Chapter 4
Learning in Multiplayer Stochastic Games

4.1 Introduction

The agents in a multiagent system can be to some degree preprogrammed with behaviors designed in advance. It is often necessary that the agents be able to learn online such that the performance of the multiagent system improves. However, typically a multiagent system is very complex and preprogramming the system is for practical reasons impossible. Furthermore, the dynamics of the agents and the environment can change over time and learning and adaptation is required.

In early work on multiagent reinforcement learning (MARL) for stochastic games [1], it was recognized that no agent works in a vacuum. In his seminal paper, Littman [1] focused on only two agents that had opposite and opposing goals. This means that they could use a single reward function which one tried to maximize and the other tried to minimize. The agent had to work with a competing agent and had to behave so as to maximize their reward in the worst possible case. They also recognized the need for mixed strategies because the agent or player could not be certain of the action taken by its opponent. Littman [1] introduced the minimax Q-learning algorithm. We have already shown the idea of the minimax Q-learning algorithm in Chapter 3, Section 3.2.

Multi-Agent Machine Learning: A Reinforcement Approach, First Edition. Howard M. Schwartz.

In a rational multiagent game, each agent must keep track in some way of what the other learning agents are doing. The types of games and situations that the learning agent may encounter include fully competitive games that are zero-sum games. There are also general-sum cooperative games, where the agents try to get the maximum reward by cooperating with each other. For example, in the prisoners' dilemma problem, if it is a cooperative game, then each player should lie to the police and cooperate with each other and get the least time in jail. However, if it is a competitive game, then they should each defect to mitigate against the worst case scenario where one lies to police and the opponent confesses and the first one goes to jail for life. However, the agents must communicate to cooperate with each other.

Typically, in a multiagent system, the agents must keep track of the other agents' behaviors such that a coherent behavior results. Furthermore, we have the issue of scalability to consider. The agents must keep track of a large number of possible states and joint actions.

Learning in stochastic games can be formalized as a MARL problem [2]. Agents select actions simultaneously at the current state and receive rewards at the next state. Different from the algorithm that can solve for a Nash equilibrium in a stochastic game, the goal of a reinforcement learning algorithm is to learn equilibrium strategies through interaction with the environment. Generally, in a MARL problem, agents may not know the transition function or the reward function from the environment. Instead, agents are required to select actions and observe the received reward and the next state in order to gain information of the transition function or the reward function.

Rationality and convergence are two desirable properties for multiagent learning algorithms in stochastic games [2]. When we say that a player is rational, we mean that, if the other players' policies converge to stationary policies, then the learning algorithm for will converge to a policy that is a best response to the other players' policies. What does this mean? Let us say that you are playing the matching pennies game against a *bad* player who always plays heads. You win whenever you both play heads or tails. At first, you assume that your opponent is a rational player and, therefore, you assume that your opponent is playing the rational strategy of playing heads and tails each 50% of the time. Therefore, you start out playing heads and tails 50% of the time. However, after a number of plays you realize that your opponent always seems to be playing heads as his stationary strategy. You would then quickly shift and begin to also play heads all the time. Then you always win and you are playing the rational strategy but your opponent is not playing a rational strategy.

In the stochastic learning algorithms, we also have the idea of convergence. Let us say that all the other players are playing a stationary strategy. They are not learning or changing their strategy in any way. Then you adapt to this behavior and converge to some rational strategy. Or, let us say that everyone in the game is adapting according to the same algorithm. Then do all the players converge to an optimal strategy or a Nash equilibrium? If all the players use rational learning algorithms and their policies converge, then they must have converged to an equilibrium. Each player will play the best response to all other players.

In this chapter, we review some existing reinforcement learning algorithms in stochastic games. We analyze these algorithms based on their applicability, rationality, and convergence properties.

Isaacs [3] introduced a differential game of guarding a territory where a defender tries to intercept an invader before the invader reaches the territory. In this chapter, we introduce a grid version of Isaacs' game called the *grid game of guarding a territory*. It is a two-player zero-sum stochastic game where the defender plays against the invader in a grid world. We then study how the players learn to play the game using MARL algorithms. We apply two reinforcement learning algorithms to this game and test the performance of these learning algorithms based on the convergence and rationality properties.

4.2 Multiplayer Stochastic Games

A Markov decision process contains a single player and multiple states, whereas a matrix game contains multiple players and a single state. For a game with more than one player and multiple states, we define a stochastic game (or Markov game) as the combination of Markov decision processes and matrix games. A stochastic game is a tuple $(n, S, A_1, \ldots, A_n, T, \gamma, R_1, \ldots, R_n)$ where n is the number of the players, $T : S \times A_1 \times \cdots \times A_n \times S \to [0, 1]$ is the transition function, $A_i (i = 1, \ldots, n)$ is the action set for the player i, $\gamma \in [0, 1]$ is the discount factor, and $R_i : S \times A_1 \times \cdots \times A_n \times S \to \mathbb{R}$ is the reward function for player i. The transition function in a stochastic game is a probability distribution over next states given the current state and joint action of the players. The reward function $R_i(s, a_1, \ldots, a_n, s')$ denotes the reward received by player i in state s' after taking joint action $(a_1, \ldots, a_n)$ in state s. Similar to Markov decision processes, stochastic games also have the Markov property. That is, the player's next state and reward depend only on the current state and all the players' current actions.

For a multiplayer stochastic game, we want to find the Nash equilibria in the game if we know the reward function and transition function in the game. A Nash equilibrium in a stochastic game can be described as a tuple of n strategies $(\pi_1^*, \ldots, \pi_n^*)$ such that for all $s \in S$ and $i = 1, \ldots, n$,

$$V_i(s, \pi_1^*, \ldots, \pi_i^*, \ldots, \pi_n^*) \geq V_i(s, \pi_1^*, \ldots, \pi_i, \ldots, \pi_n^*) \text{ for all } \pi_i \in \Pi_i \quad (4.1)$$

where Π_i is the set of strategies available to player i, and $V_i(s, \pi_1^*, \ldots, \pi_n^*)$ is the expected sum of discounted rewards for player i given the current state and all the players' equilibrium strategies. To simplify notation, we use $V_i^*(s)$ to represent $V_i(s, \pi_1^*, \ldots, \pi_n^*)$ as the state-value function under Nash equilibrium strategies. We can also define the action-value function $Q^*(s, a_1, \ldots, a_n)$ as the expected sum of discounted rewards for player i given the current state and the current joint action of all the players, and following the Nash equilibrium strategies thereafter. Then we can get

$$V_i^*(s) = \sum_{a_1, \ldots, a_n \in A_1 \times \cdots \times A_n} Q_i^*(s, a_1, \ldots, a_n) \pi_1^*(s, a_1) \cdots \pi_n^*(s, a_n) \quad (4.2)$$

$$Q_i^*(s, a_1, \ldots, a_n) = \sum_{s' \in S} T(s, a_1, \ldots, a_n, s')[R_i(s, a_1, \ldots, a_n, s') + \gamma V_i^*(s')] \quad (4.3)$$

where $\pi_i^*(s, a_i) \in \text{PD}(A_i)$ is a probability distribution over action a_i under player i's Nash equilibrium strategy, $T(s, a_1, \ldots, a_n, s') = \Pr\{s_{k+1} = s' | s_k = s, a_1, \ldots, a_n\}$ is the probability of the next state being s' given the current state s and joint action $(a_1, \ldots, a_n)$, and $R_i(s, a_1, \ldots, a_n, s')$ is the expected immediate reward received in state s' given the current state s and joint action $(a_1, \ldots, a_n)$. Based on (4.2) and (4.3), the Nash equilibrium in (4.1) can be rewritten as

$$\sum_{a_1, \ldots, a_n \in A_1 \times \cdots \times A_n} Q_i^*(s, a_1, \ldots, a_n) \pi_1^*(s, a_1) \cdots \pi_i^*(s, a_i) \cdots \pi_n^*(s, a_n)$$
$$\geq \sum_{a_1, \ldots, a_n \in A_1 \times \cdots \times A_n} Q_i^*(s, a_1, \ldots, a_n) \pi_1^*(s, a_1) \cdots \pi_i(s, a_i) \cdots \pi_n^*(s, a_n) \quad (4.4)$$

Stochastic games can be classified on the basis of the players' reward functions. If all the players have the same reward function, the game is called a *fully cooperative game* or a *team game*. If one player's reward function has always the opposite sign of the other player's, the game is called a *two-player fully competitive game* or *zero-sum game*. For the game with all types of reward functions, we call it a general-sum stochastic game.

To solve a stochastic game, we need to find a strategy $\pi_i : S \to A_i$ that can maximize player i's discounted future reward with a discount factor γ. Similar

to the strategy in matrix games, the player's strategy in a stochastic game is probabilistic. An example is the soccer game introduced by Littman [1], where an agent on the offensive side must use a probabilistic strategy to pass an unknown defender. In the literature, a solution to a stochastic game is described as Nash equilibrium strategies in a set of associated *state-specific matrix games* [1, 4]. A state-specific matrix game is also called a *stage game*. In these state-specific matrix games, we define the action-value function $Q_i^*(s, a_1, \ldots, a_n)$ as the expected reward for player i when all the players take joint action $a_1, \ldots, a_n$ in state s and follow the Nash equilibrium strategies thereafter. If the value of $Q_i^*(s, a_1, \ldots, a_n)$ is known for all the states, we can find player i's Nash equilibrium strategy by solving the associated state-specific matrix game [4]. Therefore, for each state s, we have a matrix game and we can find the Nash equilibrium strategies in this matrix game. Then the Nash equilibrium strategies for the game are the collection of Nash equilibrium strategies in each state-specific matrix game for all the states. We present an example here.

Example 4.1 We define a 2×2 grid game with two players denoted as $P1$ and $P2$. Two players' initial positions are located at the bottom left corner for player 1 and the upper right corner for player 2, as shown in Fig. 4-1a. Both players try to reach one of the two goals denoted as "G" in minimum number of steps. Starting from their initial positions, each player has two possible moves which are moving up or right for player 1, and moving left or down for player 2. Figure 4-1b shows the numbered cells in this game. Each player takes an action and moves one cell at a time. The game ends when either of the players reaches the goal and receives a reward 10. The dashed line between the upper cells and the bottom cells in Figure 4-1a is the barrier that the player can pass through with a probability 0.5. If both players move to the same cell, both players bounce back to their original positions. Figure 4-1c shows the possible transitions in the game. The number of possible states (players' joint positions) is seven containing the players' initial positions $s_1 = (2, 3)$ and six terminal states $(s_2, \ldots, s_7)$.

According to the above description of the game, we can find the Nash equilibrium strategies in this example. The players need to avoid the barrier and move to the goals next to them without crossing the barrier. Therefore, the Nash equilibrium is the players' joint action ($a_1 =$ Right, $a_2 =$ Left). Based on (4.2) and (4.3), the state-value function $V_i^*(s_1)$ under the Nash equilibrium strategies is

$$\begin{aligned} V_i^*(s_1) &= R_i(s_1, \text{Right}, \text{Left}, s_7) + \gamma V_i^*(s_7) \\ &= 10 + 0.9 \cdot 0 = 10 \end{aligned} \tag{4.5}$$

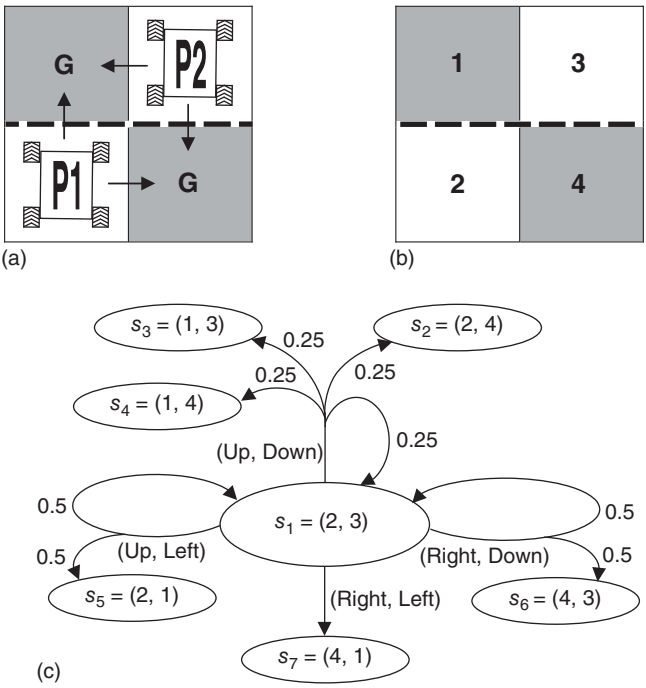


Fig. 4-1. Example of stochastic games. (a) A 2 × 2 grid game with two players. (b) The numbered cells in the game. (c) Possible state transitions given players' joint action (a_1, a_2). Reproduced from [5], © X. Lu.

where $\gamma = 0.9$, $R_i(s_1, \text{Right}, \text{Left}, s_7) = 10$, and $V_i^*(s_7) = 0$ (the state-value functions at terminal states are always zero). We can also find the action-value function $Q_i^*(s_1, a_1, a_2)$. For example, the action-value function $Q_1^*(s_1, \text{Up}, \text{Down})$ for player 1 can be written as

$$\begin{aligned} Q_1^*(s_1, \text{Up}, \text{Down}) &= \sum_{s'=s_1 \sim s_4} T(s_1, \text{Up}, \text{Down}, s')[R_1(s_1, \text{Up}, \text{Down}, s') + \gamma V_1^*(s')] \\ &= 0.25(0 + 0.9V_1^*(s_1)) + 0.25(0 + 0.9V_1^*(s_2)) \\ &\quad + 0.25(10 + 0.9V_1^*(s_3)) + 0.25(10 + 0.9V_1^*(s_4)) \\ &= 0.25 \cdot 0.9 \cdot 10 + 0.25 \cdot 0 + 0.25 \cdot 10 + 0.25 \cdot 10 \\ &= 7.25 \end{aligned} \tag{4.6}$$

Table 4.1 shows the action-value functions under the players' Nash equilibrium strategies.

Table 4.1 Action-value function $Q_i^*(s_1, a_1, a_2)$ in Example 4.1.

		a_2	
	$Q_1^*(s_1, a_1, a_2)$	Left	Down
a_1	Up	4.5	7.25
	Right	10	9.5

		a_2	
	$Q_2^*(s_1, a_1, a_2)$	Left	Down
a_1	Up	9.5	7.25
	Right	10	4.5

4.3 Minimax-Q Algorithm

Littman [1] proposed a minimax-Q algorithm specifically designed for two-player zero-sum stochastic games. The minimax-Q algorithm uses the minimax principle to solve for players' Nash equilibrium strategies and values of states for two-player zero-sum stochastic games. Similar to Q-learning, minimax-Q algorithm is a temporal-difference learning method that performs backpropagation on values of states or state-action pairs. We show the minimax-Q algorithm as follows:

In a two-player zero-sum stochastic game, given the current state s, we define the state-value function for player i as

$$V_i^*(s) = \max_{\pi_i(s,\cdot)} \min_{a_{-i} \in A_{-i}} \sum_{a_i \in A_i} Q_i^*(s, a_i, a_{-i}) \pi_i(s, a_i), \quad (i = 1, 2) \tag{4.7}$$

where $-i$ denotes player i's opponent, $\pi_i(s, \cdot)$ denotes all the possible strategies of player i at state s, and $Q_i^*(s, a_i, a_{-i})$ is the expected reward when player i and its opponent choose action $a_i \in A_i$ and $a_{-i} \in A_{-i}$, respectively, and follow their Nash equilibrium strategies after that. If we know $Q_i^*(s, a_i, a_{-i})$, we can solve Eq. (4.7) and find player i's Nash equilibrium strategy $\pi^*(s)$. Similar to finding the minimax solution for (3.8), one can use linear programming to solve Eq. (4.7). For a MARL problem, $Q_i^*(s, a_i, a_{-i})$ is unknown to the players in the game.

The minimax-Q algorithm is listed in Algorithm 4.1. The minimax-Q algorithm can guarantee the convergence to a Nash equilibrium if all the possible states and players' possible actions are visited infinitely often. The proof of convergence for the minimax-Q algorithm can be found in Reference 6. One drawback of this algorithm is that we have to use linear programming to solve for $\pi_i(s)$ and $V_i(s)$ at each iteration in Algorithm 4.1.

Algorithm 4.1 Minimax-Q algorithm

1: Initialize $Q_i(s, a_i, a_{-i})$, $V_i(s)$ and π_i
2: **for** Each iteration **do**
3: Player i takes an action a_i from current state s based on an exploration-exploitation strategy
4: At the subsequent state s', player i observes the received reward r_i and the opponent's action taken at the previous state s.
5: Update $Q_i(s, a_i, a_{-i})$:

$$Q_i(s, a_i, a_{-i}) \leftarrow (1 - \alpha) Q_i(s, a_i, a_{-i}) + \alpha \left[r_i + \gamma V_i(s') \right] \tag{4.8}$$

where α is the learning rate and γ is the discount factor.
6: Use linear programming to solve Eq. (4.7) and obtain the updated $\pi_i(s)$ and $V_i(s)$
7: **end for**

This will lead to a slow learning process. Also, in order to perform linear programming, the player i has to know the opponent's action space.

Using the minimax-Q algorithm, the player will always play a "safe" strategy in case of the worst scenario caused by the opponent. However, if the opponent is currently playing a stationary strategy which is not its equilibrium strategy, the minimax-Q algorithm cannot make the player adapt its strategy to the change in the opponent's strategy. The reason is that the minimax-Q algorithm is an opponent-independent algorithm and it will converge to the player's Nash equilibrium strategy no matter what strategy the opponent uses. If the player's opponent is a weak opponent that does not play its equilibrium strategy, then the player's optimal strategy is not the same as its Nash equilibrium strategy. The player's optimal strategy will do better than the player's Nash equilibrium strategy in this case.

Overall, the minimax-Q algorithm, which is applicable to zero-sum stochastic games, does not satisfy the rationality property but satisfies the convergence property. The following example of a 2×2 grid game demonstrates the algorithm.

4.3.1 2×2 Grid Game

The playing field of the 2×2 grid game is shown in Fig. 4-2. The territory to be guarded is located at the bottom-right corner. Initially, the invader starts

at the top-left corner while the defender starts at the same cell as the territory. To better illustrate the guarding a territory problem, we simplify the possible number of actions of each player to 2. The invader can only move down or right, while the defender can only move up or left. The capture of the invader happens when the defender and the invader move into the same cell excluding the territory cell. The game ends when the invader reaches the territory or the defender catches the invader before it reaches the territory. We suppose both players start from the initial state s_1, as shown in Fig. 4-2a. There are three nonterminal states (s_1, s_2, s_3) in this game shown in Fig. 4-2. If the invader moves to the right cell and the defender happens to move left, then both players reach the state s_2 in Fig. 4-2b. If the invader moves down and the defender moves up simultaneously, then they will reach the state s_3 in Fig. 4-2c. In states s_2 and s_3, if the invader is smart enough, it can always reach the territory no matter what action the defender will take. As such, the game only has one step because, if the invader gets to state s_2 or s_3, it de facto wins. Therefore, starting from the initial state s_1, a clever defender will try to intercept the invader by guessing which direction the invader will go.

We define the reward functions for the players. The reward function for the defender is defined as

$$R_D = \begin{cases} dist_{IT}, & \text{defender captures the invader} \\ -10, & \text{invader reaches the territory} \end{cases} \tag{4.9}$$

where

$$dist_{IT} = |x_I(t_f) - x_T| + |y_I(t_f) - y_T|$$

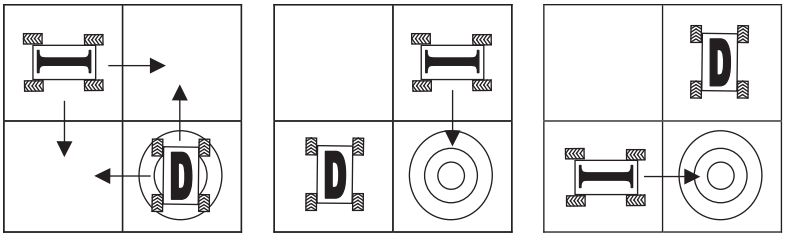


Fig. 4-2. A 2 × 2 grid game. (a) Initial positions of the players: state s_1. (b) Invader in top-right versus defender in bottom-left: state s_2. (c) Invader in bottom-left versus defender in top-right: state s_3. Reproduced from [5], © X. Lu.

The reward function for the invader is given by

$$R_I = \begin{cases} -dist_{IT}, & \text{defender captures the invader} \\ 10, & \text{invader reaches the territory} \end{cases} \tag{4.10}$$

The reward functions (4.9) and (4.10) are also used in the 6×6 grid game.

