only slightly smaller than the average payoffs of the corresponding NBSs. Thus, *S-agent (max)* typically learns efficient solutions in these randomly generated games. However, *S-agent (random)* does not perform nearly so well, thus demonstrating the importance of initial aspirations.

In addition to comparing the algorithms' abilities to learn efficient, rewarding solutions, we also compare the algorithm's with respect to have fast they learn. These results are shown in Figure 12 for the random generated games. The figure shows that both *S-agent (max)* and *S-agent (random)* have converged in all 50 randomly generated matrix games within approximately 100 episodes, whereas the other learning algorithms still have not converged after 15,000 episodes. This represents a substantial performance advantage of the S-algorithm over the other learning algorithms.

	MASD	Battle of the Sexes	Chicken	Shapley's Game	Staghunt	Tricky Game	Preclusion Game	Security Game
S-agent (max)	PE (NBS)	PE	PE (NBS)	PE*	PE (NBS)	PE (NBS)	PE (NBS)	
S-agent (random)	?PE	PE	PE (NBS)	PE*	PE (NBS)	PE (NBS)		?PE
WoLF-PHC		PE	PE	PE*				
Q-Learning		PE	PE	PE				
Exp3	PE		PE	PE				

Table 5: Summary of efficiency properties for the learning algorithms in the several matrix games. PE indicates that the algorithms is efficient, PE (NBS) indicates that the algorithm learns the (pareto efficient) Nash bargaining solution, ?PE indicates that the algorithm is efficient for some initial aspirations, and PE* indicates that the algorithm is efficient with respect to the game's pure strategies only.

Table 5 summarizes the algorithms' efficiency properties in the games tested. The table shows that both *S-agent (max)* and *S-agent (random)* are both efficient in more games than WoLF-PHC, Q-learning, and Exp3. Furthermore, the S-algorithm is typically efficient when each agent sets its initial aspirations to its highest payoff. However, other initial aspirations are necessary for the S-algorithm to be efficient in some games (e.g., the Security game). In the next section, we introduce an extension of the S-algorithm that eliminates this dependence on initial aspiration levels.

5 The S-Algorithm with Learned Trembles (SALT)

In this section, we introduce SALT (*S-a*lgorithm with *L*earned *T*rembles). In the next section, we show that (a) SALT is provably secure, (b) it empirically demonstrates efficiency in self play in many general-sum matrix games, and (c) it performs reasonably well in general-sum matrix games when it associates with other learners with similar representational and reasoning capacities. Together, these properties make SALT a robust learning algorithm for general-sum matrix games played in minimal information environments.

Like the S-algorithm, SALT uses an aspiration relaxation search to learn a sustainable aspiration level and a behavior to sustain it; the SALT algorithm is shown in Algorithm 2. In addition to explicitly stating how initial aspiration levels are set in steps 1 and 2, SALT differs from the S-algorithm in two significant ways. First, SALT trembles its aspiration level with some probability in each episode (step 3.D). Once SALT chooses to tremble, it sets its aspiration level to a new value (called the *tremble value*) (step 3.D.iii-iv). Formally, let η_i^t be the probability that agent i trembles its aspiration level in episode t. Then, with probability η_i^t, SALT sets its aspiration level α_i^t to the tremble value γ_i^τ, where τ is the number of times that SALT has trembled its aspirations up to time t. We describe how η_i^t and γ_i^τ are determined later in this section.

Second, when its most recent tremble value γ_i^τ does not exceed its maximin value, SALT modifies the S-algorithm's action selection and aspiration update rules. Specifically, when $\gamma_i^\tau \leq r_i^{\text{mm}}$, SALT plays its maximin action a_i^{mm} (step 3.A) and maintains the constant aspiration level $\alpha_i^t = r_i^{\text{mm}}$ (step 3.B) until it again trembles its aspirations. These modifications form a secure trigger strategy[5]. When aspirations are trembled to or below the maximin value, SALT reasons that it can gain nothing by exploring other options or updating its aspiration level since the maximin action will sustain its current aspirations. Similar trigger strategies are given by [9] and [29].

Provided that SALT effectively determines when to tremble (i.e., the tremble probability η_i^t) and which tremble value γ_i^τ to select, these two modifications give SALT the two properties identified in the outset of this paper. Regardless of initial aspiration levels, SALT is efficient in many games (first modification) and provable secure (second modification). To effectively determine when and how to tremble, SALT learns the utility of trembling its aspirations to the various values of its reward space, which it estimates with the

[5]Trigger strategies form the basis for the proof of the folk theorem [15].

1. Play randomly for D_i episodes to estimate r_i^{mm}, a_i^{mm}, and $r_i^{\max}$

2. Initialize
   A. Aspiration level: $\alpha_i^0 = r_i^{\max} + \varepsilon_i$
   B. Initialize time counts: $t = 0$, $\tau_i = 0$
   C. Record epoch 0's initial aspiration level: $\gamma_i^0 = \alpha_i^t$

3. Repeat
   A. Select action a_i^t

$$a_i^t \leftarrow \begin{cases} a_i^{\text{mm}} & \text{if } (r_i^{\text{mm}} \geq \gamma_i^\tau) \\ a_i^{t-1} & \text{if } (r_i^{t-1} \geq \alpha_i^{t-1}) \\ \text{rand}(A_i) & \text{otherwise} \end{cases}$$

   where $\text{rand}(A_i)$ is a random selection from A_i

   B. Receive reward r_i^t and update r_i^{mm}, a_i^{mm}, $r_i^{\max}$, and

$$\alpha_i^{t+1} \leftarrow \begin{cases} r_i^{\text{mm}} & \text{if } (r_i^{\text{mm}} \geq \gamma_i^\tau) \\ \lambda_i \alpha_i^t + (1-\lambda_i) r_i^t & \text{otherwise} \end{cases}$$

   C. $t \leftarrow t + 1$

   D. With probability η_i^t (determined using Eq. (10))
      i. Update $U_i^t(\gamma_i^\tau)$, the utility for trembling to γ_i^τ, using Eq. (7)
      ii. Start a new epoch: $\tau_i \leftarrow \tau_i + 1$
      iii. Select tremble value γ_i^τ using Eq. (8)
      iv. Tremble aspiration level: $\alpha_i^t = \gamma_i^\tau$

Algorithm 2: The S-algorithm with Learned Trembles (SALT) for agent i.

utility function $U_i^t(\cdot)$ (updated in step 3.D.i of Algorithm 2). In the remainder of this section, we describe how $U_i^t(\cdot)$ is estimated, and how η_i^t and γ_i^τ are determined from $U_i^t(\cdot)$. Note that we present functional forms for determining these values, forms that satisfy the conditions of security and efficiency described in Section 3. Future work should explore alternatives to these forms that could accelerate learning while still satisfying the conditions of security and efficiency.

