Study of Multiple Multiagent Reinforcement Learning Algorithms in Grid Games

by

Pascal De Beck-Courcelle

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master in Applied Science

in

Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

Abstract

This thesis studies multiagent reinforcement learning algorithms and their implementation; In particular the Minimax-Q algorithm, the Nash-Q algorithm and the WOLF-PHC algorithm. We evaluate their ability to reach a Nash equilibrium and their performance during learning in general-sum game environments. We also test their performance when playing against each other. We show the problems with implementing the Nash-Q algorithm and the inconvenience of using it in future research. We fully review the Lemke-Howson algorithm used in the Nash-Q algorithm to find the Nash equilibrium in bimatrix games. We find that the WOLF-PHC is a more adaptable algorithm and it performs better than the others in a general-sum game environment.

Acknowledgements

I would like to thank my thesis supervisor Dr Howard Schwartz for his help and guidance in this endeavour. I would also like to thank some members from the machine learning community that answered my questions related to their research: Michael Wellman, Michael Littman and Junling Hu. Finally, I would like to thank my family, especially my wife Laura and my son Julien for their support during the last three years.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

> "There is only one thing more painful than learning from experience and that is not learning from experience." - Archibald MacLeish

1.1 Overview

Reinforcement Learning is a new word for a human concept that can be traced since our origins. Humanity learned long ago that you can learn from your mistakes and that you should if you want to get better over time. Learned lessons can be passed from generation to generation by changing the way we work, interact or think. In the last fifty years, we discovered that we can make machines learn in a way similar to humans. The term Machine Learning is used to define many different applications from a military drone to a robotic carpet cleaner. It includes different types of learning like Supervised or Unsupervised Learning. In this thesis we will look into Reinforcement Learning algorithms.

The evolution of Reinforcement Learning from its origin in the 1950s to the present day has been impressive. In the last 20 years, faster computers with more memory led to the implementation of learning algorithms like the single agent Q-Learning algorithm by Watkins [1] [2]. The Q-Learning algorithm was a breakthrough in Re-

inforcement Learning and became the foundation of many algorithms.

The scope of this thesis is to compare three multiagent Reinforcement Learning algorithms in zero-sum and general-sum stochastic game environment. We looked into three algorithms used in Multiagent Reinforcement Learning: the Minimax-Q algorithm, the Nash-Q algorithm and the WOLF-PHC algorithm. We followed a logical path from the Q-Learning algorithm and introduced the different algorithms by showing the evolution from one to the other. We wanted to test their capability to play games against each other and to see which one is able to adapt to the situation. The main criteria we used to compare the algorithms were their ability to reach a Nash equilibrium strategy and their performance during learning.

1.2 Thesis Organization

This thesis research is organized into three main chapters. This is a description of each of the chapters:

- **Chapter 2** presents a description of Reinforcement Learning starting with the basic principles like the Markov Decision Process and Q-Learning. It also covers the important definitions necessary to understand the next chapters of this thesis like matrix games and the Nash equilibrium.

- **Chapter 3** presents four algorithms that were part of our research. It looks at the Minimax-Q, the Nash-Q, the Friend-or-Foe and the WOLF-PHC algorithm in detail.

- **Chapter 4** includes all the results that we obtained during our research process. It covers the implementation and the comparison of each algorithm in multiple categories. It presents a discussion of the effect of the different learning rates on the WOLF-PHC algorithm. Finally, there is a study on the adaptability of the WOLF-PHC algorithm in larger and more complex grid games.

1.3 Thesis Contributions

The work in this thesis can be categorized into the following contributions:

1. The complete implementation of the Nash-Q algorithm in Matlab with recommendations for its use in a research context.

2. A comprehensive comparison of the three studied algorithms on their learning techniques, their ability to converge in different environments, their complexity, their computational cost and their performance during learning.