Before the simulation, we can simply solve this game similar to solving Example 4.1. In the states s_2 and s_3, a smart invader will always reach the territory without being intercepted. The value of the states s_2 and s_3 for the defender will be $V_D(s_2) = -10$ and $V_D(s_3) = -10$. We set the discount factor as 0.9 and we can get $Q_D^*(s_1, a_{left}, o_{right}) = \gamma V_D(s_2) = -9$, $Q_D^*(s_1, a_{up}, o_{down}) = \gamma V_D(s_3) = -9$, $Q_D^*(s_1, a_{left}, o_{down}) = 1$, and $Q_D^*(s_1, a_{up}, o_{right}) = 1$, as shown in Table 4.2 and Table 4.3a. Under the Nash equilibrium, we define the probabilities of the defender moving up and left as $\pi_D^*(s_1, a_{up})$ and $\pi_D^*(s_1, a_{left})$, respectively. The probabilities of the invader moving down and right are denoted as $\pi_I^*(s_1, o_{up})$ and $\pi_I^*(s_1, o_{left})$, respectively. Based on the Q-values in Table 4.3a, we can find the value of the state s_1 for the defender by solving a linear programming problem shown in Table 4.3b. The approach for solving a linear programming problem can be found in Section 3.2.

We define $R(s, a, o)$ to denote the immediate reward to the agent for taking action $a \in A$ in state $s \in S$ when its opponents take action o $\in O$. The idea is for the agent to maximize its reward in the worst possible case. In other words, the agent assumes that the other agent is playing its best possible strategy in a purely competitive game. Therefore, the agent's optimal policy would be to maximize the minimum possible reward. Just like in the case of the prisoners' dilemma, the action should be to defect because this will limit cost of the worst possible outcome regardless of what the other player does. If the prisoner/player decides not to defect, and if the other player does defect, then the prisoner/player will go to jail for a long time; therefore, to maximize the worst possible outcome, the prisoner/player should defect.

Now imagine that we would be satisfied with a policy that guarantees an expected reward $\overline{R}$ no matter what the other player, in this case an opponent, chooses as its action. In this case, the defender's reward matrix in state s_1 is

$$R_D(s_1) = \begin{bmatrix} -9 & 1 \\ 1 & -9 \end{bmatrix} \tag{4.11}$$

Recall that the policy in this case is given as the probability that the agent in state s_1 will choose to go up, $\pi_D(s_1, a_{up})$, and the probability that the agent will choose to go left, $\pi_D(s_1, a_{left})$.

Table 4.2 Minimax solution for the defender in the state s_1.

		Defender	
	Q_D^*	Up	Left
Invader	Down	–9	1
	Right	1	–9

Then we get the following equations for the expected reward, regardless of which action the opponent takes:

$$(-9) \cdot \pi_D(s_1, a_{up}) + (1) \cdot \pi_D(s_1, a_{left}) = \bar{R}$$
$$(1) \cdot \pi_D(s_1, a_{up}) + (-9) \cdot \pi_D(s_1, a_{left}) = \bar{R}$$
$$\pi_D(s_1, a_{up}) + \pi_D(s_1, a_{left}) = 1$$

Therefore, the goal is to maximize the expected reward $\bar{R}$ regardless of what the opponent does.

After solving the linear constraints in Table 4.3b, we get the value of the state s_1 for the defender as $V_D(s_1) = -4$ and the Nash equilibrium strategy for the defender as $\pi_D^*(s_1, a_{up}) = 0.5$ and $\pi_D^*(s_1, a_{left}) = 0.5$. For a two-player zero-sum game, we can get $Q_D^* = -Q_I^*$. Similar to the approach in Table 4.3, we can find the minimax solution of this game for the invader as $V_I(s_1) = 4$, $\pi_I^*(s_1, o_{down}) = 0.5$, and $\pi_I^*(s_1, o_{right}) = 0.5$. Therefore, the Nash equilibrium strategy of the invader is to move down or right with probability 0.5, and the Nash equilibrium strategy of the defender is to move up or left with probability 0.5.

Table 4.3 Minimax solution for the defender in the state s_1. (a) Q-values of the defender for the state s_1. (b) Linear constraints for the defender in the state s_1.

(a)

		Defender	
	Q_D^*	Up	Left
Invader	Down	–9	1
	Right	1	–9

(b)

Objective: Maximize $\bar{R}$
$(-9) \cdot \pi_D(s_1, a_{up}) + (1) \cdot \pi_D(s_1, a_{left}) \geq \bar{R}$
$(1) \cdot \pi_D(s_1, a_{up}) + (-9) \cdot \pi_D(s_1, a_{left}) \geq \bar{R}$
$\pi_D(s_1, a_{up}) + \pi_D(s_1, a_{left}) = 1$

The implementation of the minimax Q-learning algorithm begins by initializing the Q matrix, the value function $V(s)$, and the policy $\pi_i(a)$ for each agent. In the example shown here, we initialize $Q(s, a_i, a_{-i}) = 0$ and $V_i(s) = 0$. We arbitrarily initialized the probability of the defender to go up as $\pi_d(up) = 1.0$ and $\pi_d(left) = 0.0$ and of the invader $\pi_i(right) = 1.0$ and $\pi_i(down) = 0.0$. We set the discount factor as $\gamma = 0.9$, the learning rate to $\alpha = 0.1$, and the exploration probability to $\varepsilon = 0.1$. The constraint condition for the linear programming problem for the defender is

$$\begin{aligned} R_D(s, up, right)\pi(s_1, up) + R_D(s, left, right)\pi(s_1), left) \geq V_D(s_1) \\ R_D(s, up, down)\pi(s_1, up) + R_D(s, left, down)\pi(s_1), left) \geq V_D(s_1) \\ \pi_D(s_1, up) + \pi_D(s_1, left) = 1 \end{aligned} \tag{4.12}$$

Similarly, for the invader we get

$$\begin{aligned} R_I(s, right, up)\pi_I(s_1, right) + R_I(s, down, up)\pi_I(s_1), down) \geq V_I(s_1) \\ R_I(s, right, left)\pi_I(s_1, right) + R_I(s, down, left)\pi_I(s_1), down) \geq V_I(s_1) \\ \pi_I(s_1, right) + \pi_I(s_1, down) = 1 \end{aligned} \tag{4.13}$$

The linear programming algorithm is run separately for both the invader and the defender. The algorithm (such as the simplex method) will determine the values of $\pi_D(s_1, up)$, $\pi_D(S_1, left)$, and $V_D(s_1)$ at each iteration. The agents do not know a priori the rewards. The best estimate of the rewards is the state-action function $Q(s, a)$. Therefore, the minimax Q-learning algorithm updates both the expected rewards and the policy, $\pi(s, a)$, simultaneously. To use MATLAB to compute the linear programming solution, we need to get the equations into the correct form. MATLAB structures the linear programming problem as

$$\min_x f^T x \tag{4.14}$$

given the constraints

$$\begin{aligned} A \cdot x \leq b \quad & \text{inequality constraints} \\ A_{eq} \cdot x = b_{eq} \quad & \text{equality constraints} \end{aligned}$$

and

$$\text{lb} \leq x \leq \text{ub lower and upper bounds on x}$$

Notice that MATLAB formulates the problem as a minimization problem. We convert the reward maximization problem in the minimax algorithm into

a minimization problem by multiplying by -1. We substitute the agents' best estimate of its reward for $R_D(s_1, a_i, a_{-i})$ and $R_I(s_1, a_i, a_{-i})$ with $Q_D(s_1, a_i, a_{-i})$ and $Q_I(s_1, a_i, a_{-i})$ and rewrite the minimization problem for the defender and the invader. For the case of the defender, we write the minization problem as

$$\min_{x_d} f_d^T x_d$$

where $f_D^T = [0, 0, -1]$ and $x_d^T = [\pi_D(s_1, up), \pi_D(s_1, left), V_D(s_1)]$, subject to the constraint

$$-Q_D(s_1, up, right)\pi_D(s_1, up) - Q_D(s_1, left, right)\pi_D(s_1, left) + V_D(s_1) \leq 0$$
$$-Q_D(s_1, up, down)\pi_D(s_1, up) - Q_D(s_1, left, down)\pi_D(s_1, left) + V_D(s_1) \leq 0$$

and

$$\pi_D(s_1, up) + \pi_D(s_1, left) = 1$$

Then the matrix A becomes

$$A = \begin{bmatrix} -Q_D(s_1, up, right) & -Q_D(s_1, left, right) & 1 \\ -Q_D(s_1, up, down) & -Q_D(s_1, left, down) & 1 \end{bmatrix}$$

The matrix b becomes $b = [0 \quad 0]^T$. The equivalency condition, which states that the action probabilities sum to 1, is given by

$$A_{eq} = [\pi_D(s_1, up) \qquad \pi_D(s_1, left)]$$

and $b_{eq} = 1$.

For the sake of completeness, we also write out the matrix equations for the invader as

$$\min_{x_I} f_I^T x_I$$

where $f_I^T = [0, 0, -1]$ and $x_I^T = [\pi_I(s_1, right), \pi_I(s_1, down), V_I(s_1)]$, subject to the constraint

$$-Q_I(s_1, right, up)\pi_I(s_1, right) - Q_I(s_1, down, up)\pi_I(s_1, down) + V_I(s_1) \leq 0$$
$$-Q_I(s_1, right, left)\pi_D(s_1, right) - Q_I(s_1, down, left)\pi_I(s_1, down) + V_I(s_1) \leq 0$$

and

$$\pi_I(s_1, right) + \pi_I(s_1, down) = 1$$

Then the matrix A becomes

$$A = \begin{bmatrix} -Q_I(s_1, up, right) & -Q_I(s_1, left, right) & 1 \\ -Q_I(s_1, up, down) & -Q_I(s_1, left, down) & 1 \end{bmatrix}$$

The matrix b becomes $b = [0 \quad 0]^T$. The equivalency condition, which states that the action probabilities sum to 1, is given by

$$A_{eq} = [\pi_I(s_1, right) \quad \pi_I(s_1, down)]$$

and $b_{eq} = 1$.

We first apply the minimax-Q algorithm to the game. To better examine the performance of the minimax-Q algorithm, we use the same parameter settings as in Reference 1. We use the ε-greedy policy as the exploration-exploitation strategy. The ε-greedy policy is defined such that the player chooses an action randomly from the player's action set with a probability ε and a greedy action with a probability $1 - \varepsilon$. The greedy parameter ε is given as 0.2. The learning rate α is chosen such that the value of the learning rate will decay to 0.01 after one million iterations. The discount factor γ is set to 0.9. The number of iterations represents the number of times the step 2 is repeated in Algorithm 4.1. After learning, we plot the players' learned strategies in Fig. 4-3.

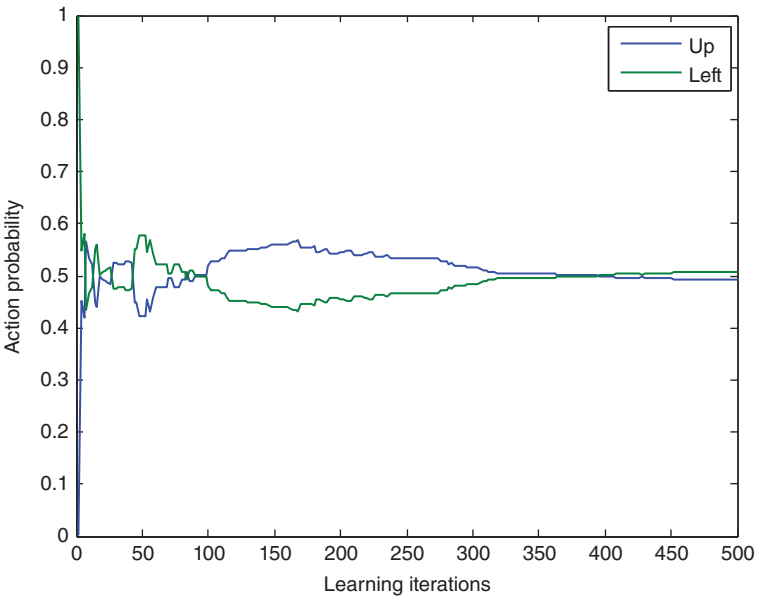


Fig. 4-3. Minimax Q-learning for the defender/invader game. Action probability for the defender.

4.4 Nash Q-Learning

The Nash Q-learning algorithm, first introduced in Reference 7, extends the minimax-Q algorithm [1] from zero-sum stochastic to general-sum stochastic games. In the Nash Q-learning algorithm, the Nash Q-values need to be calculated at each state in order to update the action-value functions and find the equilibrium strategies. Although Nash Q-learning is applied to general-sum stochastic games, the conditions for the convergence to a Nash equilibrium do not cover a correspondingly general class of environments [8]. The corresponding class of environments are actually limited to cases where the game being learned only has coordination or adversarial equilibrium [8, 10]. The Nash Q-Learning algorithm is shown in Algorithm 4.2.

Algorithm 4.2 Nash Q-learning algorithm

1: Initialize $Q_i(s, a_1, \ldots, a_n) = 0$, $\forall a_i \in A_i$, $i = 1, \ldots, n$
2: **for** Each iteration **do**
3: Player i takes an action a_i from current state s based on an exploration-exploitation strategy
4: At the subsequent state s', player i observes the rewards received from all the players $r_1, \ldots, r_n$, and all the players' actions taken at the previous state s.
5: Update $Q_i(s, a_1, \ldots, a_n)$:

$$Q_i(s, a_1, \ldots, a_n) \leftarrow (1 - \alpha) Q_i(s, a_1, \ldots, a_n) + \alpha \left[r_i + \gamma \text{Nash} Q_i(s') \right] \quad (4.15)$$

where α is the learning rate and γ is the discount factor
6: Update $\text{Nash}Q_i(s)$ and $\pi_i(s)$ using quadratic programming
7: **end for**

To guarantee the convergence to Nash equilibria in general-sum stochastic games, the Nash Q-learning algorithm needs to hold the following condition during learning: every stage game (or state-specific matrix game) has a global optimal point or a saddle point for all time steps and all the states [8]. Since the above strict condition is defined in terms of the stage games as perceived during learning, it cannot be evaluated in terms of the actual game being learned [8]. Similar to the minimax Q-learning, the Nash Q-learning algorithm needs to solve the appropriate search algorithm (such as the Lemke–Howson algorithm) at each iteration in order to obtain the Nash Q-values, which leads to a slow learning process.

The Nash Q-learning algorithm extends the Q-learning algorithm to the non-cooperative multiagent context. The learning agent maintains a Q-function over joint actions and performs updates by assuming the existence of Nash equilibrium. This algorithm provably converges given certain constraints on stage games. Hu and Wellman [8] find that agents are more likely to reach a joint optimal path with Nash-Q than with single-agent Q-learning. Although the single-agent properties of Q-learning do not transfer to the multiagent case, the ease of application does.

Direct implementation of Q-learning to the multiagent context is affected by three issues. The environment is no longer stationary, the familiar guarantees are no longer true, and the nonstationary environment is populated by other agents which we assume to be rational. Explicitly accounting for the fact that the agent is operating with other "rational" agents improves the learning process [1, 8, 11]. The reward depends on the joint actions of the other learners. The default or baseline solution concept for general-sum games is the Nash equilibrium. In the framework of general-sum stochastic games, we define optimal Q-values as those values received in Nash equilibrium and refer to them as *Nash Q-values*. The goal is to find the Nash Q-values through repeated play. Agents have to learn the behavior of the other agents and then determine their own best response. In Reference 8, two grid games are proposed. In grid game 1, there are three equally valued global optimal points. Grid game 2 does not have a saddle point or a global optimal point but three sets of other Nash equilibria and in these cases the algorithm does not always converge.

As we recall from the single-agent Q-learning, the agent's objective is to find a strategy (policy) π so as to maximize the sum of discounted future rewards, given by

$$V(s,\pi) = \sum_{t=0}^{\infty} \beta' E(r_t|\pi, s_0 = s) \tag{4.16}$$

The search algorithm attempts to find the stationary point of

$$V(s,\pi^*) = \max_a \left\{ r(s,a) + \beta \sum_{s'} p(s'|s,a,) v(s',\pi^*) \right\} \tag{4.17}$$

The solution to the above Bellman equation is guaranteed to be optimal. Define the optimal Q-function as

$$Q^*(s,a) = r(s,a) + \beta \sum_{s'} p(s'|s,a) v(s', \pi^*) \tag{4.18}$$

The term $Q^*(s,a)$ is the total discounted reward for taking action a, in state s and then following the optimal policy thereafter. By definition, we have

$$V(s,\pi^*) = \max_a Q^*(s,a) \tag{4.19}$$

If we know $Q^*(s,a)$, then the problem can be solved simply by taking the action that maximizes $\max_a Q^*(s,a)$. The Q-learning algorithm is a stochastic approximation algorithm, given by

$$Q_{t+1}(s_t,a_t) = (1-\alpha_t)Q_t(s_t,a_t) + \alpha_t(r_t + \beta \max_a Q_t(s_{t+1},a)) \tag{4.20}$$

Hu and Wellman [8] limit their work to stationary strategies and state the following theorem:

Theorem 4.1 Theorem 4 (Fink, 1964): Every n-player discounted stochastic game possesses at least one Nash equilibrium point in stationary strategies.

The Q-learning algorithm is extended to multiagent games on the basis of the framework of stochastic games. The Nash Q-value is the expected sum of future discounted rewards when all agents follow Nash equilibrium strategies from the next step onwards. This is different from the single-agent case where the future rewards are based only on the agent's own optimal strategy.

Definition 4.1 (Hu and Wellman [8]) *Agent i's Nash Q-function is defined over* $(s, a^1, \ldots, a^n)$ *as the sum of agent i's current reward plus its future rewards when all agents follow a joint Nash equilibrium strategy. That is,*

$$Q^i(s,a^1,\ldots,a^n) = r^i(s,a^1,\ldots,a^n) + \beta \sum_{s' \in S} p(s'|s,a^1,\ldots,a^n) v^i(s',\pi^1,\ldots,\pi^n) \tag{4.21}$$

where $(\pi^1, \ldots, \pi^n)$ *is the joint Nash strategy,* $r^i(s,a^1,\ldots,a^n)$ *is agent i's reward in state s and under joint action* $(a^1,\ldots,a^n)$*, and* $V^i(s',\pi^1,\ldots,\pi^n)$ *is agent i's total discounted reward from* s' *given the other agents follow their Nash equilibrium strategy.*

In the Nash Q-learning algorithm, the multiagent Q-learning algorithm updates with future Nash equilibrium payoffs, whereas the single-agent

Q-learning updates are based on the maximum A-values in the agent's own payoff table. To know the Nash equilibrium payoffs, the agent must also know the reward received by the other agents. The agent must be able to observe these rewards in some way. We define the difference between the Nash equilibria for a stage game and for a stochastic game.

Definition 4.2 (Hu and Wellman [8]) *An n-player stage game is defined as* $(M^1, \ldots, M^n)$*, where for* $k = 1, \ldots, n$ *the term* M^k *is agent k's payoff function over the space of joint actions* $M^k = \{r^k(a^1, \ldots, a^n) | a^1 \in A^1, \ldots, a^n \in A^n\}$ *and* r^k *is agent k's payoff.*

The Nash Q-learning algorithm executes as follows: initialize the Q-table as $Q_0^i(s, a^1, \ldots, a^n) = 0, \forall s \in S, a^1 \in A^1, \ldots, a^n \in A^n$. At each time t, agent i observes the current state and takes an action. Then the agent observes its reward, the rewards received by the other agents, and the new state. The agent then computes a Nash equilibrium at the new state for the stage or matrix game, $Q_t^1(s'), \ldots, Q_t^n(s')$, and updates its Q-values as

$$Q_{t+1}^i(s, a^1, \ldots, a^n) = (1 - \alpha_t) Q_t^i(s, a^1, \ldots, a^n) + \alpha_t [r_t^i + \beta Nash Q_t^i(s')] \quad (4.22)$$

The notation for all players or agents playing their Nash equilibrium strategy is $\pi_1(s'), \ldots, \pi_n(s')$, and the term $Q_t^i(s', \pi_1(s'), \ldots, \pi_n(s'))$ is the Nash equilibrium payoff for state s'. Therefore, for agents to determine the Nash equilibrium, they each have to know the other agents' Q-values. Thus agent i must learn the Q-values for the other agents. For example, agent i may initialize the Q-values for the other agents as $Q_0^j(s, a^1, \ldots, a^n) = 0$ for all j and all $s, a^1, \ldots, a^n$. Agent i observes the other agents' rewards and actions and then updates the other agents' Q-values. The update rule is the same as above, that is

$$Q_{t+1}^j(s, a^1, \ldots, a^n) = (1 - \alpha_t) Q_t^j(s, a^1, \ldots, a^n) + \alpha_t [r_t^i + \beta Nash Q_t^j(s')] \quad (4.23)$$

Therefore, only the entry in the Q-table associated with the current state and action is updated. Although the Nash Q-learning algorithm can be written easily as in Algorithm 4.2, the algorithm is extremely complex. The user has to maintain multiple Q-tables and then compute a Nash equilibrium that all agents agree upon. One of the difficulties in implementing the Nash Q-learning algorithm is the computation of the Nash equilibrium. Hu and Wellman [8] use the Lemke–Howson algorithm [12]. We will present a detailed description of this algorithm in Section 4.6. We will present the examples of the following two grid games that are also proposed in Reference 8 and used to evaluate several other algorithms as well. The games are illustrated in Figs. 4-4 and 4-5.