5.1 Learning the Utility of Aspiration Trembles

Aspiration trembles partition a repeated game into a sequence of epochs. Each epoch consists of a series of episodes[6] beginning and ending with an aspiration tremble. Let κ_i^τ be the number of episodes that elapsed between agent i's $(\tau_i)^{\text{th}}$ and $(\tau_i + 1)^{\text{th}}$ aspiration trembles, and let epoch τ_i denote the set of episodes in the time interval $I_i^\tau = \left[\sum_{j=0}^{\tau_i - 1} \kappa_i^j, \sum_{j=0}^{\tau_i - 1} \kappa_i^j + \kappa_i^\tau\right)$. Let the payoff attributed to the agent's $(\tau_i)^{\text{th}}$ aspiration tremble be the average payoff received by the agent in each episode of epoch τ_i, denoted $u_i^\tau = \frac{1}{\kappa_i^\tau} \sum_{j \in I_i^\tau} r_i^j$. Then, epoch τ_i can be summarized with the 3-tuple $(\gamma_i^\tau, \kappa_i^\tau, u_i^\tau)$.

SALT uses the set of 3-tuples up to time t as a set of samples to estimate the utility of the tremble value γ, represented by the utility function $U_i^t(\gamma)$. Since $U_i^t(\cdot)$ is continuous, SALT use a function approximator to estimate $U_i^t(\gamma)$ from the finite set of samples. Specifically, it represents the sample from epoch τ_i as the scaled Gaussian function $p(\gamma, \gamma_i^\tau) = u_i^\tau \kappa_i^\tau \exp\left(-\frac{(\gamma - \gamma_i^\tau)^2}{2\sigma_i^2}\right)$. SALT then approximates $U_i^t(\cdot)$ with a mixture

[6]Recall that an *episode* consists of one round, or stage, of the repeated matrix game, in which each agent selects and plays an action and observes the resulting outcome. An epoch is a series of joint actions.

of Gaussians. Formally, the utility $U_i^t(\gamma)$ is given by

$$U_i^t(\gamma) = \frac{\sum_{j=1}^{\tau_i} u_i^j \kappa_i^j p(\gamma, \gamma_i^j)}{\sum_{j=1}^{\tau_i} \kappa_i^j p(\gamma, \gamma_i^j)}, \tag{7}$$

where the denominator normalizes the utilities within the reward space of the game.

5.2 Determining the Tremble Value γ_i^τ

SALT uses the utility function $U_i^t(\cdot)$ to determine where to tremble its aspirations (i.e., how to determine the tremble value γ_i^τ in step 3.D.iii from Algorithm 2). Aspiration trembles serve two purposes, both of which imply an interval within the reward space to which SALT should consider trembling its aspirations. First, aspiration trembles allow an agent to search for more profitable payoffs than it is currently receiving. In general, these searches will be more profitable if aspirations are perturbed upward. Thus, SALT should consider trembling it aspirations to values within the interval $\Gamma_i^{\max}(t) = (\alpha_i^t - \varepsilon_i, r_i^{\max} + \varepsilon_i)$. Allowing the agent to tremble its aspirations downward by the quantity ε_i allows the agent to remain satisfied with its current payoffs and behavior.

Second, aspiration trembles allow SALT to avoid being exploited. Recall that SALT avoids being exploited by trembling its aspirations (possibly downward) to its maximin value r_i^{mm} and then playing its maximin action a_i^{mm} for the remainder of the epoch. Hence, SALT also considers trembling its aspirations to values within a second interval $\Gamma_i^{\text{mm}} = (r_i^{\text{mm}} - \varepsilon_i, r_i^{\text{mm}}]$. The union of the compact intervals $\Gamma_i^{\max}(t)$ and Γ_i^{mm}, given by

$$\Gamma_i(t) = \Gamma_i^{\text{mm}} \cup \Gamma_i^{\max}(t) = (r_i^{\text{mm}} - \varepsilon_i, r_i^{\text{mm}}] \cup (\alpha_i^t - \varepsilon_i, r_i^{\max} + \varepsilon_i),$$

defines the set of values to which SALT considers trembling its aspirations.

SALT seeks to tremble its aspirations to values within $\Gamma_i(t)$ that produce high payoffs. To do this, SALT selects a tremble value within the set $\Gamma_i(t)$ with probability based on the utility function $U_i^t(\cdot)$. Formally, SALT selects tremble value $\gamma_i^\tau \in \Gamma_i(t)$ with probability determined by the Boltzmann distribution:

$$Pr(\gamma_i^\tau = \gamma_i) = \frac{\exp\left(\frac{1}{T_i} U_i^t(\gamma_i)\right)}{\sum_{\gamma_i \in \Gamma_i(t)} \exp\left(\frac{1}{T_i} U_i^t(\gamma)\right)} \tag{8}$$

where T_i is the temperature parameter. T_i should start high and cool with time. In so doing, SALT will explore the effects of various aspiration trembles in early epochs and exploit what it has learned in later epochs.