3. Comparison of the different algorithms supported by simulations and empirical data.

4. Discussion on the adaptivity of the WOLF-PHC algorithm.

Chapter 2

Reinforcement Learning

2.1 Introduction

In this chapter we will introduce Reinforcement Learning by going through the major aspects of this machine learning technique. We will start by explaining Reinforcement Learning in general and the reason behind the development of this technique. It will be followed by the explanation of the Markov Decision Process which is very important in Reinforcement Learning. It defines the relationship between the different elements of the learning process. We will follow with an explanation of the Q-learning algorithm which is the basis for all the others that we will discuss in this thesis. A big part of reinforcement learning is still using the principles behind Q-learning in recent research. Finally, we will go through some very important definitions like matrix games, Nash equilibria and Stochastic games. These definitions are used in the rest of the thesis to explain the different algorithms that we tested.

2.2 Reinforcement Learning

"Reinforcement Learning is learning what to do, how to map situations to actions, so as to maximize a numerical reward signal."[3] This general definition describes

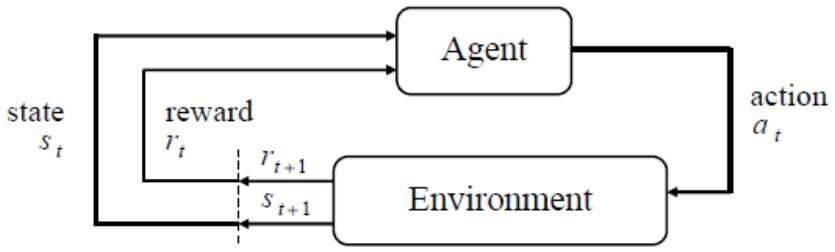


Figure 2.1: Reinforcement learning interaction between the agent and the environment

the fundamental principles of Reinforcement Learning (RL). The concept of reward depending on different actions is not a new concept in our society. We can see RL in our everyday life. The elevator system in high rise building uses a RL method to insure proper uses of each of their elevators. It tries to minimize the waiting time for all the users by finding the best way of dispatching its elevators. Another example would be that a person will be more inclined to do something if there is a positive reward at the end. Of course, it cannot represent all of human behaviour, but we can see how RL affects our life. It was only a matter of time before this type of learning would be attempted in software agents or machine learning.

To understand Reinforcement Learning we will start with the simple case of an agent in an environment. We will use the term agent to describe the learning agent. In a single agent situation, the agent interacts with the environment and discovers the actions that will earn the most rewards. The agent possesses a system of inputs and outputs. For example, the environment provides information to the agent (inputs) and the agent can interact with the environment with different actions (outputs). This is where the difference with other learning methods comes into play. There is also a reward that will be given by the environment to the agent after each action. The agent will learn that some actions produce better rewards than others and it will learn to reproduce these actions to maximize its future rewards.

Figure 2.1 from [3] illustrates clearly the different interactions between the agent

and the environment. Both the agent and the environment interact at finite time steps $t = 0, 1, 2, 3, \ldots$ [3]. This means that each interaction will be done at a predetermined time step. The environment provides the agent with the state s_t element of S, where S is the set of possible states [3]. The agent is able to choose an action a_t element of $A(s_t)$, where $A(s_t)$ is the set of possible actions in state s_t [3]. At time step $t+1$, the environment will provide a reward, r_{t+1}, which is the reward function R and a new state s_{t+1} to the agent [3]. This reward is due to its action in the previous state s_t and the transition to the new state s_{t+1}.

This is where RL comes into play. The agent will have to associate each reward with different actions and develop strategies that we call policies. We represent the policy, $\pi_t(s, a)$, as the probability that $a_t = a$ if $s_t = s$ where the t represents the next time step [3]. This means that the agent has to associate different probabilities to each action to maximize its rewards. We call this a mixed-strategy and it will be explained in this chapter. All of the RL research starts with this simple concept and develops different methods of using the reward function. At this point, we need to define the notion of expected return and discounted return. An agent's expected return is linked to the environmental reward function. The reward function can be very different from one environment to the other. It describes how the rewards are given to the agent. For example, it describes which actions or series of actions will provide what reward. In the concept of discounted reward, the expected reward diminishes over time. We can illustrate this by the equation of the expected discounted return:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{2.1}$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the discount factor [3].

As per Equation (2.1), we can see that the same reward is worth more if received now than if it is received in the future. We can change the behaviour of the agent by

changing the discount rate. If the rate is close to 0, it is called "myopic" and it means that the agent is only concerned about immediate reward [3]. If the rate is close to 1, it means that the agent considers future rewards to be more important and future rewards will have more weight.