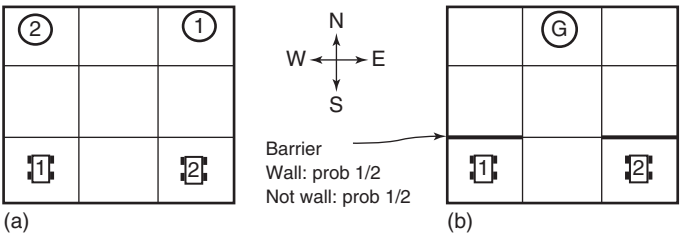


Fig. 4-4. Two stochastic games [7]. (a) Grid game 1. (b) Grid game 2.

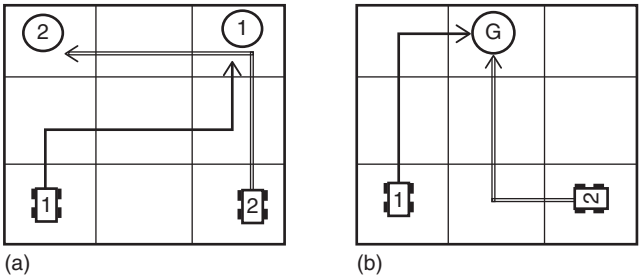


Fig. 4-5. (a) Nash equilibrium of grid game 1. (b) Nash equilibrium of grid game 2. Reproduced from [8] with permission from MIT press.

The agents can move Up, Down Right and Left. If two agents move to the same cell, then they bounce back unless it is the goal cell. The game ends when an agent reaches the goal state. If both agents reach the goal state at the same time, then both get rewarded with positive payoff. Agents do not initially know their goals or their payoffs. The agents choose actions simultaneously. The grid cells are defined as starting with cell state 0 at the bottom left corner and incrementing from left to right until the top right hand corner which is cell state 8. For example the bottom right hand corner is cell state 2. The action space is $A^i = \{Left, Right, Down, Up\}$ and a given state is given by $s = (l^1, l^2)$ where l^1 and l^2 represent the locations of the first and the second agents. If an agent reaches its goal, then it gets a reward of 100. If the agents collide, both agents bounce back to their original position and get a negative reward of -1, and if the agent moves to an empty cell, it receives 0 reward.

In grid game 1, the state transitions are deterministic, and in grid game 2 the transition through the barrier is 50%. For example, if agent 1 chooses action Up, and agent 2 chooses action Left, from state (0, 2), we get the state transition probabilities as

$$P((0,1)|(0,2), Up, Left) = 0.5$$

and

$$P((3,1)|(0,2), Up, Left) = 0.5$$

Similarly, we have $P((1,2)|(0,2), Right, Up) = 0.5$ and $P((1,5)|(0,2), Right, Up) = 0.5$. The Nash Q-learning algorithm assumes that the strategies are stationary. A stationary strategy assigns probability distributions over an agent's actions based on the current state, regardless of the history. This means that, if the agents are in the same state, then the action strategy would be the same for all the agents.

Assuming we have pure strategies, such as in game 1, and the strategy is based only on the location of the agents, then the strategies represent a path. The notation (l^1, any) refers to agent 1 being in state l^1 and agent 2 being in *any* state. Then we get a Nash equilibrium strategy as in Table 4.4. One of the Nash equilibrium strategies is depicted in Fig. 4-5. The value of the game for agent 1 is defined so that its accumulated reward when both agents follow their Nash equilibrium is given as

$$v^1(s_0) = \sum_t \beta^t E(r_t|\pi^1, \pi^2, s_0) \tag{4.24}$$

In grid game 1 and initial state $s_0 = (0,2)$, this becomes, given $\beta = 0.99$

$$v^1(s_0) = 0 + 0.99 \times 0 + 0.99 * 2 \times 0 + 0.99^3 \times 100 = 97 \tag{4.25}$$

Based on the value for each state, we can derive the Nash Q-values for agent 1 in state s_0 as

$$Q^1(s_0, a^1, a^2) = r^1(s_0, a^1, a^2) + \beta \sum_{s'} p(s'|s_0, a^1, a^2) v^1(s') \tag{4.26}$$

We will evaluate the Q-value for different actions for agent 1 in state (0, 2). Let us start with the action (Right, Left). If we take the action (Right, Left), then the two agents will bump into each other. This will give a penalty of $r^1((0,2), Right, Left) = -1$. Therefore, the Q-value for taking the action (Right,

Table 4.4 States and strategies.

State	$\pi^1(s)$
(0, 2)	Up
(3, 5)	Right
(4, 8)	Right
(5, any)	Up

Left) in state (0, 2) and then following the optimal path afterwards, from state (0, 2) (because the agents bounce back to state (0, 2)) is

$$Q^1(s_0, Right, Left) = -1 + \beta v^1((0, 2)) \tag{4.27}$$

We have already computed that $v^1((0, 2)) = 97$, and therefore

$$Q^1(s_0, Right, Left) = -1 + 0.99 \times 97 = 95.1 \tag{4.28}$$

But, for the Q-value for both agents going Up, we get

$$Q^1(s_0, Up, Up) = 0 + 0.99 v^1(3, 5) = 97 \tag{4.29}$$

The Nash Q-values for the agents when in state $(0, 2)$ are given in Table 4.5. For grid game 2, we can derive the Nash Q-values. In this case, it is more complicated because we do not have deterministic state transitions. Let us start the agents in state (0, 1) as shown in Fig. 4-6. Then the optimal path for this position is one in which agent 2 takes two steps Up and gets the reward. Agent 1 cannot get to the goal before agent 2. Therefore, the value for agent 1 is

$$v^1(0, 1) = 0 + 0.99 \times 0 + 0.99^2 \times 0 = 0 \tag{4.30}$$

Table 4.5 Grid game 1: Nash Q-values in state (0, 2).

Action	Left	Up
Right	95.1, 95.1	97, 97
Up	97, 97	97, 97

Fig. 4-6. Grid game with barriers, start position (0,1).

However, if we start in state (1, 2), then agent 1 takes two steps Up and wins, and the value for agent 1 is

$$v^1(1,2) = 0 + 0.99 \times 100 = 99 \tag{4.31}$$

However, if we start in state (0, 2), then we can only compute the value in expectation because if the agents choose the action Up, there is only 50% probability that they would go Up and a 50% probability that they stay in the same spot. For starting in state (0, 2), we get the Q-value as

$$Q^1((0,2), Right, Left) = -1 + 0.99v^1((0,2)) \tag{4.32}$$

The next action is $Q^1((0,2), Right, Up)$. In this case, there is only a 50% chance that the agent goes Up. So we write out the Q^1-value as

$$Q^1(s_0, Right, Up) = 0 + 0.99\left(\frac{1}{2}v^1(1,2) + \frac{1}{2}v^1(1,5)\right) \tag{4.33}$$

If the agents take actions (Right, Up), then we may end up in the position (1, 2) or (1, 5). Now, recall that if the agents are in state (1, 2), then the optimal solution from there is to take two steps Up. Similarly, the agent in state (1, 5) has the optimal solution of two steps, Up and Left. Recall, the values of $v^1(1,2) = v^1(1,5) = 0 + 0.99 \times 100 = 99$. Then we can compute

$$Q^1((0,2), Right, Up) = 0 + 0.99\left(\frac{1}{2}(0.99) + \frac{1}{2}(0.99)\right) = 0.98 \tag{4.34}$$

Now let us take the case of $Q^1((0,2), Up, Left)$. In this case, there is only 50% chance that agent 1 moves Up and 50% chance that the agent stays in the same position. Then we can compute

$$Q^1((0,2), Up, Left) = 0 + 0.99\left(\frac{1}{2}v^1(0,1) + \frac{1}{2}v^1(3,1)\right) \tag{4.35}$$

We already know that the value of $v^1 = 0$; this is because agent 2 will take two steps Up and win. If agent 1 gets through the barrier, then it is in position (3, 1) and agent 1 then takes two steps, Up and Left, and wins at the same time as agent 2 takes two steps, Up and Up, and wins. Therefore, we get $v^1(1,3) = 0 + 0.99 \times 100 = 0.99$, and we can compute the optimal Q-value as

$$Q^1((0,2), Up, Left) = 0 + 0.99\left(\frac{1}{2} \times 0 + \frac{1}{2} \times 99\right) = 49 \tag{4.36}$$

Finally, we compute the value for $Q^1((0,2)\,Up, Up)$. In this case, there are four possible outcomes for the next state. Each agent has a 50% probability of moving Up, therefore there is 25% probability for both agents to move Up at the same time. The probability of achieving state (3, 2) is 25%, state (3, 5) is 25%, state (0, 2) is 25%, and state (0, 5) is 25%. So the Q-value for $Q^1((0,2), Up, Up)$ is

$$Q^1((0,2), Up, Up) = 0 + 0.99\left(\frac{1}{4}v^1(3,2) + \frac{1}{4}v^1(3,5) + \frac{1}{4}v^1(0.2) + \frac{1}{4}v^1(0,5)\right) \tag{4.37}$$

We know the value of $v^1(3,2) = 99$, $v^1(3,5) = 99$, and $v^1(0,5) = 0$. Then,

$$Q^1((0,2), Up, Up) = 0 + 0.99\left(\frac{1}{4}v^1(0,2) + \frac{1}{4}\times 99 + \frac{1}{4}\times 99 + \frac{1}{4}\times 0\right)$$

$$= 0.99\times\frac{1}{4}v^1(0,2) + 24.5 + 24.5$$

$$= 0.99\times\frac{1}{4}v^1(0,2) + 49$$

Now let us define $R_1 = v^1(0,2)$ to be agent 1's optimal value by following a Nash equilibrium strategy starting from state (0, 2). The obvious Nash equilibrium for agent 1 of game 2 from the initial state (0, 2) is (Right, Up). Given that (Right, Up) is the Nash equilibrium then, $v^1(0,2) = 0.98$. Then we can derive the other values in the Q-table. On the other hand, the obvious Nash equilibrium for the second agent is (Up, Left), but for agent 1 $Q^1((0,2), Up, Left) = 49$. Furthermore, there is also a mixed strategy of $(\{P(Right) = 0.97, P(Up) = 0.03\}, \{P(Left) = 0.97, and\ P(Up) = 0.03\})$. There are three sets of possible Nash equilibria for grid game 2.

4.4.1 The Learning Process

Let us say that learning of agent 1 begins by initializing the Q-table as $Q^1(s, a^1, a^2) = 0$ for all s, a^1, and a^2. These are agent i's internal beliefs, and have nothing to do with the other agent. We start the game from the initial state (0, 2). The agents then move simultaneously and observe the action taken by the agents and the rewards that the agents receive. The agents then update their Q-tables according to the following:

$$Q^j_{i+1}(s, a^1, a^2) = (1-\alpha_t)Q^j_i(s, a^1, a^2) + \alpha_t[r^j_t + \gamma NashQ^j_t(s)] \tag{4.38}$$

The process is repeated in the next state until the goal state is reached. Then a new game starts and each agent is randomly assigned a new starting position

except for the goal state. The training stops after 5000 episodes. Each episode takes approximately eight steps. Therefore, one experiment takes 40,000 steps. Furthermore, the learning rate is given by

$$\alpha_t(s, a^1, a^2) = \frac{1}{\eta_t(s, a^1, a^2)} \tag{4.39}$$

where $\eta_t(s, a^1, a^2)$ is the number of times the states and actions (s, a^1, a^2) have been visited.

If we look at the algorithm, we see that we have to compute the Nash equilibrium of the stage game with $(Q^1(s^1)$ *and* $Q^2(s^1))$. There may be a choice between multiple Nash equilibria. Hu and Wellman [8] use the Lemke–Howson [12] algorithm to compute the Nash equilibrium for the two-player game. The Lemke–Howson algorithm resembles the simplex algorithm from linear programming.

4.5 The Simplex Algorithm

The simplex algorithm is a well-known algorithm for solving linear programming problems. These are problems of maximizing a utility function or a cost function subject to a number of linear constraints. The linear programming model is as follows: Maximize

$$V = \sum_{j=1}^{n} c_j x_j \tag{4.40}$$

subject to the set of constraints

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad i = 1, 2, \ldots, m \text{ and } x_j \geq 0 \tag{4.41}$$

The simplex algorithm searches for vertices of a polytope that defines the accessible region of the solution space. One goes from vertex to vertex to find the solution. The algorithm assumes nonnegativity. Inequalities are converted into equalities by adding slack variables. Let us take the following example. This example is taken from *Design and Planning of Engineering Systems* [13] (Fig. 4-7). Maximize

$$V = 13x_1 + 11x_2 \tag{4.42}$$

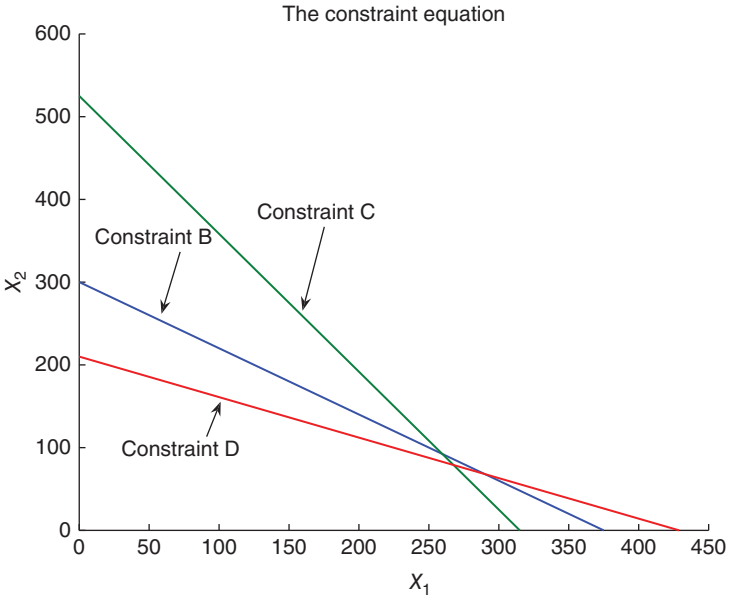


Fig. 4-7. Constraint equations plotted for the simplex method.

subject to the constraints

$$\begin{aligned} 4x_1 + 5x_2 &\leq 1500 \\ 5x_1 + 3x_2 &\leq 1575 \\ x_1 + 2x_2 &\leq 420 \end{aligned} \tag{4.43}$$

and the nonnegativity constraint $x_1 \geq 0$ and $x_2 \geq 0$. For each one of the constraints, we add a slack variable x_3, x_4, or x_5. We therefore convert the inequality constraints into equality constraints, as

$$\begin{aligned} 4x_1 + 5x_2 + x_3 &= 1500 \\ 5x_1 + 3x_2 + x_4 &= 1575 \\ x_1 + 2x_2 + x_5 &= 420 \end{aligned} \tag{4.44}$$

The slack variables represent how much room one has before one hits the constraint. One initializes the system to start with the variables set to $x_1 = 0$ and $x_2 = 0$. This gives the solution to the slack variables as $x^T = [0, 0, 0, 1500, 1575, 420]$. The nonzero variables are known as the *variables in the basis*. In this first step we have selected, the origin as the first

vertex and the value of the reward or value function z is zero, $z = 0$. We now have to move to the next extremal point. This is equivalent to moving one variable out of the basis and moving another variable into the basis. The next step is to determine which variable leaves the basis and which one enters the basis. We put into the basis the variable that causes the greatest growth in the payoff or reward function. Therefore, in this case we put x_1 into the basis because the payoff function will grow by 13 units for every unit of x_1 whereas it only grows by 11 units for every unit of x_2. Then the first constraint is reached when $x_1 = 1500/4 = 375$ and then $x_3 = 0$. The second constraint is reached when $x_1 = 1575/5 = 315$ and then $x_4 = 0$. The final constraint is reached when $x_1 = 420$ and $x_5 = 0$. Therefore, x_1 hits constraint C first when it is equal to 315. Then x_1 enters the basis, and $x_4 = 0$ leaves the basis. We then write the variables in the basis in terms of only the variables not in the basis. This is a form of Gaussian elimination. At the next vertex we get $x_1 = 315$, $x_2 = 0$, and $x_4 = 0$. We then write the system of equations as

$$\begin{aligned} z - 13x_1 - 11x_2 &= 0 && \text{A} \\ 4x_1 + 5x_2 + x_3 &\leq 1500 && \text{B} \\ 5x_1 + 3x_2 + x_4 &\leq 1575 && \text{C} \\ x_1 + 2x_2 + x_5 &\leq 420 && \text{D} \end{aligned} \tag{4.45}$$

Recall that it is the x_4 slack variable that goes to 0. Therefore, we use Eq. C, to solve for x_1. Essentially, using Gaussian elimination we multiply Eq. C by 13/5 and add Eq. C to Eq. A and get

$$z - 16/5x_2 + 13/5x_4 = 4095 \quad \text{A1}$$

Then multiply Eq. C by $-4/5$ and add to Eq. B and we get

$$13/5x_2 + x_3 - 4/5x_4 = 240 \quad \text{B1}$$

Similarly, divide Eq. C by 5 and we get

$$x_1 + 3/5x_2 + 1/5x_4 = 315 \quad \text{C1}$$

And, finally, multiply Eq. C by $-1/5$ and add it to Eq. D and we get

$$7/5x_2 - 1/5x_4 + x_5 = 105 \quad \text{D1}$$

We see from these equations that, if x_2 increases, then so does the payoff function z. This is because the coefficient of x_2 in Eq. A1 is negative. So, x_2 will enter the basis and either x_3 or x_5 will leave the basis. The slack variable x_4 remains zero because we are moving along the edge defined by constraint C. Along this edge, the slack variable is zero.

Therefore, when x_3 and x_4 are zero, x_2 can grow to $5.24/13 = 92.3$; if x_4 and x_5 are zero, then x_2 can grow to $5105/7 = 75$; and if x_1 and x_4 are zero, then x_2 can grow to $5315/3 = 525$. The limiting case is when x_5 is zero and $x_4 = 0$. This represents the intersection of the two constraints C and D. Then we use Gaussian elimination again and write out the equations in terms of x_4 and x_5 only. Then we multiply Eq. D1 by $16/5 \times 5/7$ and add it to Eq. A1 and get

$$z + 15/7x_4 + 16/7x_5 = 4335 \quad \text{A2}$$

We can longer increase z without violating a constraint condition. Similarly, we remove x_2 from Eq. B1 by multiplying Eq. D1 by $-13/5 \times 5/7$ and add it to Eq. B1. This gives

$$x_3 - 3/7x_4 - 13/7x_5 = 45 \quad \text{B2}$$

Similarly, for Eq. C1 we multiply Eq. D1 by $-3/5 \times 5/7$ and add it to Eq. C1 and get

$$x_1 + 10/35x_4 - 3/7x_5 = 270 \quad \text{C2}$$

Finally, we multiply Eq. D1 by $5/7$ and get

$$x_2 - 1/7x_4 + 5/7x_5 = 75 \quad \text{D2}$$

To summarize, we can write the equations as

$$\begin{aligned} z + 15/7x_4 + 16/7x_5 &= 4335 \quad \text{A2} \\ x_3 - 3/7x_4 - 13/7x_5 &= 45 \quad \text{B2} \\ x_1 + 10/35x_4 - 3/7x_5 &= 270 \quad \text{C2} \\ x_2 - 1/7x_4 + 5/7x_5 &= 75 \quad \text{D2} \end{aligned} \tag{4.46}$$

Recall that the slack variables x_4 and x_5 are zero at the intersection of the constraint equations C and D. Then the optimal value for the cost or payoff function is $z = 4335$ and $x_1 = 270$ and $x_2 = 75$ and the slack value of the constraint given by Eq. B is $x_3 = 45$.

4.6 The Lemke–Howson Algorithm

This algorithm is applicable to the two-player game. Player 1 has m actions and player 2 has n actions. We label the actions of player 1 as $M = \{1, \ldots, m\}$ and the actions of player 2 as $N = \{m+1, \ldots, m+n\}$. The payoff for the players is given by the usual payoff matrix. We have two payoff matrices, matrix A and matrix B for player 1 and player 2, respectively. If there is a mixed strategy, given by the action profile (x, y), then the payoff for player 1 is $x^T Ay$ and the payoff for player 2 is $x^T By$. The support of a vector is defined as $supp\{\cdot\}$ and represents the indices of elements of the vector that are nonzero. The term x denotes all possible mixed strategies for agent 1, and y denotes all possible mixed strategies for agent 2. We assume that A and B have all positive elements and that neither A nor B have all zero rows or columns. Let B_j denote the column of B corresponding to action j, and let A^i denote the row of A corresponding to action i. Define the two polytopes

$$P_1 = \{x \in R^m | (\forall i \in M : x_i \geq 0) \qquad (\forall j \in N : x^T B_j \leq 1)\} \tag{4.47}$$

and

$$P_2 = \{y \in R^n | (\forall j \in N : y_i \geq 0) \qquad (\forall i \in M : A^i y \leq 1)\} \tag{4.48}$$

The strategy is given by $nrml(x) := (\sum_i x_i)^{-1} x$. The inequalities that define P_1 have the following meaning:

- if $x \in P_1$ meets $x_i \geq 0$ with equality, for example, $x_i = 0$, then i is not in the support of x;
- if $x \in P_1$ meets $x^T B_j \leq 1$ with equality, then j is the best response to $nrml(x)$.