5.3 Determining the Tremble Rate η_i^t

The tremble rate, η_i^t, determines how frequently SALT trembles its aspirations, and thus how much exploration is performed during learning. Two principles govern SALT's tremble rate (step 3.D of Algorithm 2). First, the tremble rate is dependent on how SALT's current payoffs (estimated by its current aspiration level α_i^t) compare with the utility of trembling its aspiration level (estimated by $\max_{\gamma \in \Gamma_i(t)} U_i^t(\gamma)$). When aspiration trembles are expected to produce substantially higher payoffs than its current aspiration level (i.e., $\max_{\gamma \in \Gamma_i(t)} U_i^t(\gamma) - \varepsilon_i > \alpha_i^t$), the agent can benefit from trembling its aspirations. Otherwise, the agent should not be inclined to tremble. The sigmoid function $P(x_i^t) = \frac{1}{1+e^{-x_i^t}}$ captures this decision rule when x_i^t is set to

$$x_i^t = \left(\frac{\tau_i}{r_i^{\max} - r_i^{\text{mm}}}\right)\left(\max_{\gamma \in \Gamma_i(t)} U_i^t(\gamma) - \alpha_i^t - \varepsilon_i\right). \tag{9}$$

The first term in Eq. (9) is a scaling factor that (a) normalizes the sigmoid function to the game's payoffs and (b) conveys SALT's confidence based on the quantity of its experience (expressed by τ_i). The second term compares the utility of trembling (estimated by $\max_{\gamma \in \Gamma_i} U_i^t(\gamma)$) with the agent's current payoffs (estimated by α_i^t). When current payoffs are substantially less than the utility for trembling, $P(x_i^t)$ is close to unity. Otherwise, it approaches zero.

The second variable that affects the tremble rate, η_i^t, is experience. As is typical in reinforcement learning, the exploration (i.e., tremble) rate should decrease over time to allow the agent to exploit what it has learned. We express this rule with the function $f(\tau_i)$, which is monotonically decreasing and bound to the interval (0,1). In practice, we use $f(\tau_i)$ of the form $f(\tau_i) = \frac{1}{a+b\tau_i}$, where $a \geq 1$ and b is a positive constant.

Together, $f(\tau_i)$ and the sigmoid function $P(x_i^t)$ define SALT's tremble rate η_i^t. Formally,

$$\eta_i^t = P(x_i^t) f(\tau_i) = \frac{f(\tau_i)}{1 + e^{-x_i^t}}. \tag{10}$$

6 Properties of the S-Algorithm with Learned Trembles

In this section, we analyze SALT's performance properties, beginning with security and efficiency. We show that SALT is provably secure, and that it is efficient in a wide range of general-sum matrix games. We then show that SALT's efficiency and security properties make it a robust learning algorithm, capable of preforming well in associations with other learning algorithms with similar representational and reasoning capacities in minimal information environments.

6.1 Security

The main contribution of this subsection is a theorem which states that SALT is secure. This theorem applies under reasonable assumptions regarding SALT's parameter settings. These assumptions and the proof of the theorem are given in Appendix A.

Theorem 2. *SALT is secure in all repeated general-sum matrix games regardless of the behavior of its associates.*

Due to potential discrepancies between the security guarantees provided by the pure-strategy maximin value that acts as a lower bound on SALT payoffs and the mixed-strategy maximin values often learned by competing algorithms, it is useful to empirically evaluate SALT's performance in competitive constant-sum games against other learners with similar reasoning capabilities. To do so, we observe the performance of the S-algorithm and SALT in competition with Exp3, WoLF-PHC, and Q-learning in randomly-chosen 2-player, 5-action zero-sum games. We chose these games since (a) randomly generating payoff matrices provides an unbiased selection of games, (b) many of these games will have mixed-strategy equilibrium solutions, and (c) the number of actions is sufficiently high to be interesting but not to cause prohibitively slow learning.

Parameter settings for the algorithms are given in Table 6 (for SALT and the S-algorithm) and in Table 4 (for the other algorithms). Note that, for all results presented in this section, SALT and the S-algorithm initialize their aspirations as specified in steps 1 and 2 of Algorithm 2 in order to satisfy the constraints of minimal information environments (rather than allow the S-algorithm to know *a priori* the highest payoff $r_i^{\max}$, as was done in Section 3 for *S-agent (max)*). Also, the parameter settings used by the S-algorithm and SALT are randomly selected from the range of reasonable values shown in Table 6 to provide a form of sensitivity analysis.

The average payoffs to SALT and the S-algorithm against each of the other algorithms are shown in Figure 13 along with the average pure-strategy and mixed-strategy maximin values of these games. We make several observations. First, the average payoffs of both the S-algorithm and SALT are closer to the mixed-strategy maximin than the pure-strategy maximin. However, both SALT and the S-algorithm have small negative average payoffs against Exp3 and WoLF-PHC. On the other hand, both SALT and the S-agent

Parameter	Value	Explanation
λ_i	$\lambda_i \in [0.95, 0.99]$	Aspiration learning rate
D_i	$D_i = \lvert A_i \rvert^2 c$, where $c \in [1, 20]$	# of episodes in observation period
$f(\tau_i)$	$f(\tau_i) = \frac{1}{a+b\tau_i}$, where $a \in [10, 30]$, $b \in [0.5, 1.5]$	Function used in Eq. (10)
T_i	$T_i = \frac{T^c}{\tau_i}$, where $T^c \in [10, 40]$	Temperature used in Eq. (8)
ε_i	$\varepsilon_i = \max(0.001, 0.02(r_i^{\max} - r_i^{\min}))$	Error tolerance
σ_i	$\sigma_i = 2\varepsilon_i$	Standard deviation used in mixture of Gaussians

Table 6: Parameter values used by SALT and the S-algorithm (where applicable).

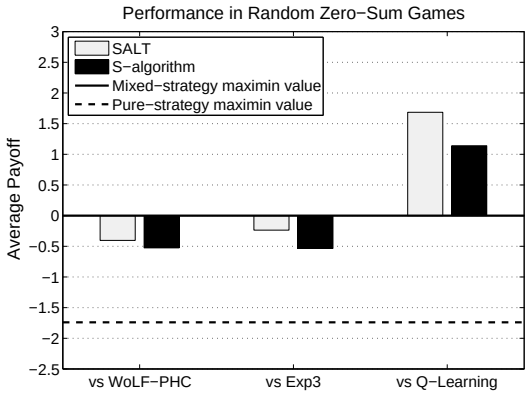


Figure 13: Average payoffs to SALT and the S-algorithm when they associate with WoLF-PHC, Exp3, and Q-learning in 50 randomly-chosen 2-player, 5-action zero-sum matrix games with payoffs between -8 and 8. Payoffs are the average of the last 10,000 (out of 200,000) episodes.