An agent in RL has to choose an action from the state s_t provided by the environment. The information provided by the environment is called the state signal [3]. The agent needs this information from the present state s_{t+1}, but also from the previous states s_t to make the best decision possible and maximize its rewards. A state signal is said to have Markov properties if it has all the necessary information to define the entire history of the past states. We can illustrate this by looking at a chess game. The agent has all the information needed with the immediate state. The agent does not need to know every past move to choose its next action. In other words, if we can predict the next state and the next expected reward given the current state and the current reward with a probability of $\mathcal{P} = \mathsf{Pr}\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\}$ for all s', r, s_t, and a_t then the environment has the Markov property[3].

2.3 Markov Decision Process

The Markov Decision Process is a system that is characterised by the Markov property. In RL, the Markov Decision Process (MDP) is the framework used for most problems. We will explain in detail this concept because it is used throughout this thesis. For every state s and action a, the probability of each possible next state, s', is

$$P^a_{ss'} = Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\} \tag{2.2}$$

where P is the transition probabilities [3]. We can also determine the expected value of the next reward R by

$$R^a_{ss'} = E\left\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\right\} \tag{2.3}$$

where a and s are the current action and state and s' is the next state [3]. From these equations, we can formally define an MDP.

An MDP is a tuple $\langle S, A, R, T\rangle$, where S is the discrete state space, A is the discrete action space, $R : S \times A \to \mathbb{R}$ is the reward function of the agent, and $T : S \times A \times S \to [0, 1]$ is the transition function. The transition function represents a probability distribution over next states given the current state and action as per Equation (2.4).

$$\sum_{s' \in S} T\left(s, a, s'\right) = 1 \quad \forall\, s \in S,\ \forall\, a \in A \tag{2.4}$$

The next state is represented by s', the current by s, and the agent's action by a. This means that the current state is linked to a set of future states by a transition function that depends only on the current state and the set of actions possible in that state.

The MDP explains how the agent can transition from one state to the other, but the agent needs to learn from this process and choose the best possible action. The agent receives a reward, either positive or negative, every time it chooses an action depending on its current state. With time, some states will be more attractive than others. Certain actions executed in the right state will provide better rewards than others. This is where the concept of value function comes into play. We explained in the last section the concept of policy and that it represented the probability distribution of each action in every state such that

$$\sum_{a \in A} \pi\left(s, a\right) = 1 \quad \forall\, s \in S. \tag{2.5}$$

The goal of an RL agent in an MDP is to maximize its expected returns as mentioned in the previous section. The agent will modify its policy, or strategy, to ensure optimal rewards which in turn will affect the value of the different states. We can define the state-value function for a policy π as per Equation (2.6). The equation calculates the expected return for the agent if it starts in state s and follows the policy π indefinitely. The $E_\pi \{\cdot\}$ is the expected value if the agent follows the policy π and γ is the discount factor as explained in Equation (2.1) [3].

$$V^\pi(s) = E_\pi \{R_t \mid s_t = s\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s \right\} \tag{2.6}$$

There is also an action-value function that represents the expected return when taking action a in state s and following the policy π. Equation (2.7) is defined as the action-value function for policy π [3].

$$Q^\pi(s,a) = E_\pi \{R_t \mid s_t = s, a_t = a\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right\} \tag{2.7}$$

In an MDP we can use the Bellman Optimization Equation [4] to find the optimal expected return from the state-value function. The value of the present state is affected by the future states and by the quality (or value) of those states. It is easy to see that a state that is just before a goal state will be worth more than a state which is further away. Each visit in a state will provide a better estimation of $V^\pi(s)$ until we get the optimal value. This is one method used in reinforcement learning to estimate the optimal value of a state:

$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s') \right] \ \forall\, s \in S,\ \forall\, a \in A. \tag{2.8}$$

The agent tries to maximize its reward over time and one way to achieve this is to optimize its policy. A policy π is optimal when it produces better or equal return

than any other policy π'. From the equation of the state-value, we can say that in any state s, if $V^{\pi}(s) \geq V^{\pi'}(s)\ \forall\, s \in S$ then the policy π is better than π'. The previous state-value and action-value functions can be optimized as per Equations (2.9) and (2.10).