The solution techniques to find the Nash equilibrium of general-sum games are the *Lemke–Howson algorithm* and its extensions. For completeness, we will present the Lemke–Howson algorithm. Furthermore, these algorithms are based on methods from linear programming such as the simplex method. The Lemke–Howson algorithm is the central algorithm of the Nash Q-learning algorithm even though the seminal paper by Hu and Wellman [8] only cite the algorithm in passing.

The Lemke–Howson algorithm resembles the simplex algorithm. This algorithm is used for two-player bimatrix games. The Nash Q-learning algorithm uses this algorithm on each iteration to solve for the Nash equilibrium of the two-player game. In particular, Hu and Wellman [8] use this algorithm for their

examples. Let player 1 have m actions given by the set $M = \{1, \ldots, m\}$, and player 2 n actions given by the set $N = \{m+1, \ldots, m+n\}$. We represent the payoff matrix for the players as an $m \times n$ matrix. We define the payoff matrix for player 1 with the matrix A and that for player 2 as the matrix B. We think of player 1 as choosing actions that are represented by rows, and player 2 as choosing the columns. We define an m-dimensional row vector x that represents the probabilities of player 1 choosing each of the possible actions. The elements of the row vector x will sum to 1. Similarly, we define an n-dimensional column vector to represent the probabilities of player 2 choosing each action. Therefore, $x \in R^m$ is the mixed strategy for player 1, and $y \in R^n$ is a column vector representing the mixed strategy for player 2. The expected reward for player 1 can then be written as

$$R_1 = x^T A y \tag{4.49}$$

and the expected reward for player 2 can be written as

$$R_2 = x^T B y \tag{4.50}$$

Similar to the simplex method, we define the support of any strategy as those actions that do not have zero entries in x or y. We assume that matrix A has no all-zero columns and that B has no all-zero rows and that the entries of A and B are all nonnegative. Let B_j denote a column of b, and a^i denote a row of A. Then we define the two polytopes as follows:

$$P_1 = \left\{x \in R^m | (\forall i \in M : x_i \geq 0) \quad (\forall j \in N : x^T B_j \leq 1)\right\} \tag{4.51}$$

$$P_2 = \left\{y \in R^n | (\forall j \in N : y_i \geq 0) \quad (\forall i \in M : A^i y \leq 1)\right\} \tag{4.52}$$

The inequalities described in the polytopes defined above have the following meanings: If $x_i = 0$, then x_i is not in the support or basis of x. Recall this language from the description in the simplex method. The second equality constraint is $x^T B_j = 1$. This means that player 2's action j is the best response to strategy x of player 1. If the sum of the elements in column B^j is greater than 1, then the sum of the elements of x will be less than 1, and to get the strategy we will need to normalize x as

$$nrml(x) := \left(\sum_i x_i\right)^{-1} x \tag{4.53}$$

Let us define labels as follows: A strategy $x \in P_1$ has label $k \in M \cup N = \{1, 2, \ldots, m+n\}$ if either $k \in M$ and $x_k = 0$ or $k \in N$ and $x^T B_k = 1$. We then can state the following theorem:

Theorem 4.2 Suppose that $x \in P_1$ and $y \in P_2$ and neither x nor y is the all-zero vector. Then, x and y together have all labels from 1 to k, iff $(nrml(x), nrml(y))$ is a Nash equilibrium.

We give a simple example to illustrate how the Lemke–Howson algorithm is implemented. Let us take the case of a two-player two-action game, where the reward matrices have the following values:

$$A = \begin{bmatrix} 4 & 6 \\ 5 & 3 \end{bmatrix} \quad B = \begin{bmatrix} 3 & 2 \\ 1 & 4 \end{bmatrix} \tag{4.54}$$

The polytopes P_1 and P_2 are defined by the limits of the constraints $A^i y \leq 1$ and $B^{jT} x \leq 1$, and both $x_i \geq 0$ and $y_j \geq 0$ as illustrated in Figs. 4-8 and 4-9. We will convert these inequalities into equality constrains by adding slack variables as we did in the simplex method. Define the slack variable r_i as

$$A^i y + r_i = 1 \tag{4.55}$$

Similarly

$$B^{jT} x + s_j = 1 \tag{4.56}$$

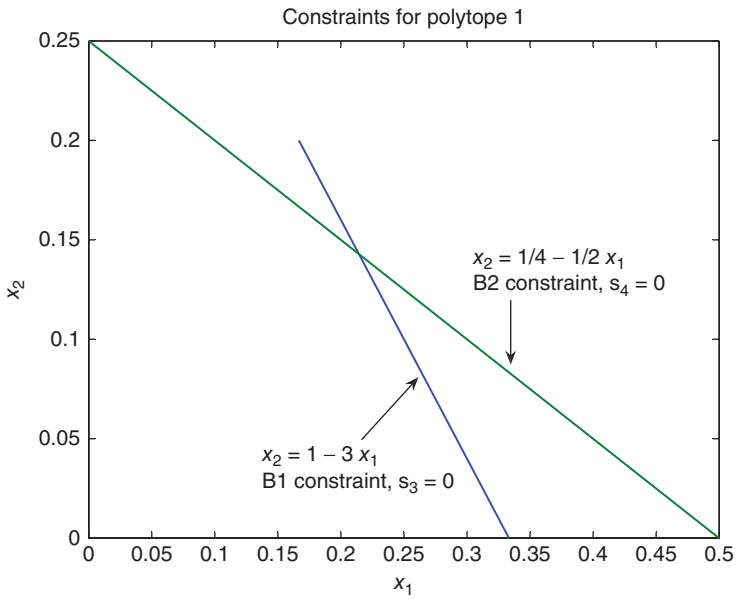


Fig. 4-8. Polytope defined by P_1.

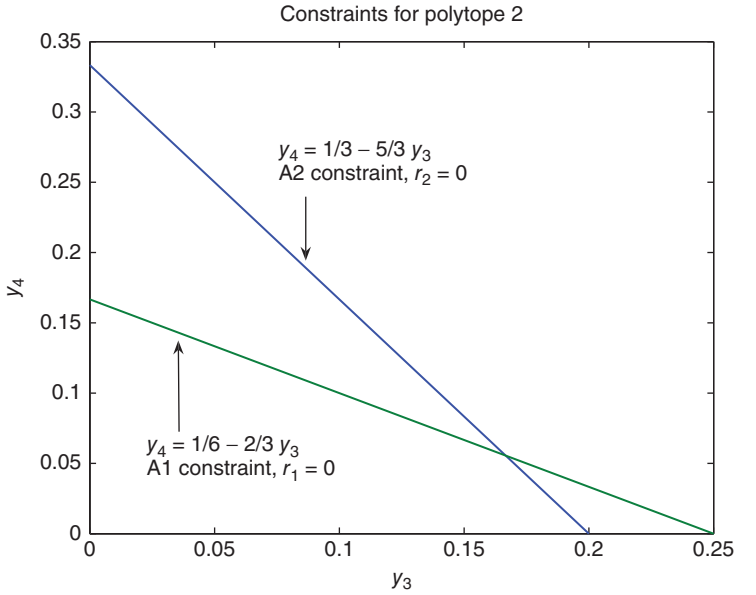


Fig. 4-9. Polytope defined by P_2.

Then, similar to the simplex method, we write $r_i = 1 - A^i y$ as

$$r_1 = 1 - 4y_3 - 6y_4 \quad \text{A1}$$
$$r_2 = 1 - 5y_3 - 3y_4 \quad \text{A2}$$

Similarly, the s constraint $s_j = 1 - B^{jT} x$ becomes

$$s_3 = 1 - 3x_1 - x_2 \quad \text{B1}$$
$$s_4 = 1 - 2x_1 - 4x_2 \quad \text{B2}$$

We can now draw out the constraint conditions as illustrated in Fig. 4-8. Taking the slack variables as being equal to zero, $s_1 = s_2 = 0$. The equality constraints for the (x_1, x_2) polytope become

$$x_2 = 1 - 3x_1$$
$$x_2 = \frac{1}{4} - \frac{1}{2}x_1$$

Similarly, for the (y_3, y_4) polytope we have the constraints

$$y_4 = \frac{1}{6} - \frac{2}{3}y_3$$

$$y_4 = \frac{1}{3} - \frac{5}{3}y_3$$

We will start by arbitrarily allowing x_1 into the basis. Then the constraint B1 limits the growth of x_1 to 1/3, as seen in Figs. 4-8. Along the B1 constraint, the slack variable s_3 is zero. Then, from the complementary condition $s_3 y_3 = 0$, we bring y_3 into the basis. From Eq. B1 we solve for x_1 as

$$3x_1 = 1 - x_2 - s_3$$
$$x_1 = \frac{1}{3} - \frac{1}{3}x_2 - \frac{1}{3}s_3$$

where $x_2 = 0$ and $s_3 = 0$. Substituting for x_1 into constraint Eq. B2, we get

$$s_4 = 1 - 2\left(\frac{1}{3} - \frac{1}{3}x_2 - \frac{1}{3}s_3\right) - 4x_2$$
$$s_4 = 1 - \frac{2}{3} + \frac{2}{3}x_2 + \frac{2}{3}s_3 - 4x_2$$
$$s_4 = \frac{1}{3} - \frac{10}{3}x_2 + \frac{2}{3}s_3 \quad \text{B2}'$$

This is what just happened: we arbitrarily let agent 1 choose action 1 and found that condition B1 was satisfied first when $s_3 = 0$, at $x_1 = \frac{1}{3}$ and $x_2 = 0$. This means that the best response by agent 2 to agent 1 choosing action 1 is for agent 2 to take action 3, which represents the first column of the payoff function given by matrix B. This is obvious because, if agent 1 picks action 1, then agent 2 should pick action 3, which is the first column of the B matrix and then agent 2 receives a reward of 3. If agent 2 chooses action 4, then its reward would be only 2. Given that agent 1 chooses action 1, this means that agent 2 chooses action 3, and this is equivalent to having y_3 enter the basis. The limiting condition on y_3 is given by the constraint A2. We then solve for y_3 because of constraint A2 as

$$y_3 = \frac{1}{5} - \frac{3}{5}y_4 - \frac{1}{5}r_2 \quad \text{where} \quad y_4 = 0 \quad \text{and} \quad r_2 = 0$$

We then compute r_1 as

$$r_1 = 1 - 4\left(\frac{1}{5} - \frac{3}{5}y_4 - \frac{1}{5}r_2\right) - 6y_4$$
$$r_1 = \frac{1}{5} - \frac{18}{5}y_4 + \frac{4}{5}r_2 \quad \text{A2}'$$

Therefore, the best response to agent 2 taking action 3 is for agent 1 to take action 2 because $r_2 = 0$. We then pivot along the $s_3 = 0$ line in the (x_1, x_2)

polytope. We then add x_2 to the basis because we have $x_2 r_2 = 0$, and solve for x_2 from Eq. B2' as

$$\begin{aligned}
\frac{10}{3}x_2 &= \frac{1}{3} + \frac{2}{3}s_3 - s_4 \\
x_2 &= \frac{1}{10} + \frac{1}{5}s_3 - \frac{3}{10}s_4 \quad \text{where} \quad s_3 = s_4 = 0 \\
x_1 &= \frac{1}{3} - \frac{1}{3}\left(\frac{1}{10} + \frac{1}{5}s_3 - \frac{3}{10}s_4\right) - \frac{1}{3}s_3 \\
&= \frac{9}{30} - \frac{16}{15}s_3 + \frac{1}{16}s_4
\end{aligned}$$

Then we choose to take y_4 into the basis because $s_4 = 0$, and it is clear that the best response to agent 1 taking action 2 is for agent 2 to take action 4 and get a reward of 4. We then solve for y_4 from Eq. A2′ and get

$$\begin{aligned}
\frac{18}{5}y_4 &= \frac{1}{5} - r_1 + \frac{4}{5}r_2 \\
y_4 &= \frac{1}{18} - \frac{5}{18}r_1 + \frac{4}{18}r_2 \quad \text{A}''
\end{aligned}$$

Then $r_1 = 0$, and x_1 would enter the basis, but it is already in the basis so the algorithm ends. We now normalize the game strategy. The current solution is given by $x_1 = \frac{9}{30} = \frac{3}{10}$, $x_2 = \frac{1}{10}$, $y_3 = \frac{1}{6}$, and $y_4 = \frac{1}{18}$. We then normalize the strategy as

$$x_{10} = \frac{\frac{3}{10}}{\frac{3}{10} + \frac{1}{10}} = \frac{3}{4} \qquad \text{and} \qquad x_{20} = \frac{\frac{1}{10}}{\frac{3}{10} + \frac{1}{10}} = \frac{1}{4}$$

$$y_{30} = \frac{\frac{3}{18}}{\frac{3}{18} + \frac{1}{18}} = \frac{3}{4} \qquad \text{and} \qquad y_{40} = \frac{\frac{1}{18}}{\frac{3}{18} + \frac{1}{18}} = \frac{1}{4}$$

Now let us check if this works. Given that $x_0 = [\frac{3}{4}, \frac{1}{4}]^T$ and $y_0 = [\frac{3}{4}, \frac{1}{4}]^T$, the payoff to agent 1 is

$$\begin{aligned}
R_1 &= x_0^T A y_0 \\
&= \begin{bmatrix} \frac{3}{4} \\ \frac{1}{4} \end{bmatrix} \begin{bmatrix} 4 & 6 \\ 5 & 3 \end{bmatrix} \begin{bmatrix} \frac{3}{4} \\ \frac{1}{4} \end{bmatrix} = 4.5
\end{aligned}$$

We can also write this in the form before normalization, in which case we get the constraint condition as

$$Ay^* = \begin{bmatrix} 4 & 6 \\ 5 & 3 \end{bmatrix} \begin{bmatrix} \frac{1}{6} \\ \frac{1}{18} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

which satisfies the constraint condition. We can also check the payoff for agent 2.

$$R_2 = x_0^T B y_0$$

$$= \begin{bmatrix} \frac{3}{4} \\ \frac{1}{4} \end{bmatrix} \begin{bmatrix} 3 & 2 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} \frac{3}{4} \\ \frac{1}{4} \end{bmatrix} = 2.5$$

and once again we can verify the constraint conditions as

$$B^T x^* = \begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} \frac{3}{10} \\ \frac{1}{10} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Recall the conditions $x_0^T A y_0 \geq x^T A y_0$ and $x_0^T B y_0 \geq x_0^T B y$. This says that if agent 2 plays its best policy, and if agent 1 plays any policy, it does no better than the optimal policy x_0, and similarly for agent 2. Therefore, we now show that we cannot find any x that will make R_1 greater than the optimal policy selection. We know that the optimal payoff for agent 1 is given by $R_1^* = x_0^T A y_0 = 4.5$. Can we find a policy $x \neq x_0$ such that $R_1 = x^T A y_0 > 4.5$? We have

$$R_1 = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \begin{bmatrix} 4 & 6 \\ 5 & 3 \end{bmatrix} \begin{bmatrix} \frac{3}{4} \\ \frac{1}{4} \end{bmatrix}$$

$$= \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 4.5 \\ 4.5 \end{bmatrix} \quad \forall \quad x_1 \quad \text{and}$$

$$x_2 \text{ s.t.} \quad x_1 + x_2 = 1 \tag{4.57}$$

Similarly, for agent 2 we have $R_2 = x_0^T B y_0 \geq x_0^T B y$, but we know that $x_0^T B = [2.5, 2.5]$ and that $x^T B = [1, 1]$. Therefore, if agent 2 picks its optimal policy y_0, then agent 1 will always get the payoff 4.5, regardless of what strategy it uses; similarly, if agent 1 picks its optimal strategy x_0, then agent 2 gets its optimal payoff off 2.5 regardless of what strategy it picks. We now see the importance of setting the constraint conditions for agent 1 to ensure that $B^T x = 1$ and for agent 2 $Ay = 1$. Then for condition $B^T x = 1$, this means that the payoff to agent 2 is limited by the strategy of agent 1, and similarly the condition $Ay = 1$ means that the payoff to agent 1 is limited by the strategy of agent 2, yielding the maximum payoff for each constrained by the other.

4.7 Nash-Q Implementation

Our goal with the Nash-Q algorithm is to reproduce the results obtained by Hu and Wellman. The results were very conclusive in grid games 1 and 2. We wanted to recreate the situation when both agents were Nash-Q learners. Hu and Wellman's observations showed that both agents learned a Nash equilibrium strategy 100% of the time. The high percentage is a very good indicator that the algorithm works very well in that particular situation.

The Lemke–Howson algorithm was used to find the Nash equilibrium of bimatrix games. We used the original version from Hu and Wellman and adapted it for our environment in MATLAB. The algorithm is designed to find more than one Nash equilibrium when they exist. It is based on the work from Reference 14. The inputs are the two payoff matrices for both players, and the outputs are the Nash equilibria. In our multiple implementations, we kept the parameters from Reference 8. The author had their agents play for 5000 games. The number of games necessary to reach convergence is affected by the variation of the learning rate α. Where α is inversely proportional to the number of times each state tuples (s, a^1, a^2) are visited. We considered 5000 a reasonable number of games because after some testing we calculated that each state would be visited on average 93 times.

We started the implementation with a more general approach to the learning technique than in Reference 8. In our first implementation, our agents were always able to choose any of the four actions. This meant that if the agent chooses to go into a wall, it would be bounced back and would receive either a negative reward or no reward. Also, our Lemke–Howson algorithm would use the Nash Q-values of all four actions for each state. It meant that the algorithm was calculating more than it needed. The results obtained were not conclusive. In our second implementation, we changed the code so that the agents would not be able to choose an action that would lead out of

Table 4.6 Grid game 1. Nash *Q*-values in state (1,3).

	Up	Left
Up	96,95	92,92
Right	85,85	89,85

bounds. The fact that Nash-Q is an offline learner helped us to create the boundaries when choosing randomly the next step. This change positively affected the results, and we got a success rate of about 25%. This was still far from the success rate noted in the literature. In our last implementation, the Lemke–Howson algorithm was used only on the possible actions. This means that the agent would calculate the Nash equilibrium strategy of the next state with only the actions that are allowed in that state. We were able to achieve 100% success in getting a Nash equilibrium strategy for two Nash-Q learners. Table 4.6 shows the value of the Nash Q-values in the starting state (1,3).

We decided to use the same method of Hu and Wellman to confirm their results. This means that the agent will choose a random action in every state. The starting positions of the players change in every game. They start in a random position except for their goal cell. This ensures that each state is visited often enough. According to Reference 8, the learning rate depends on the number of times each state-action tuple has been visited. The value of α is $\alpha(s, a^1, a^2) = \frac{1}{n_t(s,a^1,a^2)}$, where n_t is the number of times the game was in the state-action tuple (s, a^1, a^2) [8]. This allows the learning rate to decay until the state-action tuple is visited often enough. We found that if we remove the states where both players occupied the same cell, the states where one or both of the players are in their goal cell, and the inaccessible actions (the players cannot try to move into the wall), we would get 424 different state-action tuples of the form (s, a^1, a^2) [8]. The learning rate would be negligible after 500 visits with a value of $\alpha = 0.002$.

We also implemented an online version of the algorithm. In this version, the agents start each episode from the state $s(1, 3)$, which is the original starting position. The main difference is that the agent now uses an exploit-explore learning technique where the agent choose a random action with a probability of $1 - \varepsilon$ and the Nash equilibrium strategy with a probability of ε. The value of ε varies during learning as $\varepsilon(s) = \frac{1}{n_t(s)}$, where n_t is the number of times the game

was in the state s [8]. This means that the chance that the agent will choose a random action increases with time. This is not desirable for an online learning agent, because it would cause the average reward to decrease with time. It is necessary for the Nash-Q agent to visit as many different states as possible to ensure convergence to a Nash equilibrium strategy.

Our final results correspond perfectly with those in the literature when looking at two Nash-Q learners playing grid games 1 and 2. We tested each grid game 20 times, and the agents found the Nash equilibrium strategy 100% of the time. We also kept track of the performance of each agent. The performance was calculated by the average reward per step the agent was able to accumulate. The performance of an agent is important because it tells us how well the agent can optimize their policy. To ensure that the values reflect not only the results of one game, we took the average of the value over five games. It also gave us a smoother curve on the graphs which made it easier to read. In Figs. 4-10–4-12, we illustrate the performance of the algorithm in grid game 1 when both agents are Nash-Q learners. It shows the average rewards per step

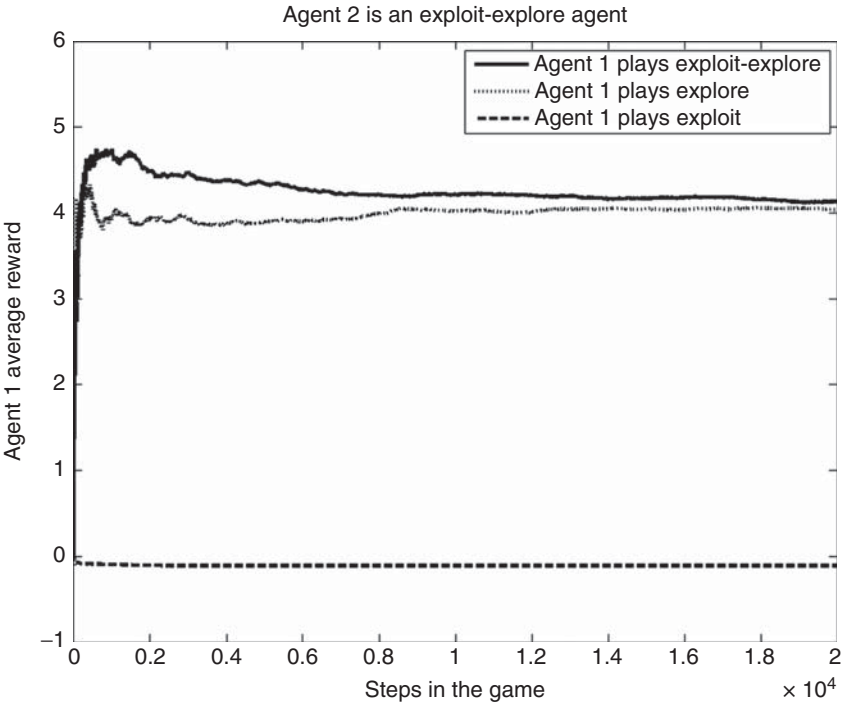


Fig. 4-10. Nash-Q learner with exploit-explore. Reproduced from [15], © P. De Beck-Courcelle.