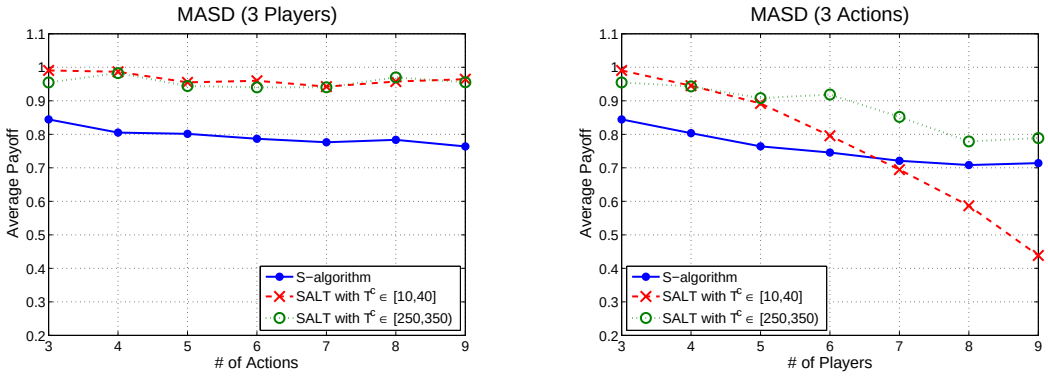
exploit Q-learning. These results are not surprising, as both Exp3 and WoLF-PHC were designed to learn mixed-strategy solutions, while Q-learning was not.

Second, SALT outperforms the S-algorithm by a small margin against each of the three algorithms. Since SALT trembles its aspirations, its behavior is more difficult to model than the S-algorithm's. Additionally, since SALT is secure, it is able to bound its losses in each game by the pure-strategy maximin, while the S-algorithm provides no such guarantee.

In short, while SALT satisfies the security definition given in Section 2, it is not as successful in purely competitive games as other learning algorithms. Thus, when it is known that the game is purely competitive, many other learning algorithms would be more desirable choices than SALT. However, in minimal information environments in which the kind of game is unknown, SALT is a desirable choice since it simultaneously (a) guarantees a bound on its losses (since it is secure in the pure-strategy sense), (b) displays efficient behavior in self play, and (c) performs well against other algorithms in many general-sum matrix games. We demonstrate these latter two properties in the rest of this section.

6.2 Efficiency

We now analyze SALT in terms of efficiency in self play. Since Section 4.2.4 demonstrated that the S-algorithm is more efficient than the other learning algorithms that we considered, we use the S-algorithm paired with the NBS as a standard of comparison to illustrate efficiency. Unless stated otherwise, both the



(a) (b)

Figure 14: Average payoffs to SALT and the S-algorithm in (a) a 3-player MASD for various numbers of actions and (b) a 3-action MASD for various numbers of players. Payoffs are an average of the last 10,000 episodes (out of 800,000) in 50 trials.

S-algorithm and SALT use the parameter settings given in Table 6 (where applicable). The algorithms' parameter values are randomized to provide an implicit sensitivity analysis. We begin by analyzing SALT in the MASD.

6.2.1 Multi-agent Social Dilemma (MASD)

Recall that the MASD is a mixed-motive game with large differences between the one-shot NE (which yields a payoff of zero to each player) and the NBS (which yields a payoff of one to each player). Additionally, since it is scalable to many actions and agents, the MASD is a good first test since it can be used to assess algorithms' scalability characteristics.

Figure 14 compares the performance of the S-algorithm and SALT in self play for varying numbers of actions (Figure 14a) and players (Figure 14b). We note several important trends. First, the S-algorithm's performance shown in the figures is substantially lower than the performance reported by [35]. Rather than converge to the NBS, which yields a payoff of one to each player, the figure shows the S-algorithm's performance is around 0.85 for the 3-player, 3-action MASD. In minimal information environments, the value $r_i^{\max}$ must be estimated during an observation period before the S-agent can set its initial aspiration level. Since each S-agent randomly selects the length D_i of this observational period, the length of each S-agent's observational period is distinct. This results in a violation of the "similarity" requirement noted by [35] that is necessary for the S-algorithm to converge to the NBS. As such, the S-agents' payoffs are reduced.

However, aspiration trembles eliminate dependence on initial aspiration levels. For example, in a 3-player MASD, SALT agents learn to tremble their aspirations so that the resulting joint aspiration vector satisfies the conditions of pareto efficiency. As a result, SALT is efficient in this game (Figure 14a).

Figure 14 shows that SALT scales better to increases in the number of actions in the MASD than to increases in the number of players. While SALT's performance is relatively constant as the number of actions increases (Figure 14a), its performance drops sharply as the number of players increases for some parameter settings (i.e., $T^c \in [10, 40]$; see Figure 14b), though not as much for others (i.e., $T^c \in [250, 350)$).

Sensitivity to parameter settings in the MASD is not unique to SALT. The probability with which the S-algorithm converges to the NBS is contingent on the learning rate λ_i [35]. As the number of players and actions increases, λ_i must be increased in order for the S-algorithm to converge to the NBS with high probability. In like manner, as the number of players in the MASD increases, SALT must learn more slowly which tremble values yield the highest utility. In so doing, the agents are able to effectively adapt to each others' changing behaviors. A higher temperature constant achieves this effect (Figure 14b).