$$V^{*}(s) = \max_{\pi} V^{\pi}(s)\ \forall\, s \in S \tag{2.9}$$

$$Q^{*}(s,a) = \max_{\pi} Q^{\pi}(s,a)\ \forall\, s \in S,\ \forall\, a \in A \tag{2.10}$$

At this point, we have two functions that can be used to calculate the optimal value of a state $V^{*}(s)$ or the optimal value of actions in a state $Q^{*}(s,a)$ when using the policy π. The next equation named after its founder, Robert Bellman, provides us with a way to calculate the previous function without referring to any policies. The Bellman optimality equation for $V^{*}(s)$ is used to calculate the value of a state s, when we know the reward function $R^{a}_{ss'}$ and the transition probability $P^{a}_{ss'}$. Also, we can do the same with the state-action value function to create the Bellman optimality equation for $Q^{*}(s,a)$.

$$V^{*}(s) = \max_{a} \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V^{*}(s')\right] \tag{2.11}$$

$$Q^{*}(s,a) = \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma \max_{a'} Q^{*}(s',a')\right] \tag{2.12}$$

where π is the policy that the agent will follow from now on, $P^{a}_{ss'}$ and $R^{a}_{ss'}$ come from Equations (2.2) and (2.3), and γ is the discounted factor. The Bellman Equation represents the value of a state s by looking into the value of the possible future state s' and by calculating the probability that each of these will be visited. This is an example of dynamic programming which is used frequently in RL. The Bellman Equation can be used in an iteration algorithm over the different states and an estimation of the state-value can be calculated for each of the states. This particular function will be very important in the rest of this thesis as it is used in all the algorithms that we will

analyse.

The Markov Decision Process is a stepping stone in reinforcement learning. It gives the framework to develop multiple methods. Dynamic programming was a method developed from the MDP and Robert Bellman was a pioneer with his optimality equations. Dynamic programming methods are used to compute optimal policies that can be defined as a Markov Decision Process. In its classical form, Dynamic programming needed a perfect model of the environment [3]. We will not go into detail on dynamic programming because the algorithms in the rest of the thesis are based on the Temporal-Difference (TD) method. There are two main aspects to TD:

1. It does not need to have a complete picture of the environment or of the different dynamics inside the environment to work. It can find the optimal values only with the information collected during learning.

2. It uses a technique where the state-value V^{π} of each state s (where π is the followed policy) can be estimated by updating the value of the previous estimates every time the state is visited. With time the value will converge to an estimated optimal value.

The TD method is the starting point for our first learning algorithm: Q-Learning.

2.4 Single Agent Q-Learning

The Q-Learning algorithm was introduced by Watkins in 1989 and was a major breakthrough in reinforcement learning. We will focus on this method in this these because it is used in each algorithm evaluated later on. The simplicity of Q-Learning is one of the major attractions to the method. It uses an off-policy TD learning method. An off-policy method is a technique that is able to approximate the optimal action-value Q^{*} from the learned action-value function independently of the current

policy [3]. The agent can learn its optimal value even when choosing random actions. The only requirement is that each state is visited often enough to converge to the optimal action-value Q^*. Q-Learning was also proven to converge in [2]. Q-Learning is a method using a recursive equation of the form

$$Q_{k+1}(s_t, a_t) = Q_k(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q_k(s_{t+1}, a) - Q_k(s_t, a_t) \right] \tag{2.13}$$

where α is the learning rate and γ is the discount factor. As we can see in the equation, Q-Learning keeps track of the value of each action in each state. With time, it updates its estimate of the optimal value of each action in all the states. The table built by the algorithm is called a Q-Table and is shown in Table 2.1. The agent interacts with the environment and receives rewards depending on the state and action performed. This table represents the "quality" of each action when performed in a specific state. We define quality by how good the payoffs are for each state. The policy used by the agent has a limited effect on the algorithm as long as all the actions and states are visited and updated [3]. The learning rate α is a variable used to change how the new information will override the old. This means that a learning rate of zero will stop the learning and a learning rate of one will completely change the old data with the new one. This is an important notion in RL. During learning it is preferable to decrease the learning rate over time because we do not want to relearn the same values over and over. A constant learning rate has advantages also in certain circumstances. We will go over the principles of the learning rate in Chapter 4 Section 5. Q-Learning was the basis for many algorithms and is still part of research and development in RL. In the following chapters we will look into three algorithms that have come from Q-Learning : Minimax-Q, Nash-Q and Friend-or-Foe-Q.