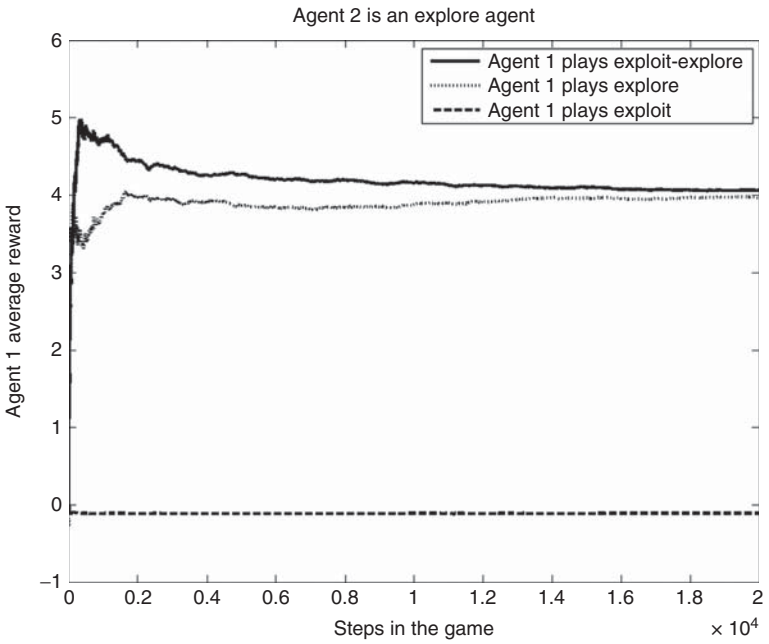


Fig. 4-11. Nash-Q learner with explore only. Reproduced from [15], © P. De Beck-Courcelle.

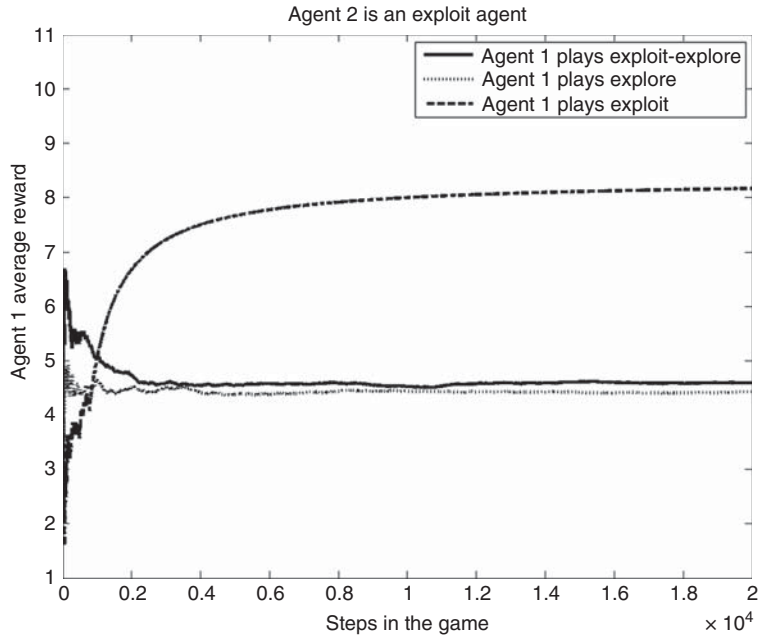


Fig. 4-12. Nash-Q learning with exploit only. Reproduced from [15], © P. De Beck-Courcelle.

when the two agents use the three learning technique: exploit-explore, explore only, and exploit only.

The Nash-Q algorithm converges to a Nash equilibrium strategy only when all the assumptions and conditions are met. The algorithm is also very demanding on computation time. Its needs to keep track of all the players' actions and rewards in a separate Q-table. It also needs to evaluate the Nash equilibrium at every step to update its Q-values. The Lemke–Howson algorithm is equivalent to the simplex algorithm in terms of computation. Both these actions require a large amount of processing power and consume a lot of time.

4.8 Friend-or-Foe Q-Learning

For a two-player zero-sum stochastic game, the minimax-Q algorithm [1] is well suited for the players to learn a Nash equilibrium in the game. For general-sum stochastic games, Littman proposed a friend-or-foe Q-learning (FFQ) algorithm such that a learner is told to treat the other players as either a "friend" or a "foe" [10]. The FFQ algorithm assumes that the players in a general-sum stochastic game can be grouped into two types: player i's friends and player i's foes. Player i's friends are assumed to work together to maximize player i's value, while player i's foes work together to minimize the value [10]. Thus, an n-player general-sum stochastic game can be treated as a two-player zero-sum game with an extended action set [10].

The FFQ algorithm for player i is given in Algorithm 4.3. Note that the FFQ algorithm is different from the minimax-Q algorithm for a two-team zero-sum stochastic game. In a two-team zero-sum stochastic game, a team leader controls the team players' actions and maintains the value of the state for the whole team. The received reward is also the whole team's reward. For the FFQ algorithm, there is no team leader to send commands to control the team players' actions. The FFQ player chooses its own action and maintains its own state-value function and equilibrium strategy. In order to update the action-value function $Q_i(s, a_1, \ldots, a_{n_1}, o_1, \ldots, o_{n_2})$, the FFQ player needs to observe its friends and opponents' actions at each time step.

Littman's FFQ algorithm can guarantee the convergence to a Nash equilibrium if all states and actions are visited infinitely often. The proof of convergence for the FFQ algorithm can be found in Reference 10. Similar to the minimax Q- and Nash Q-learning algorithms, the learning speed is low because of the execution of linear programming at each iteration in Algorithm 4.3.

Algorithm 4.3 Friend-or-foe Q-learning algorithm

Initialize $V_i(s) = 0$ and $Q_i(s, a_1, \ldots, a_{n_1}, o_1, \ldots, o_{n_2}) = 0$ where $(a_1, \ldots, a_{n_1})$ denotes player i and its friends' actions and $(o_1, \ldots, o_{n_2})$ denotes its opponents' actions.

for Each iteration **do**

Player i takes an action a_i from current state s based on an exploration-exploitation strategy.

At the subsequent state s', player i observes the received reward r_i, its friends' and opponents' actions taken at state s.

Update $Q_i(s, a_1, \ldots, a_{n_1}, o_1, \ldots, o_{n_2})$:

$$Q_i(s, a_1, \ldots, a_{n_1}, o_1, \ldots, o_{n_2}) \leftarrow (1-\alpha) Q_i(s, a_1, \ldots, a_{n_1}, o_1, \ldots, o_{n_2}) + \alpha \left[r_i + \gamma V_i(s') \right]$$

where α is the learning rate and γ is the discount factor.

Update $V_i(s)$ using linear programming:

$$V_i(s) = \max_{\pi_1(s,\cdot), \ldots, \pi_{n_1}(s,\cdot)} \min_{o_1, \ldots, o_{n_2} \in O_1 \times \cdots \times O_{n_2}} \sum_{a_1, \ldots, a_{n_1} \in A_1 \times \cdots \times A_{n_1}} Q_i(s, a_1, \ldots, a_{n_1}, o_1, \ldots, o_{n_2}) \pi_1(s, a_1) \cdots \pi_{n_1}(s, a_{n_1}) \quad (4.58)$$

end for

4.9 Infinite Gradient Ascent

It is known in the game theory literature that the strategies need not converge when they are computed by the gradient ascent in two-person iterated games. Singh et al. proved that the average payoffs of two players will always converge to the expected payoffs of some Nash equilibrium even if their strategies do not converge. Let a two-player two-action general-sum game be defined by the following matrices:

$$R = \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

where R is the payoffs of the row player (player 1) and C is the payoffs of the column player (player 2). Assume that player 1 plays action i and player 2 plays action j. Thus, the payoffs that player 1 and player 2 get are R_{ij} and C_{ij}, respectively.

Player 1 and player 2 are said to follow a mixed strategy as they can choose their actions stochastically. Let us assume that $\alpha \in [0,1]$ is the probability of player 1 to choose action 1 and $(1-\alpha)$ is the probability of player 1 to choose action 2. Let us also assume that $\beta \in [0,1]$ is the probability of player 2 to choose action 1 and $(1-\beta)$ is the probability of player 2 to choose action 2. Thus, player 1's expected payoff to the strategy pair (α, β) is

$$V_r(\alpha,\beta) = r_{11}(\alpha\beta) + r_{22}((1-\alpha)(1-\beta)) + r_{12}(\alpha(1-\beta)) + r_{21}((1-\alpha)\beta) \quad (4.59)$$

and player 2's expected payoff to the strategy pair (α, β) is

$$V_c(\alpha,\beta) = c_{11}(\alpha\beta) + c_{22}((1-\alpha)(1-\beta)) + c_{12}(\alpha(1-\beta)) + c_{21}((1-\alpha)\beta) \quad (4.60)$$

The effect of changing a player's strategy is estimated by calculating the partial derivative of its expected payoff with respect to its mixed strategy as follows:

$$\frac{\partial V_r(\alpha,\beta)}{\partial \alpha} = \beta u - (r_{22} - r_{12}) \quad (4.61)$$

$$\frac{\partial V_c(\alpha,\beta)}{\partial \beta} = \alpha \acute{u} - (c_{22} - c_{21}) \quad (4.62)$$

where $u = (r_{11} + r_{22}) - (r_{21} + r_{12})$ and $\acute{u} = (c_{11} + c_{22}) - (c_{21} + c_{12})$.

In the gradient ascent algorithm, at each time step the current strategy of each player is adjusted in the direction of its current gradient with some step size η so as to maximize its expected payoff:

$$\alpha_{k+1} = \alpha_k + \eta \frac{\partial V_r(\alpha_k, \beta_k)}{\partial \alpha} \quad (4.63)$$

$$\beta_{k+1} = \beta_k + \eta \frac{\partial V_c(\alpha_k, \beta_k)}{\partial \beta} \quad (4.64)$$

The step size η is usually in the range $0 < \eta \ll 1$. It is clear that each player assumes that the opponent's strategy is known. Singh et al. proved that the agents, their average payoffs, or both of them will converge to a Nash equilibrium in case of the infinitesimal step size ($lim_{\eta \to 0}$). However, the IGA (infinite gradient ascent) algorithm is not practical because it cannot be applied to a large number of real problems because of the following two reasons:

1. The opponent's strategy is assumed to be totally known by the player;

2. The IGA algorithm is designed for two-player two-action iterated general-sum games; for many-player many-action general-sum games, the extension will not be straightforward.

4.10 Policy Hill Climbing

Policy hill climbing (PHC) is a simple practical algorithm that can play mixed strategies. This algorithm was first proposed by Bowling and Veloso (2002). The PHC does not require much information as neither the player's recently executed actions nor its opponent's current strategy is required to be known. The PHC is a simple modification of the single-agent Q-learning algorithm. A hill climbing is performed by the PHC algorithm in the space of the mixed strategies. The PHC algorithm is composed of two parts. The reinforcement learning is the first part, as the Q-learning algorithm maintains the values of the particular actions in the states. The game-theoretic part is the second part in which the current strategy in each system's state is maintained.

The probability that selects the highest valued actions is increased by a small learning rate $\delta \in (0,1]$ so that the policy is improved. The algorithm is equivalent to Q-learning when $\delta = 1$, as the policy moves to the greedy policy with probability 1 while executing the highest valued action. The PHC algorithm is rational and converges to the optimal solution when a fixed (stationary) strategy is followed by the other players. However, the PHC algorithm may not converge to a stationary policy if the other players are learning although its average reward will converge to the reward of a Nash equilibrium. The PHC algorithm is illustrated in Algorithm 4.4.

4.11 WoLF-PHC Algorithm

Algorithm 4.4 Policy hill climbing algorithm

1: Initialize $Q_i(s,a_i) \leftarrow 0$ and $\pi_i(s,a_i) \leftarrow \frac{1}{|A_i|}$. Choose the learning rate α, δ and the discount factor γ.
2: **for** Each iteration **do**
3: Select action a_c from current state s based on a mixed exploration-exploitation strategy
4: Take action a_c and observe the reward r_i and the subsequent state s'
5: Update $Q_i(s,a_c)$

$$Q_i(s,a_c) = Q_i(s,a_c) + \alpha\left[r_i + \gamma \max_{a_i'} Q(s',a_i') - Q(s,a_c)\right] \tag{4.65}$$

where a_i' is player i's action at the next state s' and a_c is the action player i has taken at state s.

6: Update $\pi_i(s, a_i)$

$$\pi_i(s, a_i) = \pi_i(s, a_i) + \Delta_{sa_i} \quad (\forall a_i \in A_i) \tag{4.66}$$

where

$$\Delta_{sa_i} = \begin{cases} -\delta_{sa_i} & \text{if } a_c \neq \arg\max_{a_i \in A_i} Q_i(s, a_i) \\ \sum_{a_j \neq a_i} \delta_{sa_j} & \text{otherwise} \end{cases} \tag{4.67}$$

$$\delta_{sa_i} = \min\left(\pi_i(s, a_i), \frac{\delta}{|A_i| - 1}\right) \tag{4.68}$$

7: **end for**

The WoLF-PHC (win-or-learn-fast policy hill climbing) algorithm is an extension of the PHC algorithm [2]. This algorithm uses the mechanism of WoLF so that the PHC algorithm converges to a Nash equilibrium in self-play. The algorithm has two different learning rates: δ_w when the algorithm is winning, and δ_l when it is losing. The difference between the average strategy and the current strategy is used as a criterion to decide when the algorithm wins or loses. The learning rate δ_l is larger than the learning rate δ_w. As such, when an agent is losing, it learns faster than when it is winning. This causes the agent to adapt quickly to the changes in the strategies of the other agents when it is doing more poorly than expected, and learns cautiously when it is doing better than expected. This also gives the other agents the time to adapt to the agent's strategy changes. The WoLF-PHC algorithm exhibits the property of convergence as it makes the agent converge to one of its Nash equilibria. This algorithm is also a rational learning algorithm as it makes the agent converge to its optimal strategy when its opponent plays a stationary strategy. These properties permit the WoLF-PHC algorithm to be widely applied to a variety of stochastic games [2, 16–18]. The recursive Q-learning of a learning agent j is given as

$$Q^j_{t+1}(s, a) = (1 - \alpha) Q^j_t(s, a) + \alpha(r^j + \gamma \max_{a'} Q^j_t(s', a')) \tag{4.69}$$

Algorithm 4.5 describes the WoLF-PHC algorithm for a learning agent j, and the algorithm updates the strategy of agent j by the following equation:

$$\pi^j_{t+1}(s, a) = \pi^j_t(s, a) + \Delta_{sa} \tag{4.70}$$

where

$$\Delta_{sa} = \begin{cases} -\delta_{sa} & \text{if } a \neq \arg\max_{a'} Q_t^j(s,a') \\ \sum_{a' \neq a} \delta_{sa'} & \text{otherwise} \end{cases}$$

$$\delta_{sa} = \min\left(\pi_t^j(s,a), \frac{\delta}{|A_j|-1}\right)$$

$$\delta = \begin{cases} \delta_w & \text{if } \sum_{a'} \pi_t(s,a') Q_{t+1}^j(s,a') > \sum_{a'} \bar{\pi}_{t+1}(s,a') Q_{t+1}^j(s,a') \\ \delta_l & \text{otherwise} \end{cases}$$

$$\bar{\pi}_{t+1}^j(s,a') = \bar{\pi}_t^j(s,a') + \frac{1}{C_{t+1}(s)}(\pi_t^j(s,a') - \bar{\pi}_t^j(s,a')) \;\; \forall a' \in A_j$$

$$C_{t+1}(s) = C_t(s) + 1.$$

The WoLF-PHC algorithm for player i is provided in Algorithm 4.5.

Algorithm 4.5 WoLF-PHC learning algorithm

Initialize $Q_i(s,a_i) \leftarrow 0$, $\pi_i(s,a_i) \leftarrow \frac{1}{|A_i|}$ and $C(s) \leftarrow 0$.Choose the learning rate α, δ and the discount factor γ

for Each iteration **do**

Select action a_c from current state s based on a mixed exploration-exploitation strategy

Take action a_c and observe the reward r_i and the subsequent state s'

Update $Q_i(s,a_c)$

$$Q_i(s,a_c) = Q_i(s,a_c) + \alpha\left[r_i + \gamma \max_{a_i'} Q(s',a_i') - Q(s,a_c)\right] \tag{4.71}$$

where a_i' is player i's action at the next state s' and a_c is the action player i has taken at state s.

Update the estimate of average strategy $\bar{\pi}_i$

$$C(s) = C(s) + 1 \tag{4.72}$$

$$\bar{\pi}_i(s,a_i) = \bar{\pi}_i(s,a_i) + \frac{1}{C(s)}\left(\pi_i(s,a_i) - \bar{\pi}_i(s,a_i)\right) \quad (\forall a_i \in A_i) \tag{4.73}$$

where $C(s)$ denotes how many times the state s has been visited.

Update $\pi_i(s,a_i)$

$$\pi_i(s,a_i) = \pi_i(s,a_i) + \Delta_{sa_i} \quad (\forall a_i \in A_i) \tag{4.74}$$

where

$$\Delta_{sa_i} = \begin{cases} -\delta_{sa_i} & \text{if } a_c \neq \arg\max_{a_i \in A_i} Q_i(s, a_i) \\ \sum_{a_j \neq a_i} \delta_{sa_j} & \text{otherwise} \end{cases} \tag{4.75}$$

$$\delta_{sa_i} = \min\left(\pi_i(s, a_i), \frac{\delta}{|A_i| - 1}\right) \tag{4.76}$$

$$\delta = \begin{cases} \delta_w & \text{if } \sum_{a_i \in A_i} \pi_i(s, a_i) Q_i(s, a_i) > \sum_{a_i \in A_i} \bar{\pi}_i(s, a_i) Q_i(s, a_i) \\ \delta_l & \text{otherwise} \end{cases}$$

end for

Different from the previously mentioned learning algorithms, the WoLF-PHC algorithm does not need to observe the other players' strategies and actions. Therefore, compared to the other three learning algorithms, the WoLF-PHC algorithm needs less information from the environment. Since the WoLF-PHC algorithm is based on the PHC method, neither linear programming nor quadratic programming is required in this algorithm. Since the WoLF-PHC algorithm is a practical algorithm, there was no proof of convergence provided in Reference 2. Instead, simulation results in Reference 2 illustrated the convergence of players' strategies by carefully choosing the learning rate according to different examples in matrix games and stochastic games.

4.12 Guarding a Territory Problem in a Grid World

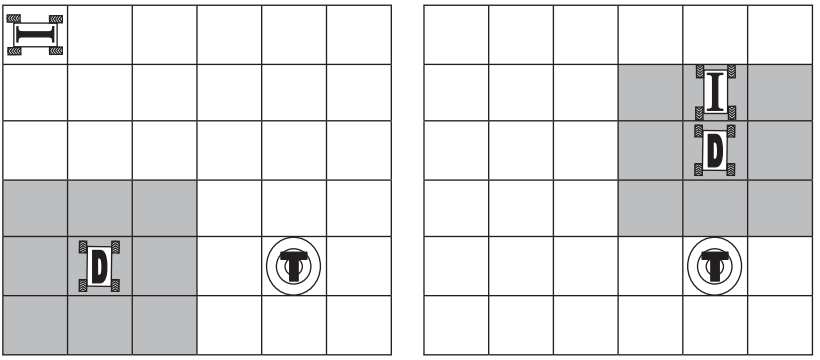
The game of guarding a territory was first introduced by Isaacs [3]. In the game, the invader tries to move to the territory as close as possible while the defender tries to intercept and keep the invader away from the territory as far as possible. The practical application of this game can be found in surveillance and security missions for autonomous mobile robots. There are a few published works in this field since the game was introduced [19, 20]. In these works, the defender tries to use a fuzzy controller to locate the invader's position [19] or applies a fuzzy reasoning strategy to capture the invader [20]. However, all these works assume that the defender knows its optimal policy and the invader's policy. There is no learning technique applied to the players in their works. In our work, we assume that the defender or the invader has no prior knowledge of his optimal policy and the opponent's policy. We apply learning algorithms to the players and let the defender or the invader obtain its own optimal behavior after learning.