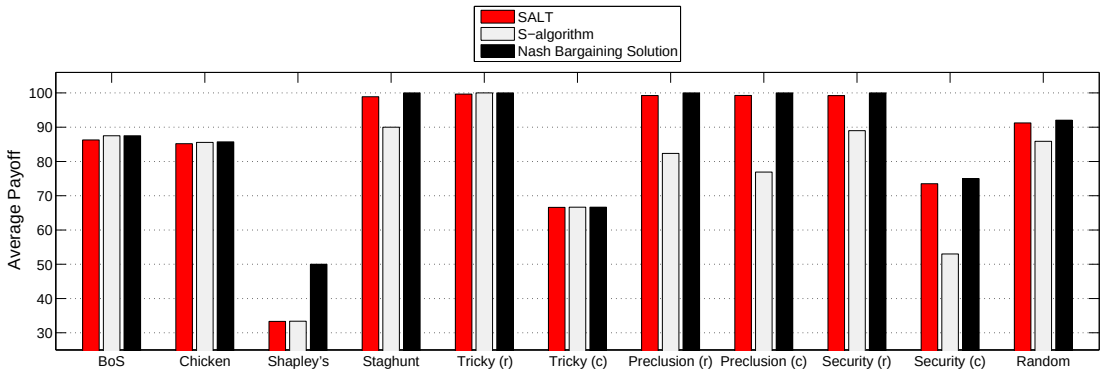


Figure 15: Average payoffs in self play to the agents in various 2-player matrix games. Payoffs are an of the last 10,000 (out of 200,000) episodes in 50 trials. For symmetric and random games, the payoffs for both row and column players are averaged together, while they are shown separately in asymmetric games (with the exception of the random games).

6.2.2 2-Player General-Sum Games

While SALT is efficient in the MASD, the main objective of this paper is to analyze SALT in general-sum games. We now evaluate SALT in 2-player general-sum games. We then analyze its performance in N-player games.

Figure 15 compares SALT's payoffs with those of the S-algorithm and the NBS in the games previously considered in Section 4.2. In each game, the SALT agents learn to tremble their aspirations so that the resulting joint aspiration vector satisfies the conditions of pareto efficiency. As a result, SALT is efficient in each game. Furthermore, it matches (minus a small penalty for exploration in some games) or exceeds the performance of the S-algorithm in each of the games. With the exception of Shapley's Game, SALT's average payoffs correspond to or approach those of the NBSs.

SALT outperforms the S-algorithm substantially in the Staghunt, the Preclusion Game, the Security Game, and randomly-chosen matrix games. In each case, the S-algorithm's initial aspirations do not satisfy the conditions of pareto efficieny. However, the SALT agents learn to tremble their aspirations so that the joint aspiration vector eventually does satisfy the conditions of pareto efficiency. In the Preclusion Game, both the row and column player learn to tremble their aspirations near their highest payoff of 100. On the other hand, in the Security Game, the column player learns to tremble its aspirations to a value near 75, while the row player learns to tremble its aspiration level to its highest payoff of 100, resulting in convergence to the NBS. Similarly, in random games, SALT outperforms the S-algorithm, as it learns to tremble its aspirations so that it learns to play the NBS.

The benefits achieved by aspiration trembles, however, come at a cost. SALT learns slower than the S-algorithm. For example, in the random games, the S-algorithm quickly converges to a profitable solution, while SALT takes longer to learn (Figure 16). Despite these slightly slower learning times, SALT eventually matches or outperforms the S-algorithm in 2-player games. Thus, as demonstrated by the results presented in Section 3, in self play, SALT performs substantially better on average than other algorithms with similar reasoning capacity in 2-player general-sum matrix games.

6.2.3 N-Player General-Sum Games

As a final test, we evaluate SALT's performance in self play in general-sum games with $N \geq 2$ players. In the interest of space, we restrict attention to 4-action randomly-chosen matrix games with between two

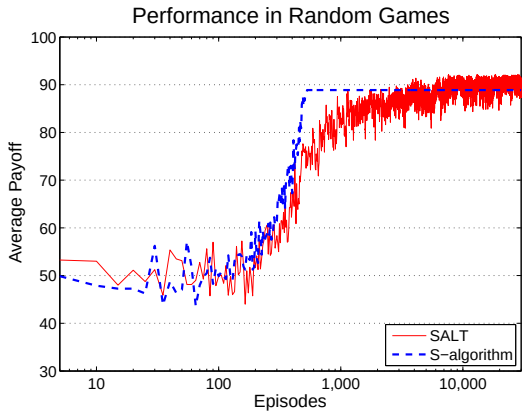


Figure 16: Average payoffs over time to SALT and S-agent (max) in random matrix games in the first 30,000 episodes. Payoffs are an average over 50 random 2-player, 5-action matrix games.

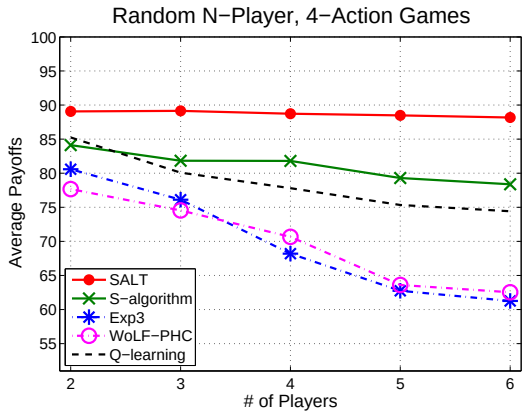


Figure 17: Average payoffs (in the last 10,000 episodes out of 500,000) to the various algorithms in 50 4-action randomly-chosen matrix games for different numbers of players. All payoffs in the games were chosen from the range $[0, 100]$.

and six players. The average payoffs of SALT, the S-algorithm, Exp3, WoLF-PHC, and Q-learning in these games are shown in Figure 17. The figure shows that SALT outperforms the other algorithms regardless of the number of players in the game. Furthermore, whereas increasing the number of players in the MASD negatively affected SALT's performance (for constant parameter settings), it does not have the same effect in randomly-chosen games. We attribute this discrepancy to the fact that the MASD represents a difficult social dilemma wherein the myopic best-response is in complete opposition to cooperative (and profitable) behavior. Thus, in the MASD, SALT agents must decrease how quickly they learn as the number of players in the game increases. However, most randomly-chosen games are not as challenging. As a result, the SALT agents can more easily learn mutually beneficial behavior without altering learning rates.