		Actions	
	Q-values	a_1	a_2
States	s_1	$Q(s_1, a_1)$	$Q(s_1, a_2)$
	s_2	$Q(s_2, a_1)$	$Q(s_2, a_2)$
	s_3	$Q(s_3, a_1)$	$Q(s_3, a_2)$
	s_4	$Q(s_4, a_1)$	$Q(s_4, a_2)$

Table 2.1: Q-table for a learning agent with two actions and four states

2.5 Matrix Games

In game theory we can separate games by the way rewards are acquired. In a zero-sum game each player will have the exact opposite rewards. In a zero-sum game if Player 1 gets a positive reward of 1 then Player 2 will receive a negative reward of -1. In a general-sum game the reward functions are different for each player. We will analyse games with both type of rewards. In addition, there are two types of strategies, or policies, in games. We refer to a pure strategy if the player's action is predetermined in all states. The actions of the player will always be the same throughout the game. The other is the mixed strategy where the player will have a probability assigned to each action in its policy. The agent's actions are not predetermined and it could chose any of the actions as per the probability assigned to each action. We will examine this strategy in more detail.

A stage game is a system with multiple agents and that has a fixed state. A matrix game is a stage game with two players for which we can represent the rewards for each player in a matrix depending on their actions. We will define in detail the matrix game because it is used in the Nash-Q algorithm. We define two $m \times n$ matrices referred as A and B. In both matrices, Player 1 is the row player and Player 2 is the column player. Each value of matrix A is represented by a_{ij} and the same goes for B with b_{ij}, where i is the index of the action taken by Player 1 and j is the index of the action taken by Player 2. If Player 1 chooses action i and Player 2 chooses action j, then the expected reward for each player is a_{ij} and b_{ij} respectively [5]. Each possible

action for Player 1 and Player 2 are represented as rows and columns respectively in the matrices. The policy π_i^A is the probability that Player 1 will choose action i, where $i = 1, \cdots, m$ and $\sum_{i=1}^{m} \pi_i^A = 1$. The policy π_j^B is the probability that Player 2 will choose action j, where $j = 1, \cdots, n$ and $\sum_{j=1}^{n} \pi_j^B = 1$. Equations (2.14) represent the expected payoff for Player 1 and Player 2 respectively. The symbol Π^A will represent the set of all the mixed-strategies for Player 1 and Π^B for Player 2 i.e. $\pi_i^A \in \Pi^A$ and $\pi_j^B \in \Pi^B$.

$$E\{R_A\} = \sum_{i=1}^{m}\sum_{j=1}^{n} \pi_i^A a_{ij} \pi_j^B \quad \text{and} \quad E\{R_B\} = \sum_{i=1}^{m}\sum_{j=1}^{n} \pi_i^A b_{ij} \pi_j^B \tag{2.14}$$

To understand this concept, we will explain a very simple game: the matching pennies game. In this game each player shows one side of a penny each turn. If both pennies show either two heads or two tails, player 1 wins, but if one shows heads and the other shows tails, player 2 wins. The reward function for this game is simple: $R = 1$ if you win and $R = -1$ if you lose. The matrix in Table 2.2 shows the rewards function for player 1 and for player 2. We can also see that in this situation, we have a zero-sum game because the rewards for player 2 are the exact opposite of player 1 : $R1 = -R2$.

$$R_1 = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \Big\| \; R_2 = -R_1 = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix}$$

Table 2.2: Reward matrices in matching pennies game

In this game, each player has to make a decision each turn. It is easy to prove that if both players choose heads and tails with the same probability that they will receive the same amount of rewards. The solution for this game is to play heads 50% of the time and to play tails 50% of the time. This would maximize the rewards for each player and they would reach an equilibrium. This equilibrium is called the Nash Equilibrium.