The problem of guarding a territory in Reference 3 is a differential game problem where the dynamic equations of the players are typically differential equations. In our work, we will investigate how the players learn to behave with no knowledge of the optimal strategies. Therefore, the above problem becomes a multiagent learning problem in a multiagent system. In the literature, there are a number of papers on multiagent systems [21, 22]. Among the multiagent learning applications, the predator–prey or the pursuit problem in a grid world has been well studied [22, 23]. To better understand the learning process of the two players in the game, we create a grid game of guarding a territory, which has not been studied so far.

Most multiagent learning algorithms are based on MARL methods [22]. According to the definition of the game in Reference 3, the grid game we established is a two-player zero-sum stochastic game. The minimax-Q algorithm [1] is well suited to solve our problem. However, if the player does not always take the action that is most damaging to the opponent, the opponent might have better performance using a rational learning algorithm than the minimax-Q [21]. The rational learning algorithm we use here is the WoLF-PHC learning algorithm. In this section, we run simulations and compare the learning performance of the minimax-Q and WoLF-PHC algorithms.

The problem of guarding a territory in this section is the grid version of the guarding a territory game in Reference 3. The game is defined as follows:

- We take a 6×6 grid as the playing field, as shown in Fig. 4-13. The invader starts from the upper-left corner and tries to reach the territory before the capture. The territory is represented by a cell named T in Fig. 4-13. The defender starts from the bottom and tries to intercept the invader. The initial positions of the players are not fixed and can be chosen randomly.
- Both players can move up, down, left, or right. At each time step, both players take their actions simultaneously and move to their adjacent cells. If the chosen action will take the player off the playing field, the player will stay at the current position.
- The nine gray cells centered around the defender, shown in Fig. 4-13b, is the region where the invader will be captured. A successful invasion by the invader is defined as the situation where the invader reaches the territory before the capture or the capture happens at the territory. The game ends when the defender captures the invader or a successful invasion by the



(a) (b)

Fig. 4-13. Guarding a territory in a grid world. (a) Initial positions of the players when the game starts. (b) Terminal positions of the players when the game ends. Reproduced from [5], © X. Lu.

invader happens. Then the game restarts with random initial positions of the players.

- The goal of the invader is to reach the territory without interception, or move to the territory as close as possible if the capture must happen. On the contrary, the aim of the defender is to intercept the invader at a location as far away as possible from the territory.

The terminal time is defined as the time when the invader reaches the territory or is intercepted by the defender. We define the payoff as the distance between the invader and the territory at the terminal time.

$$\text{Payoff} = |x_I(t_f) - x_T| + |y_I(t_f) - y_T| \tag{4.77}$$

where $(x_I(t_f), y_I(t_f))$ is the invader's position at the terminal time t_f and (x_T, y_T) is the territory's position. Based on the definition of the game, the invader tries to minimize the payoff while the defender tries to maximize the payoff.

4.12.1 Simulation and Results

We use the minimax-Q and WoLF-PHC algorithms introduced in Sections 4.3 and 4.11 to simulate the grid game of guarding a territory. We first present a simple 2×2 grid game to explore the issues of mixed strategy, rationality, and convergence. Next, we enlarge the playing field to a 6×6 grid and examine the performance of the learning algorithms based on this large grid.

We set up two simulations for each grid game. In the first simulation, the players in the game use the same learning algorithm to play against each

Table 4.7 Comparison of multiagent reinforcement learning algorithms.

Algorithms	Applicability	Rationality	Convergence
Minimax-Q	Zero-sum SGs	No	Yes
Nash-Q learning	Specific general-sum SGs	No	Yes
Friend-or-foe Q learning	Specific general-sum SGs	No	Yes
WoLF-PHC	General-sum SGs	Yes	No

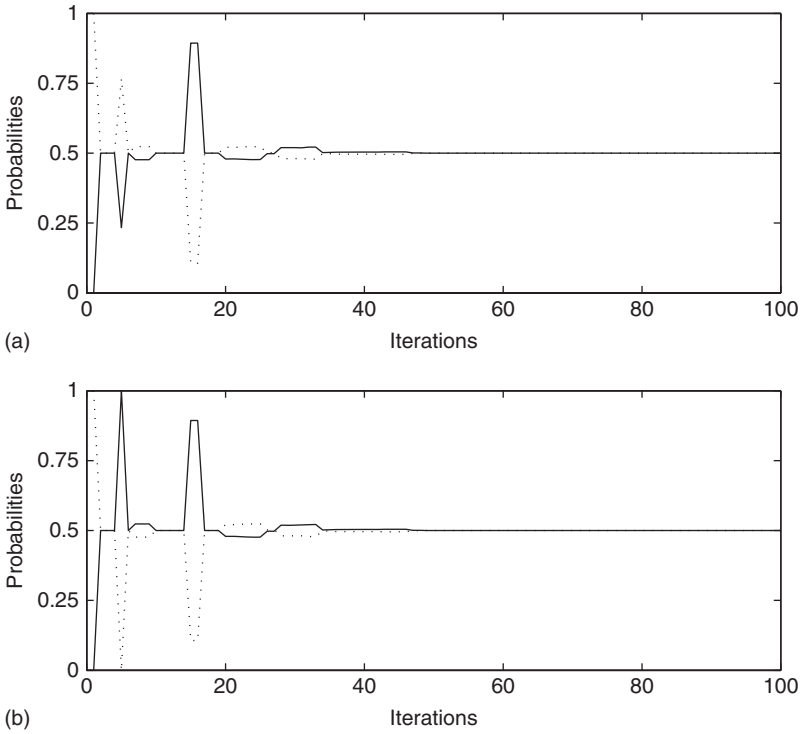


Fig. 4-14. Players' strategies at state s_1 using the minimax-Q algorithm in the first simulation for the 2×2 grid game. (a) Defender's strategy $\pi_D(s_1, a_{left})$ (solid line) and $\pi_D(s_1, a_{up})$ (dash line). (b) Invader's strategy $\pi_I(s_1, o_{down})$ (solid line) and $\pi_I(s_1, o_{right})$ (dash line). Reproduced from [5], © X. Lu.

other. We examine whether the algorithm satisfies the convergence property Fig. 4-14. In the second simulation, we will freeze one player's strategy and let the other player learn the optimal strategy against its opponent. We use the minimax-Q and WoLF-PHC algorithms to train the learner individually and compare the performance of the minimax-Q-trained player and the WoLF-PHC-trained player. According to the rationality property shown in Table 4.7, we expect the WoLF-PHC-trained defender has better performance than the minimax-Q-trained defender in the second simulation.

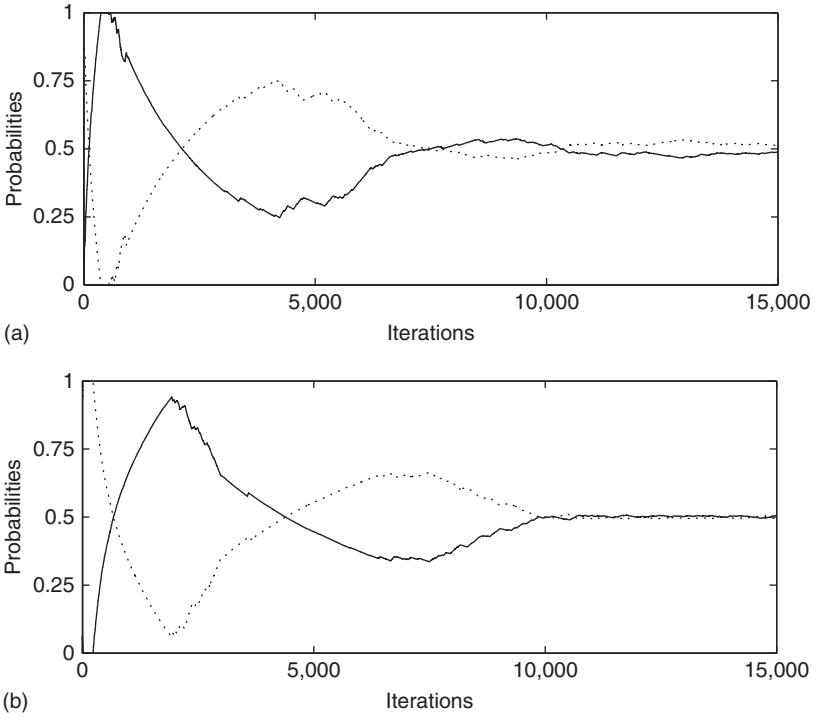


Fig. 4-15. Players' strategies at state s_1 using the WoLF-PHC algorithm in the first simulation for the 2×2 grid game. (a) Defender's strategy $\pi_D(s_1, a_{left})$ (solid line) and $\pi_D(s_1, a_{up})$ (dash line). (b) Invader's strategy $\pi_I(s_1, o_{down})$ (solid line) and $\pi_I(s_1, o_{right})$ (dash line). Reproduced from [5], © X. Lu.

We now apply the WoLF-PHC algorithm to the 2×2 grid game. According to the parameter settings in Reference 2, we set the learning rate α as $1/(10 + t/10000)$, δ_w as $1/(10 + t/2)$, and δ_l as $3/(10 + t/2)$, where t is the number of the current iteration. The number of iterations denotes the number of times the step 2 is repeated in Algorithm 4.5. The result in Fig. 4-15 shows that the players' strategies converge close to the Nash equilibrium after 15,000 iterations.

In the second simulation, the invader plays a stationary strategy against the defender at state s_1, as shown in Fig. 4-2a. The invader's fixed strategy is to move right with probability 0.8 and move down with probability 0.2. Then the optimal strategy for the defender against this invader is to move up all the time. We apply both algorithms to the game and examine the learning performance for the defender. Figure 4-16a shows that, using the minimax-Q algorithm, the defender's strategy fails to converge to its optimal strategy, whereas

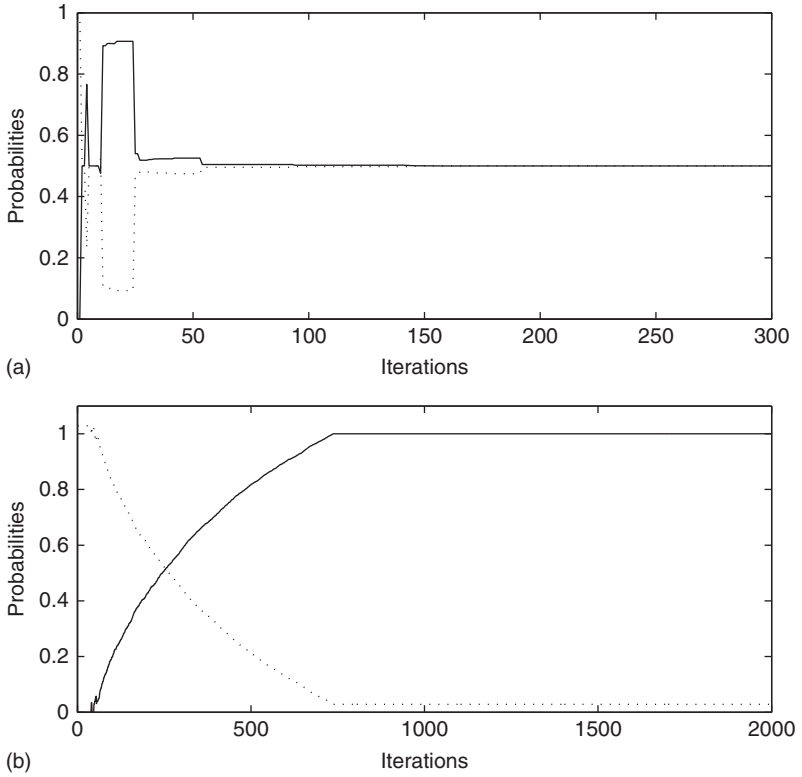


Fig. 4-16. Defender's strategy at state s_1 in the second simulation for the 2×2 grid game. (a) Minimax-Q-learned strategy of the defender at state s_1 against the invader using a fixed strategy. Solid line: probability of defender moving up; Dashed line: probability of defender moving left. (b) WoLF-PHC learned strategy of the defender at state s_1 against the invader using a fixed strategy. Solid line: probability of defender moving up; Dashed line: probability of defender moving left. Reproduced from [5], © X. Lu.

Fig. 4-16b shows that the WoLF-PHC algorithm guarantees the convergence to the defender's optimal strategy against the invader.

In the 2×2 grid game, the first simulation verified the convergence property of the minimax-Q and WoLF-PHC algorithms. According to Table 4.7, there is no proof of convergence for the WoLF-PHC algorithm. But simulation result in Fig. 4-15 shows that the players' strategies converged to the Nash equilibrium when both players used the WoLF-PHC algorithm. Under the rationality criterion, the minimax-Q algorithm failed to converge to the defender's optimal strategy in Fig. 4-16a, while the WoLF-PHC algorithm showed the convergence to the defender's optimal strategy after learning.

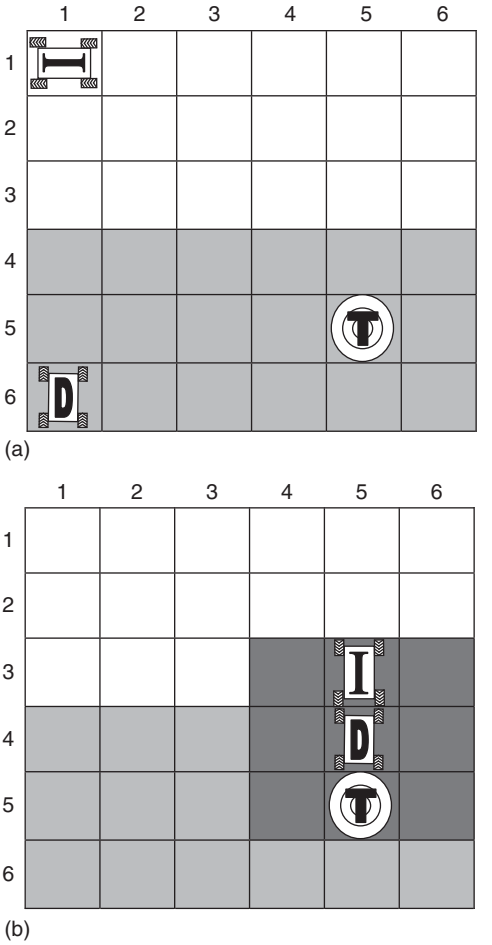


Fig. 4-17. A 6×6 grid game. (a) Initial positions of the players. (b) One of the terminal positions of the players. Reproduced from [5], © X. Lu.

We now change the 2×2 grid game to a 6×6 grid game. The territory to be guarded is represented by a cell located at $(5, 5)$ in Fig. 4-17. The position of the territory will not be changed during the simulation. The initial positions of the invader and defender are shown in Fig. 4-17a. The number of actions for each player has been changed from 2 in the 2×2 grid game to 4 in the 6×6 grid game. Both players can move up, down, left, or right. The gray cells in Fig. 4-17a is the area where the defender can reach before the invader. Therefore, if both players play their equilibrium strategies, the invader can move to the territory as close as possible with the distance of two cells shown in Fig. 4-17b. Different from the previous 2×2 grid game where we

showed the convergence of the players' strategies during the learning, in this game we want to show the average leaning performance of the players during the learning. We add a testing phase to evaluate the learned strategies after every 100 iterations. The number of iterations denotes the number of times step 2 is repeated in Algorithms 4.1 or 4.5. A testing phase includes 1000 runs of the game. In each run, the learned players start from their initial positions shown in Fig. 4-17a and end at the terminal time. For each run, we find the final distance between the invader and the territory at the terminal time. Then we calculate the average of the final distance over 1000 runs. The result of a testing phase, which is the average final distance over 1000 runs, is collected after every 100 iterations.

We use the same parameter settings as in the 2×2 grid game for the minimax-Q algorithm. In the first simulation, we test the convergence property by using the same learning algorithm for both players. Figure 4-18a shows the learning performance when both players used the minimax-Q algorithm. In Fig. 4-18a, the x-axis denotes the number of iterations and the y-axis denotes the result of the testing phase (the average of the final distance over 1000 runs) for every 100 iterations. From the result in Fig. 4-18a, the average final distance between the invader and the territory converges to 2 after 50,000 iterations. As shown in Fig. 4-17b, distance 2 is the final distance between the invader and the territory when both players play their Nash equilibrium strategies. Therefore, Fig. 4-18a indicates that both players' learned strategies converge close to their Nash equilibrium strategies. Then we use the WoLF-PHC algorithm to simulate again. We set the learning rate α as $1/(4 + t/50)$, δ_w as $1/(1 + t/5000)$, and δ_l as $4/(1 + t/5000)$. We run simulation for 200,000 iterations. The result in Fig. 4-18b shows that the average final distance converges close to the distance of 2 after the learning.

In the second simulation, we fix the invader's strategy to a random-walk strategy, which means that the invader can move up, down, left, or right with equal probability. Similar to the first simulation, the learning performance of the algorithms is tested on the basis of the result of a testing phase after every 100 iterations. In the testing phase, we play the game 1000 times and average the final distance between the invader and the territory at the terminal time for each run over 1000 runs.

We test the learning performance of both algorithms for the defender in the game and compare them. The results are shown in Fig. 4-19a and b. Using the WoLF-PHC algorithm, the defender can intercept the invader further away from the territory (distance of 6.6) than using the minimax-Q algorithm (distance of 5.9). Therefore, on the basis of the rationality criterion in

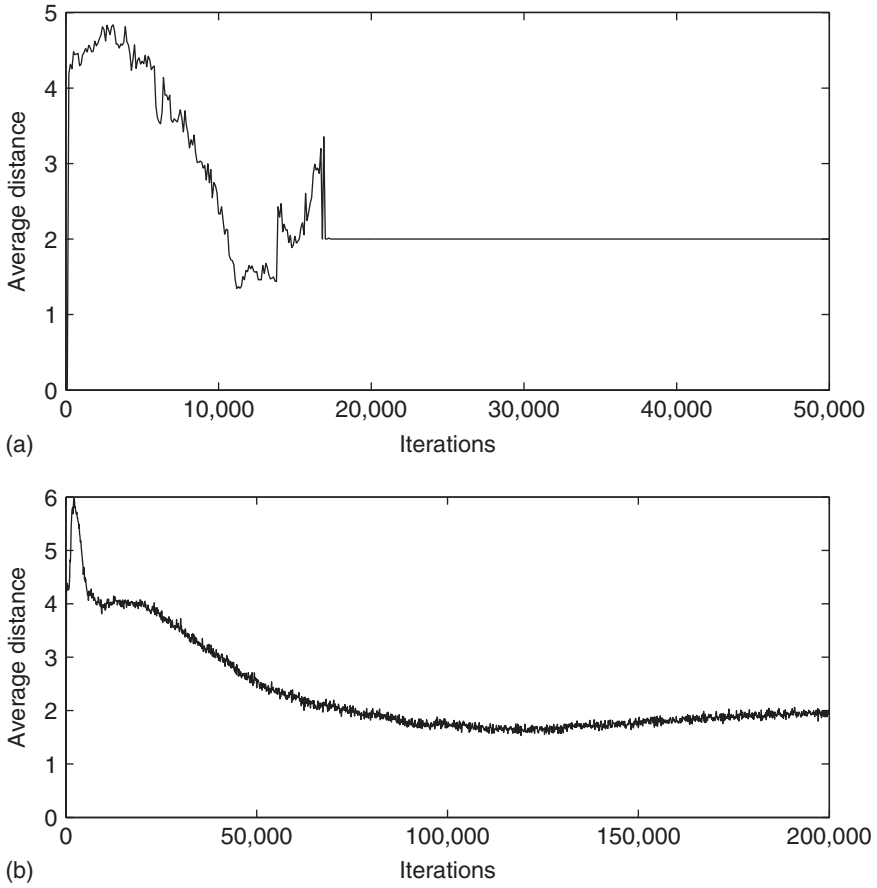


Fig. 4-18. Results in the first simulation for the 6×6 grid game. (a) Result of the minimax-Q learned strategy of the defender against the minimax-Q-learned strategy of the invader. (b) Result of the WoLF-PHC learned strategy of the defender against the WoLF-PHC-learned strategy of the invader. Reproduced from [5], © X. Lu.

Table 4.7, the WoLF-PHC-learned defender can achieve better performance than the minimax-Q-learned defender when playing against a random-walk invader.