6.3 Robustness and Evolutionary Stability

A central thesis of this paper is that, since many situations require an agent to interact with many different kinds of agents, a robust learning algorithm should be able to learn good solutions with many other kinds of

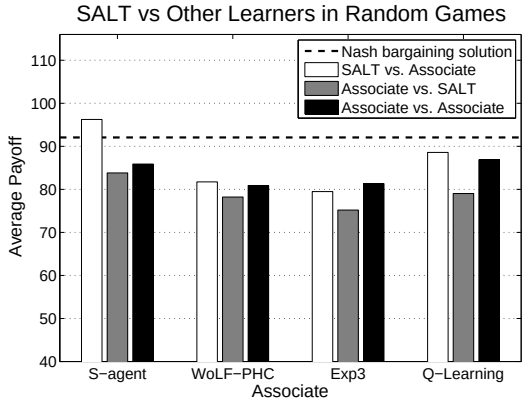


Figure 18: Average payoffs when SALT interacts with other learning algorithms in 50 randomly-chosen, 2-player, 5-action matrix games. Payoffs are averaged over the last 10,000 (out of 200,000) episodes.

learners, even in minimal information environments. Axelrod's work [2] suggests that efficiency and security lead to robust and successful behavior in these situations. Since SALT is both efficient and secure, we expect that it will perform well when associating with other learning algorithms with similar reasoning capacities. We test this hypothesis in two ways. First, we analyze SALT's performance in head-to-head associations with WoLF-PHC, Q-learning, Exp3, and the S-algorithm in minimal information environments. Second, we again build from Axelrod's work and use evolutionary concepts to demonstrate how security and efficiency lead to robust behavior.

In the interest of space, we limit much of the results in this section to randomly chosen 2-player, 5-action games. These games have enough actions to be interesting and are likely to be games of mixed motive (neither purely cooperative nor purely competitive).

SALT's average payoffs in 50 randomly chosen 2-player, 5-action games are compared to the payoffs of its associates in Figure 18. While SALT's average payoffs fall short of the average payoff of the NBSs (except against the S-algorithm), it (a) scores higher than each of the other learning algorithms in head-to-head associations, (b) it scores higher when associating with Q-learning, WoLF-PHC, and the S-algorithm than these algorithms score against themselves, and (c) it scores only marginally lower against Exp3 than Exp3 scores against itself. Thus, while both WoLF-PHC and Exp3 hold some advantage over SALT in highly competitive games (Figure 13), the data suggests that SALT is better suited for general-sum matrix games.

SALT's robustness is further demonstrated by considering evolutionary settings. To this end, consider a large society of players employing learning algorithms, each with similar reasoning capacity. In each generation, each member of the society is randomly paired with another member of the society in one of 50 repeated (200,000 iterations) randomly-chosen, 5-action matrix games. In the first generation, each player in the society is equally likely to employ each of five learning algorithms (SALT, the S-algorithm, WoLF-PHC, Q-learning, and Exp3). In each subsequent generation, the society evolves using the replicator dynamic [41]; the percentage of the population using each learning algorithm is modified proportionally to the average score obtained by the players employing that learning algorithm in the previous generation. The composition of the population for each generation is shown in Figure 19. The figure shows that SALT players quickly dominate the society, which results from SALT's secure and efficient behavior.

Our empirical studies reveal several other noteworthy observations about SALT's evolutionary behavior in in these randomly-chosen general-sum games. First, any number of SALT players can invade WoLF-PHC, Q-learning, and the S-algorithm. Second, SALT can invade Exp3 as long as it is paired with another SALT agent at least 10.16% of the time, similar to how tit-for-tat can invade a society of always defect agents in the repeated Prisoner's Dilemma if tit-for-tat meets other tit-for-tat players often enough [2]. Third, none of

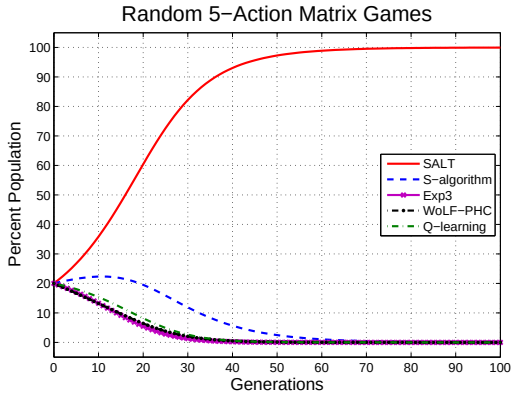


Figure 19: Percent of a population employing each learning algorithm per generation in 50 randomly-chosen, 2-player, 5-action general-sum matrix games.

the other learning algorithms can invade SALT. This robustness is due to SALT's ability to be (a) efficient in self play, (b) secure enough to avoid exploitation, and (c) perform reasonably well when associating with other learners.

While SALT empirically demonstrates evolutionary stability in randomly-chosen general-sum games played with Exp3, WoLF-PHC, Q-learning, and the S-algorithm, results vary from game to game. As an example, we investigate the evolutionary dynamics of the algorithms in the Prisoner's Dilemma and the 2-player matrix games in Table 3. In each game, each of the five learning algorithms was equally represented in the first generation. Table 7 shows the percentage of the population employing each algorithm after 1000 generations. The third and fourth columns show the results when the S-algorithm and SALT, respectively, are held out of the initial population.

We emphasize several observations from Table 7. First, when both SALT and the S-algorithm are included in the initial population, either SALT, the S-algorithm, or both eventually dominate the society in all games except Battle of the Sexes. This demonstrates the robustness of satisficing learning in general. Second, as expected from the efficiency results shown in Figure 15, SALT dominates the population in the Staghunt, the Preclusion Game, the Security Game, and randomly-chosen games rather than the S-algorithm due to SALT's ability to learn to tremble its aspirations to satisfy the conditions of pareto efficiency. Third, in Shapley's Game, Tricky Game, and the Prisoner's Dilemma, the S-algorithm gains a higher share of the population than SALT. However, the differences in performance between the two algorithms in these games are quite small. In each case, SALT's performance is slightly lower than the S-algorithm's due to small amounts of explorations (via aspiration trembles). These slight differences allow the S-algorithm to obtain a higher share of the population after 1000 generations. When the S-algorithm is not present in the initial population in these games, SALT eventually dominates the society.