2.6 Nash Equilibrium

We can define the Nash Equilibrium as the point where each player is maximizing its rewards and any changes in strategy would make it worse or stay the same. In a simple game like the matching pennies game, as explained earlier, it means that if player 1 tries to play tails or heads more often, it would result in less rewards in the end.

The Nash Equilibrium in a matrix (or stage) game can be defined as a collection of all the players' strategies $(\pi_1^*, \ldots, \pi_n^*)$ such that

$$\begin{aligned} &V_i\left(\pi_1^*, \ldots, \pi_i^*, \ldots, \pi_n^*\right) \quad \geq V_i\left(\pi_1^*, \ldots, \pi_i, \ldots, \pi_n^*\right), \\ &\forall \pi_i \in \Pi_i, i = 1, \ldots, n \end{aligned} \tag{2.15}$$

where V_i is player i's value function which is player i's expected reward given all players' strategies, and π_i is any strategy of player i from the strategy space Π_i [6].

Many game theory papers studied the mixed strategy Nash equilibria in zero-sum and general sum games. The proof that any n-player game with a fixed number of actions in the player action space possess at least one mixed strategy equilibrium was given by John Nash in 1951 [7] and his name was given to that equilibrium. The main point of the Nash Equilibrium model is that each player's strategy is affected by the other player's strategy and that each player tries to maximize their own reward function. If all the players are reasonable, they will reach equilibrium where all the strategies are the best response to each other: the Nash Equilibrium [8].

The Nash Equilibrium in a simple matrix game can be found with linear programming. We will explain the procedure in more detail. Linear programming is used to solve optimization problem by using linear function and specified constraints. It is used to optimize a function by finding the optimal values of x while respecting the constraints. In this situation, we want to optimize the state-value function by finding

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \Bigg\| B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Table 2.3: Reward matrices for two players and two actions matrix game

the optimal value of the policies. From the definition of the Nash Equilibrium we can find the equations needed to maximize the expected value for both players. To illustrate this, we will use a general example of a two player matrix game. The reward matrix for both players are in Table 2.3 using the notation from an earlier section.

For player 1, we need to maximize V_1 by finding the optimal π_1^A and π_2^A. We get the following inequalities :

$$a_{11}\,\pi_1^A + a_{12}\,\pi_2^A \geq V_1 \tag{2.16}$$

$$a_{21}\,\pi_1^A + a_{22}\,\pi_2^A \geq V_1 \tag{2.17}$$

$$\pi_1^A + \pi_2^A = 1 \tag{2.18}$$

$$\pi_i^A \geq 0,\ i = 1, 2 \tag{2.19}$$

For player 2, we need to maximize V_2 by finding the optimal π_1^B and π_2^B. We get the following inequalities :

$$b_{11}\,\pi_1^B + b_{12}\,\pi_2^B \geq V_2 \tag{2.20}$$

$$b_{21}\,\pi_1^B + b_{22}\,\pi_2^B \geq V_2 \tag{2.21}$$

$$\pi_1^B + \pi_2^B = 1 \tag{2.22}$$

$$\pi_i^B \geq 0,\ i = 1, 2 \tag{2.23}$$

With these equations we can use linear programming to find the values for π_1^A, π_2^A, π_1^B and π_2^B. As mentioned earlier, we will use the matching pennies game as our example. The matching pennies game has two players with two actions each and the reward matrix is shown in Table 2.2. We changed the equations found for the general

game for this specific game. We can also simplify π_1^A and π_2^A because there is only two actions, as such we know that $\pi_2^A = 1 - \pi_1^A$.

$$2\,\pi_1^A - 1 \geq V_1 \tag{2.24}$$

$$-2\,\pi_1^A + 1 \geq V_1 \tag{2.25}$$

$$0 \leq \pi_1^A \leq 1 \tag{2.26}$$

After using linear programming, we find the optimal value of 0.5 for π_1^A. We can do the same with player 2 and we also get 0.5. This proves the earlier assumption that the best strategy in the matching pennies game is to play heads half the time and tails the other half.