4.13 Extension of L_{R-I} Lagging Anchor Algorithm to Stochastic Games

The proposed L_{R-I} lagging anchor algorithm is designed on the basis of matrix games. In this section, we extend the algorithm to the more general stochastic games. Inspired by the WoLF-PHC algorithm in Reference 2, we design a practical decentralized learning algorithm for stochastic games based on the

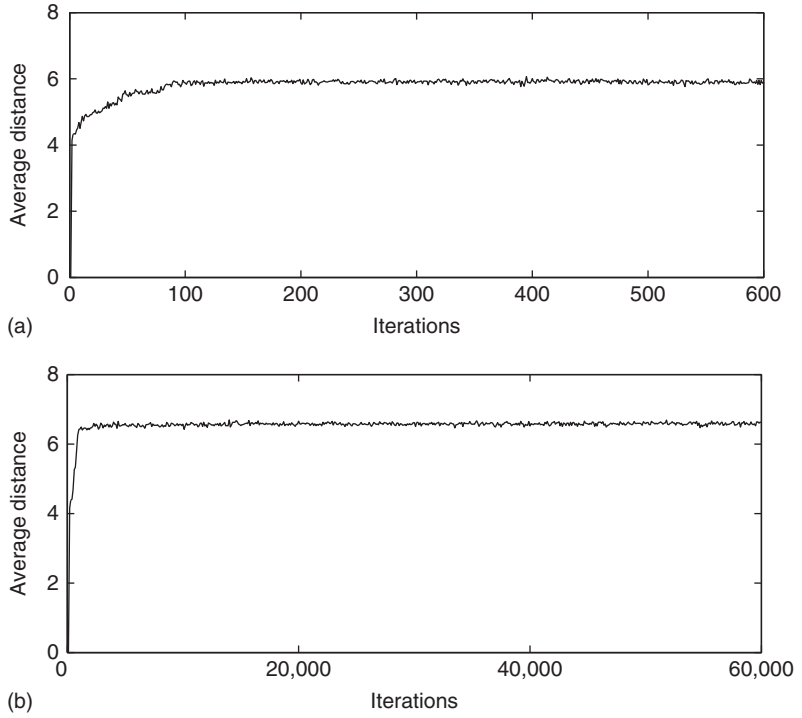


Fig. 4-19. Results in the second simulation for the 6×6 grid game. (a) Result of the minimax-Q-learned strategy of the defender against the invader using a fixed strategy. (b) Result of the WoLF-PHC-learned strategy of the defender against the invader using a fixed strategy. Reproduced from [5], © X. Lu.

L_{R-I} lagging anchor approach in (3.59). The practical algorithm is shown in Algorithm 4.6.

We now apply Algorithm 4.6 to a stochastic game to test its performance. The stochastic game we simulate is a general-sum grid game introduced by Hu and Wellman [8] and that we have already reviewed in section 4.4. Recall that. The game runs under a 3×3 grid field as shown in Fig. 4-20a. We have two players whose initial positions are located at the bottom left corner for player 1 and the bottom right corner for player 2. Both players try to reach the goal denoted as "G" in Fig. 4-20a. Each player has four possible moves which are moving up, down, left, or right unless the player is on the sides of the grid. In Hu and Wellman's game, the movement that will take the player to a wall is ignored. Since we use exactly the same game as Hu and Wellman, the possible actions of hitting a wall have been removed from the players' action sets. For example, if the player is at the bottom-left corner, its available moves are moving up or right. If both players move to the same cell at

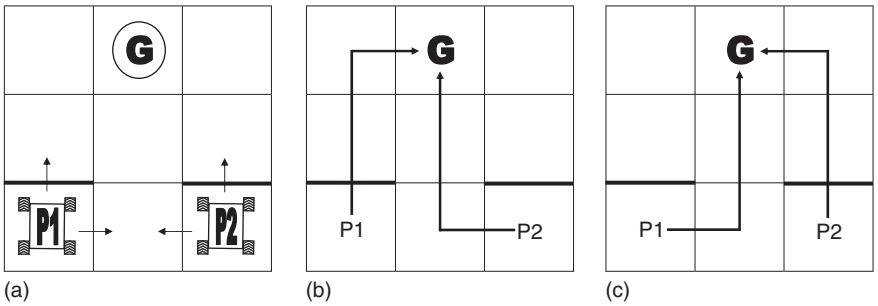


Fig. 4-20. Hu and Wellman's grid game. (a) Grid game. (b) Nash equilibrium path 1. (c) Nash equilibrium path 2. Reproduced from [24] © M. Awheda and Schwartz, H. M.

Algorithm 4.6 A practical L_{R-I} lagging anchor algorithm for player i

---

1: Initialize $Q_i(s,a_i) \leftarrow 0$ and $\pi_i(s,a_i) \leftarrow \dfrac{1}{|A_i|}$.Choose the learning rate α, η and the discount factor γ.

2: **for** Each iteration **do**

3: Select action a_c at current state s based on mixed exploration-exploitation strategy

4: Take action a_c and observe the reward r and the subsequent state s'

5: Update $Q_i(s,a_c)$

$$Q_i(s,a_c) = Q_i(s,a_c) + \alpha\Big[r_i + \gamma \max_{a'} Q_i(s',a') - Q_i(s,a_c)\Big]$$

6: Update the player's policy $\pi_i(s,\cdot)$

$$\pi_i(s,a_c) = \pi_i(s,a_c) + \eta Q_i(s,a_c)\big[1 - \pi_i(s,a_c)\big] + \eta\big[\bar{\pi}_i(s,a_c) - \pi_i(s,a_c)\big]$$

$$\bar{\pi}_i(s,a_c) = \bar{\pi}_i(s,a_c) + \eta\big[\pi_i(s,a_c) - \bar{\pi}_i(s,a_c)\big]$$

$$\pi_i(s,a_j) = \pi_i(s,a_j) - \eta Q_i(s,a_c)\pi_i(s,a_j) + \eta\big[\bar{\pi}_i(s,a_j) - \pi_i(s,a_j)\big]$$

$$\bar{\pi}_i(s,a_j) = \bar{\pi}_i(s,a_j) + \eta\big[\pi_i(s,a_j) - \bar{\pi}_i(s,a_j)\big]$$

(for all $a_j \neq a_c$)

7: **end for**

$\big(Q_i(s,a_i)$ is the action-value function, $\pi_i(s,a_i)$ is the probability of player i taking action a_i at state s and a_c is the current action taken by player i at state $s\big)$

---

the same time, they will bounce back to their original positions. The two thick lines in Fig. 4-20a represent two barriers such that the player can pass through the barrier with a probability of 0.5. For example, if player 1 tries to move up from the bottom-left corner, it will stay still or move to the upper cell with a probability of 0.5. The game ends when either of the players reaches the goal. To reach the goal in the minimum number of steps, the player needs to avoid the barrier and first move to the bottom center cell. Since both players cannot move to the bottom center cell simultaneously, the players need to cooperate such that one of the players has to take the risk and move up. The reward function for player i ($i = 1, 2$) in this game is defined as

$$r_i = \begin{cases} 100 & \text{player } i \text{ reaches the goal} \\ -1 & \text{both players move to the same cell (except the goal)} \\ 0 & \text{otherwise} \end{cases} \tag{4.78}$$

According to Reference 8, this grid game has two Nash equilibrium paths as shown in Fig. 4-20b and c. Starting from the initial state, the Nash equilibrium strategies of the players are player 1 moving up and player 2 moving left, or player 1 moving right and player 2 moving up.

We set the step size as $\eta = 0.001$, the learning rate as $\alpha = 0.001$, and the discount factor as $\gamma = 0.9$. The mixed exploration-exploitation strategy is chosen such that the player chooses a random action with probability 0.05 and the greedy action with probability 0.95. We run the simulation for 10,000 episodes. An episode is when the game starts with the players' initial positions and ends when either of the players reaches the goal. Figure 4-21 shows the result of two players' learning trajectories. We define p_1 as player 1's probability of moving up and q_1 as player 2's probability of moving up from their initial positions. The result in Fig. 4-21 shows that the two players' strategies at the initial state converge to one of the two Nash equilibrium strategies (player 1 moving right and player 2 moving up). Therefore, the proposed practical L_{R-I} lagging anchor algorithm may be applicable to general-sum stochastic games.

4.14 The Exponential Moving-Average Q-Learning (EMA Q-Learning) Algorithm

The exponential moving-average (EMA) approach is a model-free strategy estimation approach. It is a family of statistical approaches used to analyze time series data in finance and technical analysis. Typically, EMA gives the

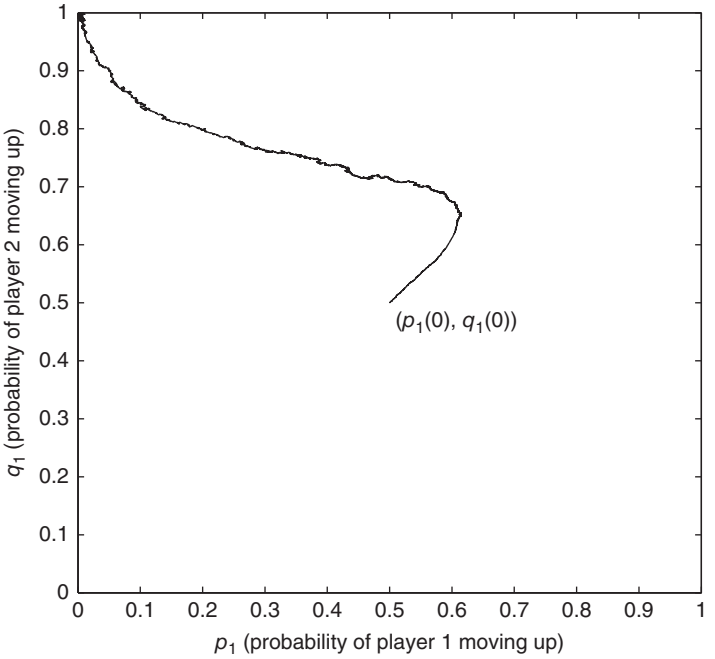


Fig. 4-21. Learning trajectories of players' strategies at the initial state in the grid game. Reproduced from [5] © X. Lu.

recent observations more weight than the older ones [25]. The EMA estimation approach is used in Reference 26 by the hyper Q-learning algorithm to estimate the opponent's strategy. It is also used in Reference 25 by the IGA agent to estimate its opponent's strategy. The EMA estimator used to estimate the strategy of the agent's opponent(s) can be described by the following equation [25, 26]:

$$\pi_{t+1}^{-j}(s) = (1 - \eta)\pi_t^{-j}(s) + \eta\vec{u}(a^{-j}) \tag{4.79}$$

where $\pi^{-j}(s)$ is the opponent's strategy, η is a small constant step size ($0 < \eta \ll 1$), and $\vec{u}(a^{-j})$ is a unit vector representation of the action a^{-j} chosen by the opponent ($-j$) at the state s. The unit vector $\vec{u}(a^{-j})$ contains the same number of elements as π^{-j}. The elements in the unit vector $\vec{u}(a^{-j})$ are all equal to zero except for the element corresponding to the action a^{-j} which is equal to 1.

In this work, we are proposing a simple algorithm that uses the EMA approach. This algorithm is called the *EMA Q-learning algorithm*. The proposed algorithm uses the EMA mechanism as a basis to update the strategy of the agent itself. Furthermore, it uses two different variable learning rates

η_w and η_l when updating the agent's strategy instead of only one constant learning rate η used in References 25, 26. The values of these variable learning rates are inversely proportional to the number of iterations. The recursive Q-learning algorithm for a learning agent j is given by the following equation:

$$Q^j_{t+1}(s,a) = (1-\theta)Q^j_t(s,a) + \theta(r^j + \zeta \max_{a'} Q^j_t(s',a')) \tag{4.80}$$

Algorithm 4.7 The exponential moving-average (EMA) Q-learning algorithm for agent j

Initialize:
learning rates $\theta \in (0,1]$, η_l and $\eta_w \in (0,1]$
constant gain k
exploration rate ϵ
discount factor ζ
$Q^j(s,a) \leftarrow 0$ and $\pi^j(s) \leftarrow \frac{1}{|A_j|}$
Repeat
(a) From the state s select an action a according to the strategy $\pi^j_t(s,a)$ with some exploration.
(b) Observe the immediate reward r^j and the new state s'.
(c) Update $Q^j_{t+1}(s,a)$ using Eq. (4.80).
(d) Update the strategy $\pi^j_{t+1}(s,a)$ by using Eq. (4.81).

The EMA Q-learning algorithm updates the strategy of the agent j by Eq. (4.81), whereas Algorithm 4.7 lists the procedure of the EMA Q-learning algorithm for a learning agent j.

$$\pi^j_{t+1}(s) = (1-k\eta)\pi^j_t(s) + k\eta\vec{u}(a) \tag{4.81}$$

where

k is a constant gain.

$$\eta = \begin{cases} \eta_w & \text{if } a = \arg\max_{a'} Q^j_t(s,a') \\ \eta_l & \text{otherwise} \end{cases}$$

$$\vec{u}(a) = \begin{cases} \vec{u}(a^j) & \text{if } a = \arg\max_{a'} Q^j_t(s,a') \\ \vec{u}(a'^j) & \text{otherwise} \end{cases}$$

$\vec{u}(a^j)$ is a unit vector representation of the action a^j with zero elements except for the element corresponding to the action a^j which is equal to 1. This is to make the EMA Q-learning learn fast when the chosen action of agent j is equal to the greedy action obtained from the agent's Q-table. On the other hand, $\vec{u}(a'^j) = \frac{1}{|A_j|-1}[\vec{1} - \vec{u}(a^j)]$. This is to make the EMA Q-learning learn cautiously and increase the opportunity of exploring the other agent's actions when the chosen action of agent j and the greedy action obtained from the agent's Q-table are different.

4.15 Simulation and Results Comparing EMA Q-Learning to Other Methods

We have evaluated the EMA Q-learning, WoLF-PHC [2], GIGA-WoLF [27], weighted policy learning (WPL) [28], and policy gradient ascent with approximate policy prediction (PGA-APP) [29] algorithms on a variety of matrix and stochastic games. We only show the EMA Q-learning, the PGA-APP, and the WPL algorithms. The results of applying the WPL, PGA-APP, and EMA Q-learning algorithms to different matrix and stochastic games are presented in this section. A comparison among the three algorithms in terms of the convergence to Nash equilibrium is provided. We use the same learning and exploration rates for all algorithms when they are applied to the same game. In some cases, these rates are chosen to be close to those used in Reference 29. In other cases, the values of these rates are chosen on a trial-and-error basis to achieve the best performance of all algorithms.

4.15.1 Matrix Games

We revisit matrix games to illustrate the improved performance of the EMA Q-learning algorithm. The EMA Q-learning, PGA-APP, and WPL algorithms are applied to the matrix games. They are also applied to the three-player matching pennies game. Figure 4-22 shows the probability distributions of the second actions for both players in the dilemma game. The EMA Q-learning, PGA-APP, and WPL algorithms are shown. In this game, the parameters of the EMA Q-learning algorithm are set as follows: $\eta_w = \frac{1}{10+i/5}$, $\eta_l = 0.01\eta_w$, $k = 1$, $\zeta = 0$, and $\theta = 0.05$ with an exploration rate $\varepsilon = 0.05$. The parameter γ is set as $\gamma = 0.5$ in the PGA-APP algorithm and the learning rate η is decayed with a slower rate in the WPL algorithm and is set as $\eta = \frac{1}{10+i/350}$. Figure 4-23 shows the probability distributions of the first actions for the three players in the three-player matching pennies game while learning with the EMA Q-learning, PGA-APP, and WPL algorithms. In this game, the

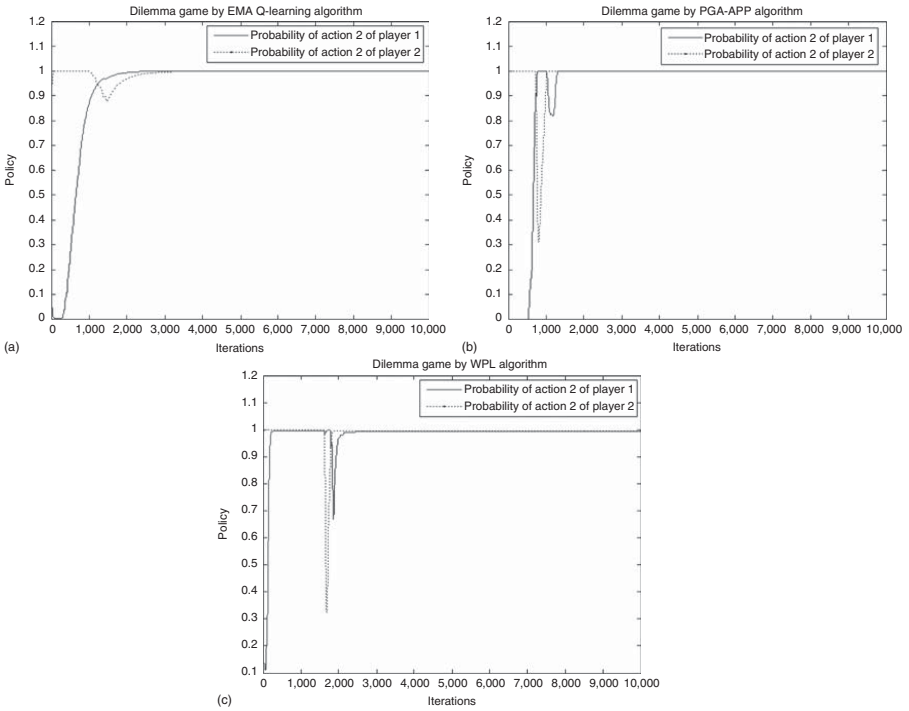


Fig. 4-22 Probability distributions of the second actions for both players in the dilemma game. (a) The EMA Q-learning, (b) PGA-APP, and (c) WPL algorithms are shown. Reproduced from [24] © M. Awheda and Schwartz, H. M.

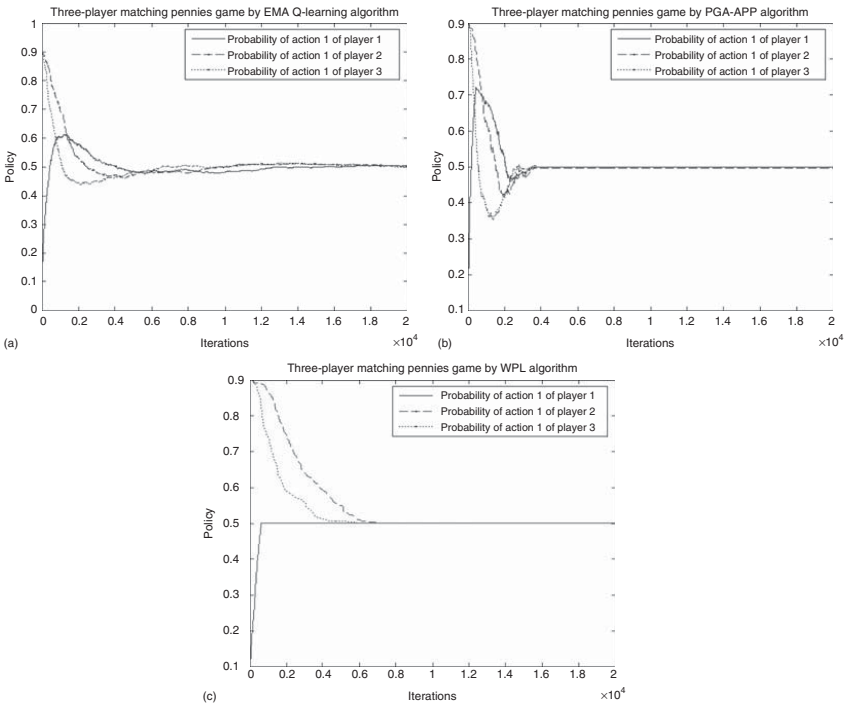


Fig. 4-23 Probability distributions of the first actions for the three players in the three-player matching pennies game. (a) The EMA Q-learning, (b) PGA-APP, and (c) WPL algorithms are shown. Reproduced from [24] © M. Awheda and Schwartz, H. M.

parameters of the EMA Q-learning algorithm are set as follows: $\eta_w = \frac{5}{5000+4i}$, $\eta_l = 2\eta_w, k = 1, \zeta = 0$, and $\theta = 0.8$ with an exploration rate $\varepsilon = 0.05$. The value of γ is set to $\gamma = 3$ in the PGA-APP algorithm and the learning rate η is decayed slowly in the WPL algorithm and is set as η as $\eta = \frac{5}{5000+i/500}$. Figure 4-24 shows the probability distributions of player 1's actions in the Shapley's game while learning with the EMA Q-learning, PGA-APP, and WPL algorithms. In this game, the parameters of the EMA Q-learning algorithm are set as follows: $\eta_w = \frac{1}{50+i}, \eta_l = 2\eta_w, k = 1, \zeta = 0$, and $\theta = 0.8$ with an exploration rate $\varepsilon = 0.05$. The learning rate η and the parameter γ in the PGA-APP algorithm are set as follows: $\eta = \frac{1}{50+i/50}$ and $\gamma = 3$. On the other hand, in the WPL algorithm, the learning rate η is decayed slowly and is set as $\eta = \frac{1}{50+i/200}$. Figure 4-25 shows the probability distributions of the first actions for both players in the biased game while learning with the EMA Q-learning, PGA-APP, and WPL algorithms. In this game, the parameters of the EMA Q-learning algorithm are set as follows: $\eta_w = \frac{1}{10+i/5}, \eta_l = 0.01\eta_w, k = 1, \zeta = 0.95$, and $\theta = 0.8$ with an exploration rate $\varepsilon = 0.05$. In the PGA-APP algorithm, the values of ζ and γ are set as follows: $\zeta = 0$ and $\gamma = 3$. In the WPL algorithm, the parameter ζ is set as $\zeta = 0$ and the learning rate η is decayed with a slower rate and is set as $\eta = \frac{1}{10+i/350}$. As shown in Figs. 4-22–4-24, the players' strategies successfully converge to Nash equilibria in all games when learning with the EMA Q-learning, PGA-APP, and WPL algorithms. It is important to mention here that the WPL algorithm successfully converges to Nash equilibrium in the three-player matching pennies game although it was presented to diverge in this game in Reference 29. On the other hand, Fig. 4-26 shows that both PGA-APP and WPL algorithms fail to converge to a Nash equilibrium in the biased game; only the EMA Q-learning algorithm succeeds to converge to a Nash equilibrium in the biased game.