7 Conclusions and Future Work

Most past and current research in multi-agent learning has focused either on the (myopic) best response or on regret minimization. However, Axelrod's seminal work in repeated games [2] suggests that a robust multi-agent learning algorithm should be *secure*, meaning that it avoids being exploited by antagonistic associates, and *efficient*, meaning that it learns near pareto efficient solutions when associates are inclined to cooperate. Best-response and regret-minimization algorithms in the literature do not satisfy both of these properties simultaneously. In fact, no learning algorithm previously found in the literature satisfies both of these properties in minimal information environments.

Game	Population	Population w/o S-algorithm	Population w/o SALT
Battle of the Sexes	Q-learning (100%)	Q-learning (100%)	Q-learning (100%)
Chicken	SALT (100%)	SALT (100%)	S-algorithm (100%)
Shapley's Game	S-algorithm (98%) SALT (2%)	SALT (100%)	S-algorithm (100%)
Staghunt	SALT (90%) S-algorithm (10%)	SALT (100%)	S-algorithm (100%)
Tricky Game	S-algorithm (90%) SALT (10%)	SALT (100%)	S-algorithm (100%)
Preclusion Game	SALT (100%)	SALT (100%)	S-algorithm (100%)
Security Game	SALT (100%)	SALT (100%)	S-algorithm (100%)
Prisoner's Dilemma	S-algorithm (84%) SALT (16%)	SALT (100%)	S-algorithm (100%)
Random Games	SALT (100%)	SALT (100%)	S-algorithm (100%)

Table 7: Population composition in various games after 1000 generations. Algorithms not listed consisted of less than 0.5% of the population after 1000 generations.

In this paper, we studied a very simple but powerful learning algorithm called the S-algorithm [19, 36], and analyzed it with respect to the security and efficiency properties in general-sum matrix games. The S-algorithm, which formalizes Simon's notion of satisficing [33, 34], uses an aspiration relaxation search to identify a sustainable payoff and a behavior to sustain it. We showed that when certain technical conditions are satisfied and when initial aspiration levels are set appropriately, the S-algorithm is efficient in self play. The technical conditions are satisfied in a wide-majority of matrix games, including all 2-player matrix games and all well-studied matrix games in the literature. The constraint on initial aspirations, however, is often more severe. We introduced a trembling-hand version of the S-algorithm, called SALT, that learns to tremble aspiration levels in order to remove the constraint on initial aspirations. The resulting algorithm is efficient in self play in a vast majority of general-sum matrix games played in minimal information environments. SALT is also provably secure, while the S-algorithm is not.

Due to its efficiency and security properties, SALT is a robust learning algorithm for general-sum matrix games played with other learning algorithms with similar representational and reasoning capacities, as demonstrated by its performance in head-to-head associations with other learning algorithms as well as from an evolutionary perspective. These results suggest that robust learning algorithms for repeated general-sum games should emphasize the properties of efficiency and security rather than focus exclusively on learning to play a best response (i.e., Nash equilibrium) or to minimize regret.

A number of extensions and modifications of SALT would be valuable topics of future work. First, we noted in Section 6.2.1 that SALT is somewhat sensitive to its parameter settings as the number of players in the game increases. It would be interesting and useful to alter the algorithm to overcome these sensitivities, such as designing alternate methods for determining when and how to tremble aspirations. Second, we note that SALT provides a natural mechanism for combining itself with an algorithm that has a stronger notion of security. Rather than resorting to the pure-strategy maximin action following an aspiration tremble to or below its maximin value, it could trigger its behavior to that of a learning algorithm with a stronger security guarantee (such as Exp3) for the rest of the epoch.

The concepts of satisficing learning can be extended to situations in which more information about the environment is available to an agent. For example, when an agent can view the actions of other agents, it should use this information to encode more representational and reasoning capacity. The M-Qubed algorithm [9] combines this information with concepts of satisficing learning in order to satisfy tighter notions of security

(it is secure with respect to the mixed-strategy space) and efficiency (it can learn coordinated, efficient solutions in games such as Shapley's Game). Future work should continue to study how the principles of satisficing learning can be incorporated into algorithms with more reasoning capacity in order to produce more robust multi-agent learning algorithms.

Appendix A

SALT is secure when the following assumptions are satisfied:

1. If, on average, SALT is exploited when it trembles its aspiration level to γ, then $U_i^t(\gamma) < r_i^{\text{mm}}$. This requirement is ensured by Eq. (7) for small enough σ_i, since $U_i^t(\gamma)$ is based on the agent's accumulated payoffs ($u_i^j \kappa_i^j$) received in each epoch.

2. SALT trembles to a value $\gamma \leq r_i^{\text{mm}}$ at least once. Since, for all t, $r_i^{\text{mm}} \in \Gamma_i(t)$, it is highly likely that SALT will eventually tremble to some value $\gamma \leq r_i^{\text{mm}}$ as long as it (a) never stops trembling (i.e., $\eta_i^t > 0$ for all t) and (b) there is always a non-zero probability that some $r_i^{\text{mm}} \in \Gamma_i(t)$ will be chosen. These assumptions are satisfied in Eqs. (10) and (8), respectively.

3. Let Θ_i^t be the set of trembles values γ_i^* such that $U_i^t(\gamma_i^*) \geq \max_{\gamma \in \Gamma_i(t)} U_i^t(\gamma) - \varepsilon_i$. Then, $Pr(\gamma_i^\tau \notin \Theta_i^t) \to 0$ as $t \to \infty$. In words, the probability that SALT does not select the tremble value with the highest utility approaches zero over time. As long as SALT continues to tremble (i.e., $\eta_i^t > 0$ for all t; see Eq. (10)), Eq. (8) ensures that this assumption holds when $T_i \to 0$ as $t \to \infty$.

Theorem 14 *SALT is secure in all repeated general-sum matrix games regardless of the behavior of its associates.*

Proof. SALT always plays its maximin action in an epoch in which it trembles to or below r_i^{mm}. Thus, given assumption 2, $U_i^t(r_i^{\text{mm}}) \geq r_i^{\text{mm}}$ for large enough t. When SALT is being exploited, assumption 1 requires that $\arg\max_{\gamma \in \Gamma_i(t)} U_i^t(\gamma) \leq r_i^{\text{mm}}$. Thus, given assumption 3, SALT will eventually learn to always tremble to its maximin value when it is being exploited, which means that it will always play its maximin strategy. □

References

[1] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. Gambling in a rigged casino: the adversarial multi-armed bandit problem. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 322–331. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[2] R. M. Axelrod. *The Evolution of Cooperation.* Basic Books, 1984.