2.7 Stochastic Games

We looked at the Markov Decision Process where we have a single player evolving in a multiple states environment. We will now look into stochastic games where there are multiple players interacting in a multiple states environment. Stochastic games, also called Markov games, have the Markov property which means that the future state only depends on the present state and action [6]. In a stochastic game, each state can be represented as a matrix game, where the joint action of the player determines the future state and rewards. We can define an n-player stochastic game as tuple $\langle S, A_1, \ldots, A_n, R_1, \ldots, R_n, T \rangle$, where S is the state space, A_i is the action space of player $i\,(i = 1, \ldots, n)$, $R_i : S \times A_1 \times \ldots \times A_n \to \mathbb{R}$ is the payoff function for player i and $T : S \times A^1 \times \ldots \times A^n \times S \to [0, 1]$ is the transition function. The transition function represents a probability distribution over the next states given the current state and the joint action of the players as per Equation (2.27). This equation is very

similar to the Markov Decision Process transition function in Equation (2.4).

$$\sum_{s' \in S} T\left(s, a_1, \ldots, a_i, s'\right) = 1 \quad \forall\, s \in S,\, \forall\, a_i \in A_i, i = (1, \ldots, n) \tag{2.27}$$

The future state is represented by s', the current state by s and the agent's actions by $(a_1, \ldots, a_i)$ where $i = (1, \ldots, n)$ is the number of players. This means that the current state is linked to a set of future states by a transition function that depends only on the current state and the set of actions possible in that state. A repeated game is a stage game that is repeated multiple times until it reaches an end state. Generally, each state is visited more than once in a stochastic game, making it a type of repeated game.

Chapter 3

Multiagent Reinforcement Learning Algorithms

3.1 Introduction

In the previous chapter, we looked at the definition of Reinforcement Learning and the main theories behind it. We will now explain in detail four RL algorithms: Minimax-Q Learning, Nash-Q Learning, Friend-or-Foe Q Learning and Win-or-Learn-Fast Policy Hill Climbing (WOLF-PHC). These algorithms were chosen because they represent an evolution from the Q-Learning algorithm and provide an insight into multiagent learning. We will start with the Minimax-Q learning algorithm because it is one of the first adaptations of the original Q-Learning algorithm and because it is still current. We will follow with Nash-Q Learning where the Nash Equilibrium is the basis of the algorithm. It was designed to reach a Nash Equilibrium strategy between two fully competitive players. We will also look at the Friend-or-Foe Q Learning algorithm which comes from the designer of the Minimax-Q learning algorithm. It was designed from the Nash-Q Learning and updated for an environment where the learning has friends and/or foes. Finally, we will study the WOLF-PHC learning algorithm which

introduces a variable policy update technique. It is a single agent algorithm like Q-learning but it works very well in a multiagent environment.

3.2 Minimax-Q Learning Algorithm

The Minimax-Q algorithm was developed by Littman in 1994 [9] when he adapted the value iteration method of Q-Learning from a single player to a two player zero-sum game. It is important to state that his research was done only for a two player zero-sum game and we will discuss the effect in a general-sum game. He developed the algorithm starting from the definition of an MDP and adapted it for a Markov game or stochastic game as seen in the previous chapter. The algorithm was designed to work with two agents or players. This is for a fully competitive game where players have opposite goals and reward functions ($R_1 = -R_2$). Each agent tries to maximize its reward function while minimizing the opponent's. We can see that Algorithm 1 is very similar to the Q-learning algorithm discussed in the previous chapter.

The agent starts by initializing its Q-table $Q(s, a_1, a_2)$ and its state-value function $V(s)$. These initial values could be any random value, but we choose a value of one arbitrarily. It is also the value used by the author Michael Littman [9]. As seen in the theory [3], those values do not affect the end results of the algorithm, but it could take more time to process and reach an equilibrium. We also initialize the policy for the agent by putting the same probability $1/|A_1|$ for each action $a \in A$ where $|A_1|$ represent the total number of actions available to each player. The learning rate α starts at one and will decrease at the rate of the *decay* variable. We chose to have a decaying learning rate because it reduces over time the changes to the learned values. It gives more importance to the value learned in the beginning. These values change depending on the number of steps necessary for the specific game that the agent is trying to learn. We are using an exploit-explore method when choosing the action to