4.15.2 Stochastic Games

It is important to mention here that we are only concerned about the first movement of both players from the initial state. Therefore, the figures that will be shown in this section will represent the probabilities of players' actions at the initial state.

4.15.2.1 Grid Game 1

The EMA Q-learning, PGA-APP, and WPL algorithms are used to learn grid game 1 depicted again in Fig. 4-27a. Grid game 1 has 10 different Nash equilibria [8]. One of these Nash equilibria is shown in Fig. 4-28a. Figure 4-28a

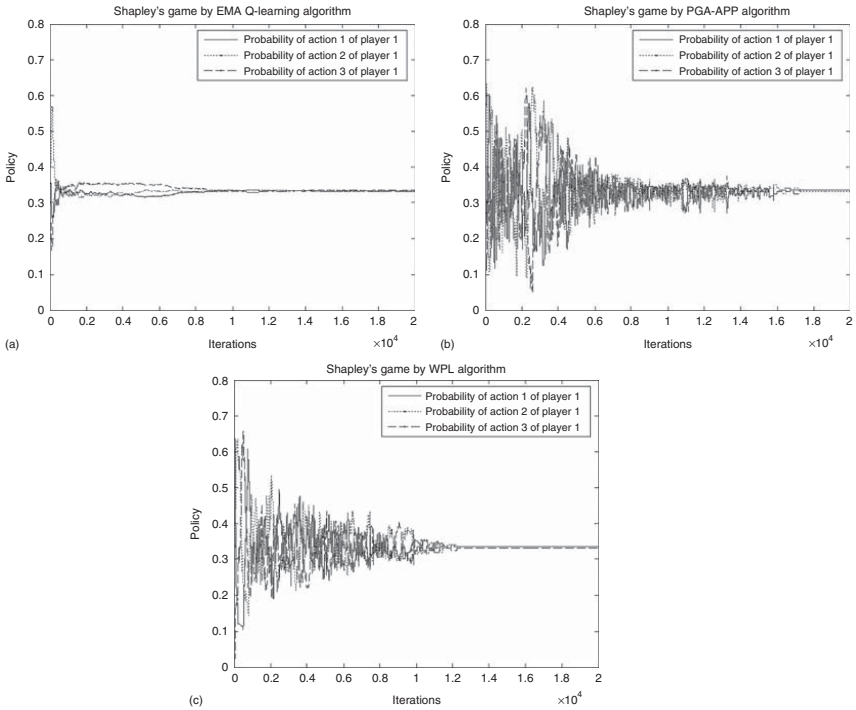


Fig. 4-24 Probability distributions of player 1's actions in the Shapley's game. (a) The EMA Q-learning, (b) PGA-APP, and (c) WPL algorithms are shown. Reproduced from [24] © M. Awheda and Schwartz, H. M.

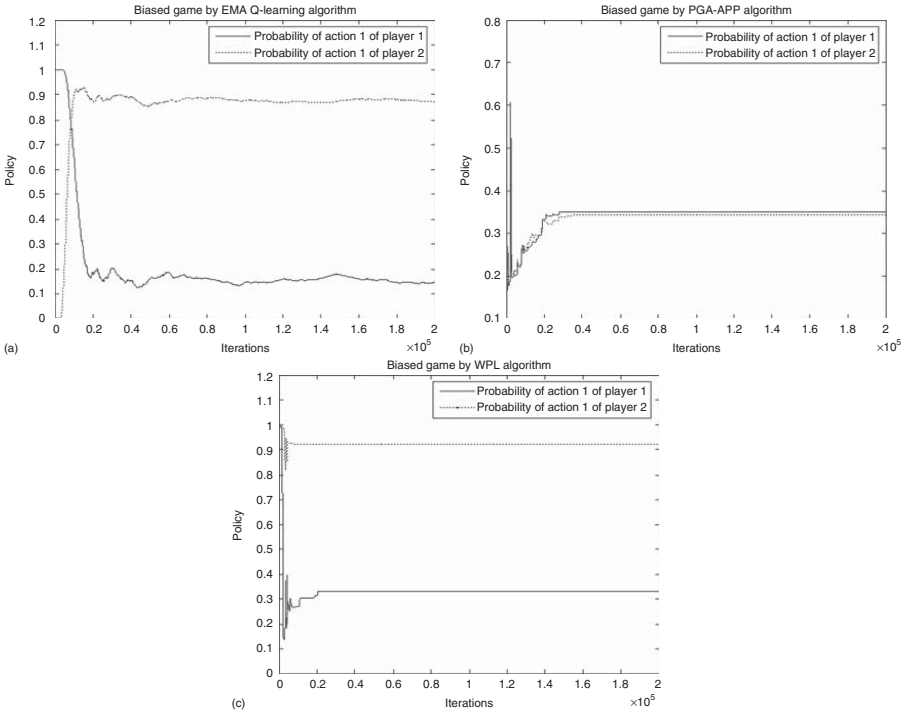


Fig. 4-25 Probability distributions of the first actions for both players in the biased game. (a) The EMA Q-learning, (b) PGA-APP, and (c) WPL algorithms are shown. Reproduced from [24] © M. Awheda and Schwartz, H. M.

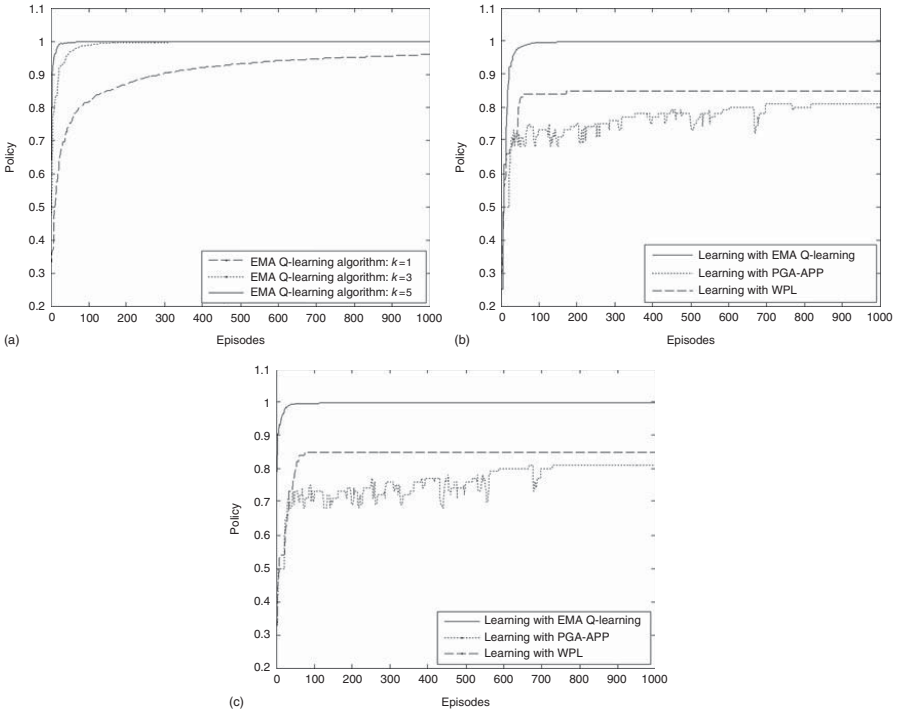


Fig. 4-26 Grid game 1. (a) Probability of action North of player 1 when learning with the EMA Q-learning algorithm with different values of the constant gain *k*. Plots (b) and (c) illustrate the probability of action North of player 1 and player 2, respectively, when learning with the EMA Q-learning, PGA-APP, and WPL algorithms. Reproduced from [24] © M. Awheda and Schwartz, H. M.

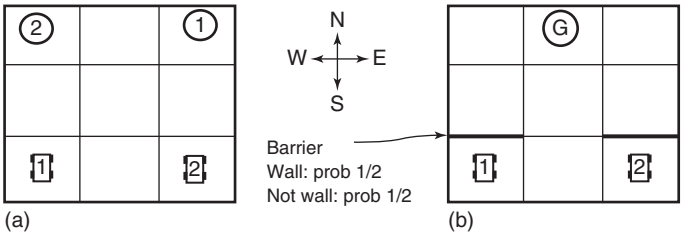


Fig. 4-27. Two stochastic games [8]. (a) Grid game 1. (b) Grid game 2. Reproduced from [24] © M. Awheda and Schwartz, H. M.

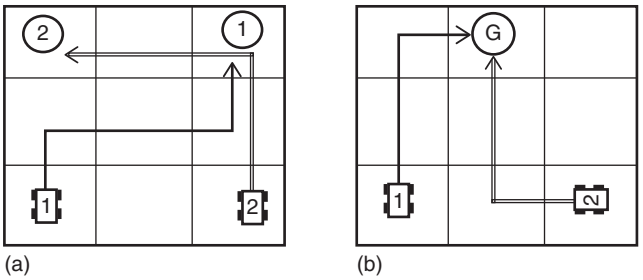


Fig. 4-28. (a) Nash equilibrium of grid game 1. (b) Nash equilibrium of grid game 2 [8] with permission from MIT press. Reproduced from [24] © M. Awheda and Schwartz, H. M.

shows that the action North is the optimal action for both players when they are at the initial state. The learning and exploration rates used by all algorithms are the same. The parameters of the EMA Q-learning algorithm are set as follows: $\eta_w = \frac{1}{10+i}$, $\eta_l = 0.001\eta_w$, $k = 5$, $\zeta = 0$, and $\theta = 0.8$ with an exploration rate $\varepsilon = \frac{1}{1+0.001i}$, where i is the current number of episodes. The values of the parameters of the PGA-APP algorithm are the same as those of the EMA Q-learning algorithm except that $\gamma = 3$ and η has a very slow decaying rate, $\eta = \frac{1}{10+i/5000}$. The WPL algorithm also has the same parameters as the EMA Q-learning algorithm except that the learning rate η has a very slow decaying rate of $\eta = \frac{1}{10+i/5000}$.

Figure 4-26a shows the probability of selecting action North by player 1 at the initial state when learning with the EMA Q-learning algorithm with different values of the constant gain k. Player 2 has similar probability distributions when learning with the EMA Q-learning algorithm with different values of the constant gain k. Figure 4-26a shows that player 1's speed of convergence to the optimal action (North) increases as the value of the constant gain k increases. Figure 4-26a shows that the probability of selecting the optimal

action North by player 1 requires almost 80 episodes to converge to 1 when k=5 and 320 episodes when k=3. However, when $k = 1$, many more episodes are still required for the probability of selecting the action North to converge to 1. Figure 4-26b and c shows the probabilities of taking action North at the initial state by both players when learning with the EMA Q-learning, PGA-APP, and WPL algorithms. This figure shows that the probabilities of taking action North by both players converge to the Nash equilibria (converge to 1) when learning with the EMA Q-learning algorithm. However, the PGA-APP and WPL algorithms fail to make the players' strategies converge to the Nash equilibria. Figure 4-26 shows that the EMA Q-learning algorithm outperforms the PGA-APP and WPL algorithms in terms of the convergence to Nash equilibria. It also shows that the EMA Q-learning algorithm can converge to Nash equilibria with a small number of episodes by adjusting the value of the constant gain k. This will give the EMA Q-learning algorithm an empirical advantage over the PGA-APP and WPL algorithms.

4.15.2.2 Grid Game 2

The EMA Q-learning, PGA-APP, and WPL algorithms are also used to learn grid game 2 as depicted in Fig. 4-27b. Grid game 2 has two Nash equilibria [8]. Figure 4-28b shows one of these Nash equilibria. It is apparent from this particular Nash equilibrium that the action North is the optimal action for player 1 at the initial state, whereas the action West is the optimal action for player 2. Thus, for the algorithms to converge to this particular Nash equilibrium, the probability of selecting the action North by player 1 should converge to 1. The probability of selecting the action West by player 2, on the other hand, should also converge to 1. The learning and exploration rates used by all algorithms are the same. The parameters of the EMA Q-learning algorithm are set as follows: $\eta_w = \frac{1}{10+i}$, $\eta_l = 0.001\eta_w$, $k = 10$, $\zeta = 0.1$, and $\theta = \frac{1}{1+0.001i}$ with an exploration rate $\varepsilon = \frac{1}{1+0.001i}$, where i is the current number of episodes. The values of the parameters of the PGA-APP algorithm are the same as those of the EMA Q-learning algorithm except that $\gamma = 3$, $\zeta = 0$, and η has a very slow decaying rate of $\eta = \frac{1}{10+i/5000}$. The WPL algorithm also has the same parameters as the EMA Q-learning algorithm except that $\zeta = 0$ and η has a very slow decaying rate of $\eta = \frac{1}{10+i/5000}$.

Figure 4-29a shows the probability of selecting action North by player 1 when learning with the EMA Q-learning, PGA-APP, and WPL algorithms. Figure 4-29a illustrates that the probability of selecting the action North by player 1 successfully converges to 1 (Nash equilibrium) when player 1 learns with the EMA Q-learning algorithm. However, the PGA-APP and WPL

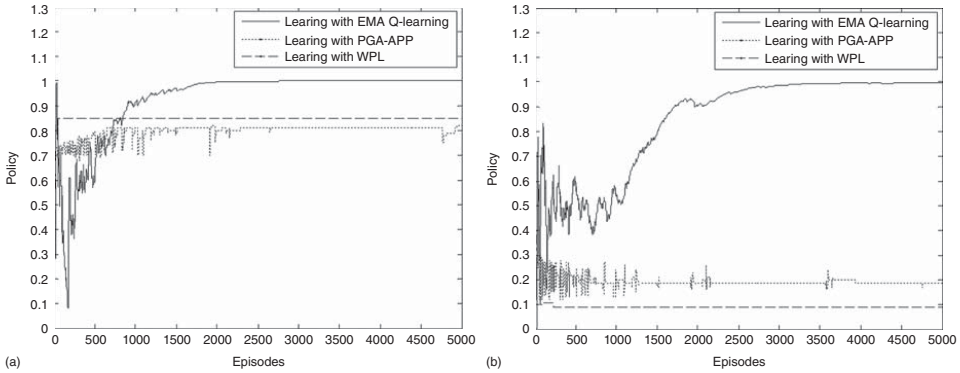


Fig. 4-29 Grid game 2. (a) Probability of selecting action North by player 1 when learning with the EMA Q-learning, PGA-APP, and WPL algorithms. (b) Probability of selecting action West by player 2 when learning with the EMA Q-learning, PGA-APP, and WPL algorithms. Reproduced from [24] © M. Awheda and Schwartz, H. M.

algorithms fail to make player 1 choose action North with a probability of 1. Figure 4-29b shows the probability of selecting action West by player 2 when learning with the EMA Q-learning, PGA-APP, and WPL algorithms. As can be seen from Fig. 4-29b, the probability of selecting action West by player 2 successfully converges to 1 (Nash equilibrium) when player 2 learns with the EMA Q-learning algorithm. The PGA-APP and WPL algorithms, on the other hand, fail to make player 2 choose action West with a probability of 1. Figure 4-29 shows that the EMA Q-learning algorithm outperforms the PGA-APP and WPL algorithms in terms of the convergence to Nash equilibria. This will give the EMA Q-learning algorithm an empirical advantage over the PGA-APP and WPL algorithms.

References

[1] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *11th International Conference on Machine Learning*, (New Brunswick, United States), July 1994, pp. 157–163, 1994.

[2] M. Bowling and M. Veloso, "Multiagent learning using a variable learning rate," *Artificial Intelligence*, vol. 136, no. 2, pp. 215–250, 2002.

[3] R. Isaacs, *Differential Games: A Mathematical Theory with Applications to Warfare and Pursuit, Control and Optimization*. New York, New York: John Wiley and Sons, Inc., 1965.

[4] M. Bowling, Multiagent Learning in the Presence of Agents with Limitations. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 2003.

[5] X. Lu, "On Multi-Agent Reinforcement Learning in Games," Ph.D. Thesis Carleton University, Ottawa, ON, Canada, 2012.

[6] M. L. Littman and C. Szepesvári, "A generalized reinforcement-learning model: Convergence and applications," in Proceedings of the 13th International Conference on Machine Learning, (Bari, Italy), July 1996, pp. 310–318, 1996.

[7] J. Hu and M. P. Wellman, "Multiagent reinforcement learning: theoretical framework and an algorithm," in Proceedings of the Fifteenth International Conference on Machine Learning (ICML 1998), Madison, Wisconsin, USA, July 24-27, 1998, pp. 242–250, 1998.

[8] J. Hu and M. P. Wellman, "Nash q-learning for general-sum stochastic games," *Journal of Machine Learning Research*, vol. 4, pp. 1039–1069, 2003.

[9] S. Abdallah, "Equilibrium in a stochastic n-person game," *Journal of Science in Hiroshima University, Series A-I,* 28: 89–93, 1964.

[10] M. L. Littman, "Friend-or-foe q-learning in general-sum games," in Proceedings of the 18th International Conference on Machine Learning, (Williams College, MA), pp. 322–328, 2001.

[11] C. Claus and C. Boutilier, "The dynamics of reinforcement learning in cooperative multiagent systems," in Proceedings of National Conference on Artificial Intelligence (AAAI-98), pp. 746–752, 1998.

[12] C. E. Lemke and J. J. T. Howson, "Equilibrium points of bimatrix games," *SIAM Journal on Applied Mathematics*, vol. 12, no. 2, pp. 413–423, 1964.

[13] D. D. Meredith, K. W. Wong, R. W. Woodhead, and R. H. Wortman, *Design and Planning of Engineering Systems*. Englewood Cliffs, New Jersey: Prentice-Hall, 1973.

[14] R. W. Cottle, J.-S. Pang, and R. E. Stone, "The linear complimentary problem," *Computer Science and Scientific Computing*, San Diego, California: Academic Press, Inc., 1992.

[15] P. De Beck-Courcelle, "Study of Multiple Multiagent Reinforcement Learning Algorithms in Grid Games", Master's thesis, Carleton University, Ottawa, ON, Canada, 2013.

[16] E. Yang and D. Gu, "A survey on multiagent reinforcement learning towards multi-robot systems," in *Proceedings of IEEE Symposium on Computational Intelligence and Games*, 2005.

[17] L. Buşoniu, R. Babuška, and B. D. Schutter, "Multiagent reinforcement learning: a survey," in *9th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pp. 1–6, 2006.

[18] X. Lu and H. M. Schwartz, "An investigation of guarding a territory problem in a grid world," in American Control Conference, pp. 3204–3210, 2010.

[19] K. H. Hsia and J. G. Hsieh, "A first approach to fuzzy differential game problem: guarding a territory," *Fuzzy Sets and Systems*, vol. 55, pp. 157–167, 1993.

[20] Y. S. Lee, K. H. Hsia, and J. G. Hsieh, "A strategy for a payoff-switching differential game based on fuzzy reasoning," *Fuzzy Sets and Systems*, vol. 130, no. 2, pp. 237–251, 2002.

[21] L. Buşoniu, R. Babuška, and B. D. Schutter, "A comprehensive survey of multiagent reinforcement learning," *IEEE Transactions on Systems, Man, and Cybernetics Part C*, vol. 38, no. 2, pp. 156–172, 2008.

[22] P. Stone and M. Veloso, "Multiagent systems: a survey from a machine learning perspective," *Autonomous Robots*, vol. 8, no. 3, pp. 345–383, 2000.

[23] J. W. Sheppard, "Colearning in differential games," *Machine Learning*, vol. 33, pp. 201–233, 1998.

[24] M. Awheda, and Schwartz, H.M., "Exponential Moving Average Q-Learning Algorithm", Proceedings of the IEEE Symposium Series on Computational Intelligence, Singapore, April 15–19, 2013.

[25] A. Burkov and B. Chaib-draa, "Effective learning in the presence of adaptive counterparts," *Journal of Algorithms*, vol. 64, no. 4, pp. 127–138, 2009.

[26] G. Tesauro, "Extending q-learning to general adaptive multi-agent systems," in *Advances in Neural Information Processing Systems 16* (S. Thrun, L. K. Saul and B. Schölkopf, eds.), (Cambridge, Massachusetts), pp. 215–250, MIT Press, 2004.

[27] M. Bowling, "Convergence and no-regret in multiagent learning," in *Advances in Neural Information Processing Systems 17* (L. K. Saul, Y. Weiss and L. Bottou, eds.), (Cambridge, Massachusetts), pp. 209–216, MIT Press, 2005.

[28] S. Abdallah and V. Lesser, "A multiagent reinforcement learning algorithm with non-linear dynamics," *Journal of Artificial Intelligence Research*, vol. 33, pp. 521–549, 2008.

[29] C. Zhang and V. Lesser, "Multi-agent learning with policy prediction," in Proceedings of the 24th National Conference on Artificial Intelligence (AAAI'10), Atlanta, GA, USA, pp. 746–752, 2010.