[3] U. Berger. Fictitious play in 2xN games. *Journal of Economic Theory*, 120(2):139–254, 2005.

[4] M. Bowling. Convergence problems of general-sum multiagent reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning*, pages 89–94, 2000.

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

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

[7] Y. Chang and L. P. Kaelbling. Hedge learning: Regret-minimization with learning experts. In *Proceedings of the 22nd International Conference on Machine Learning*, pages 121–128, 2005.

[8] J. W. Crandall. *Learning Successful Strategies in Repeated General-Sum Games*. Doctoral dissertation, Brigham Young University, Provo, Utah, USA, 2005.

[9] J. W. Crandall and M. A. Goodrich. Learning to compete, compromise, and cooperate in repeated general-sum games. In *Proceedings of the* 22^{nd} *International Conference on Machine Learning*, Bonn, Germany, 2005.

[10] J. W. Crandall and M. A. Goodrich. Learning to teach and follow in repeated games. In *AAAI workshop on Multiagent Learning*, Pittsburgh, PA, July 2005.

[11] D. de Farias and N. Megiddo. How to combine expert (or novice) advice when actions impact the environment. In *Advances in Neural Information Processing Systems 16*, 2004.

[12] N. Feltovich. Reinforcement-based vs. belief-based learning models in experimental assymetric-information games. *Econometrica*, 68(3):605–641, 2000.

[13] D. P. Foster and R. Vohra. Regret in the on-line decision problem. *Games and Economic Behavior*, 29:7–35, 1999.

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

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

[16] A. Greenwald and K. Hall. Correlated Q-learning. In *Proceedings of the* 20^{th} *International Conference on Machine Learning*, pages 242–249, 2003.

[17] J. Henrich, R. Boyd, S. Bowles, C. Camerer, E. Fehr, H. Gintis, R. McElreath, M. Alvard, A. Barr, J. Ensminger, N. Smith Henrich, K. Hill, F. Gil-White, M. Gurven, F. W. Marlowe, J. Q. Patton, and D. Tracer. "economic man" in cross cultural perspectives: Behavioral experiments in 15 small-scale societies. *Behavioral and Brain Sciences*, 28:795–855, 2005.

[18] J. Hu and M. Wellman. Multiagent reinforcement learning: Theoretical framework and and algorithm. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 242–250, 1998.

[19] R. Karandikar, D. Mookherjee, D. Ray, and F. Vega-Redondo. Evolving aspirations and cooperation. *Journal of Economic Theory*, 80:292–331, 1998.

[20] N. Littlestone and M. K. Warmuth. The weighted majority algorithm. *Information and Computation*, 108:(2):212–261, 1992.

[21] M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the* 11^{th} *International Conference on Machine Learning*, pages 157–163, 1994.

[22] M. L. Littman. Friend-or-foe: Q-learning in general-sum games. In *Proceedings of the* 18^{th} *International Conference on Machine Learning*, pages 322–328, 2001.

[23] M. L. Littman and P. Stone. Leading best-response strategies in repeated games. In *IJCAI workshop on Economic Agents, Models, and Mechanisms*, Seattle, WA, August 2001 2001.

[24] M. L. Littman and P. Stone. A polynomial-time nash equilibrium algorithm for repeated games. In *Proceedings of 2003 ACM Conference on Electronic Commerce*, San Diego, CA, USA, June 9-12 2003.

[25] D. Monderer and L. S. Shapley. Potential games. *Games and Economic Behavior*, 14:124–143, 1996.

[26] J. Nachbar. Evolutionary selection dynamics in games: Convergence and limit properties. *International Journal of Game Theory*, 19:59–89, 1990.

[27] J. F. Nash. The bargaining problem. *Econometrica*, 18:155–162, 1950. Reprinted in *Classics in Game Theory*, H. W. Kuhn, ed.

[28] E. Nudelman, J. Wortman, K. Leyton-Brown, and Y. Shoham. Run the GAMUT: A comprehensive approach to evaluating game-theorectic algorithms. In *Proceedings of the 3^{rd} International Conference on Autonomous Agents and Multi-agent Systems*, 2004.

[29] R. Powers and Y. Shoham. Learning against opponents with bounded memory. In *Proceedings of the 19^{th} International Joint Conference on Artificial Intelligence*, pages 817–822, 2005.

[30] J. Robinson. An iterative method of solving a game. *Annals of Mathematics*, 54:296–301, 1951.

[31] T. Sandholm and R. Crites. Multiagent reinforcement learning in the iterated prisoner's dilemma. *Biosystems*, 37:147–166, 1995. Special Issue on the Prisoner's Dilemma.

[32] S. Sen, S. Airiau, and R. Mukherjee. Towards a pareto-optimal solution in general-sum games. In *Proceedings of the 2^{nd} International Conference on Autonomous Agents and Multi-agent Systems*, pages 153–160, 2003.

[33] H. A. Simon. A behavioral model of rational choice. *Quart. J. Economics*, 59:99–118, 1955.

[34] H. A. Simon. *The Sciences of the Artificial*. The MIT Press, 3rd edition, 1996.

[35] J. L. Stimpson. Satisficing solutions to a multi-agent social dilemma. Master's thesis, Brigham Young University, Provo, UT, 84602, USA, 2002.

[36] J. L. Stimpson and M. A. Goodrich. Learning to cooperate in a social dilemma: A satisficing approach to bargaining. In *Proceedings of ICML*, Washington D.C., 2003.

[37] J. L. Stimpson, M. A. Goodrich, and L. C. Walters. Satisficing and learning cooperation in the prisoner's dilemma. In *Proceedings of the 2001 International Joint Conference on Artificial Intelligence*, Seattle, Washington, 2001.

[38] W. C. Stirling. *Satisficing Games and Decision Making*. Cambridge University Press, 2003.

[39] W. C. Stirling and M. A. Goodrich. Satisficing games. *Information Sciences*, 114:255–280, March 1999.

[40] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1998.

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

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