Algorithm 1 Minimax-Q for two players

Initialization

$\forall\ s \in S$, $a_1 \in A_1$ and $a_2 \in A_2$

Let $Q(s, a_1, a_2) = 1$

$\forall\ s \in S$

Let $V(s) = 1$

$\forall\ s \in S, a_1 \in A_1$

Let $\pi(s, a_1) = 1/|A_1|$

Let $\alpha = 1$

Loop

In state s

Choose a random action from A_1 with probability ε

If not a random action, choose action a_1 with probability $\pi(s, a_1)$

Learning

In state s'

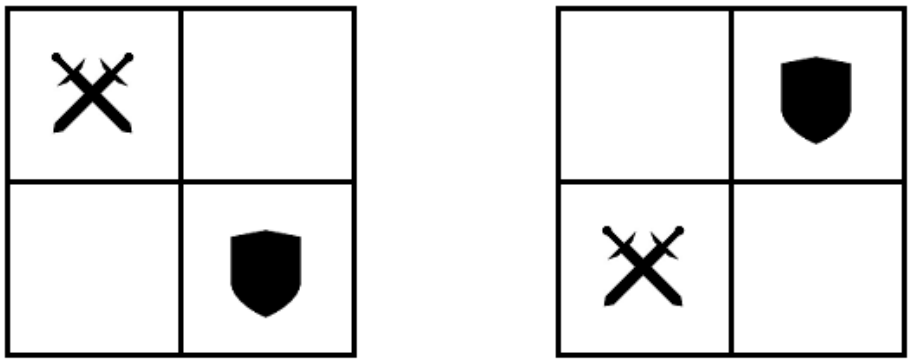
The agent observes the reward R related to action a_1 and opponent's action a_2 in state s

Update Q-Table of player 1 with equation $Q[s, a_1, a_2] = (1 - \alpha) * Q[s, a_1, a_2] + \alpha(R + \gamma * V[s'])$

Use linear programming to find the values of $\pi(s, a_1)$ and $V(s)$ with the equation $V(s) = \max_{\pi \in PD(A_1)} \min_{a_2 \in A_2} \sum_{a_1 \in A_1} Q(s, a_1, a_2)\, \pi_{a_1}$

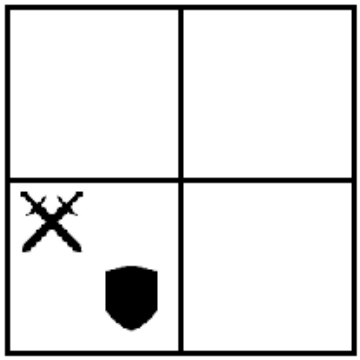
$\alpha = \alpha$ * decay

End Loop



a) Initial state

b) Attacker win state



c) Defender win state

Figure 3.1: Attacker-Defender grid game

be taken at each step. This means that every time the agent chooses its next action, it will either choose an action randomly or choose to follow its policy (greedy). The ε represent the probability that the agent will choose an action at random. The exploit-explore method has proven to be a better approach to learning than the fully greedy approach. In the experimental results chapter, we will see the effect of ε on the algorithm.

To understand the Minimax-Q algorithm we will go through a cycle of update for the simple pursuit and evasion game. Figure 3.1 a) shows the initial state of the game

$$R_{Att} = \begin{bmatrix} 10 & -1 \\ -1 & 10 \end{bmatrix} \Big\| R_{Def} = -R_{Att} = \begin{bmatrix} -10 & 1 \\ 1 & -10 \end{bmatrix}$$

Table 3.1: Reward matrices in Attacker-Defender game

with the two players. There is a defender in the lower right corner and an attacker in the top left corner. The goal of the defender is to block the attacker by moving into the same cell. The goal of the attacker is to move into an empty cell. The reward matrix is shown in Table 3.1. Each player has two actions. The defender can move up or left and the attacker can move down or right. This environment only has two states: the initial state s and the end state s'. To make the example simple to understand we will add a time t variable to $Q_t\left[s, a_{Att}, a_{Def}\right]$ in the updating equation where a_{Att} is the attacker action and a_{Def} is the defender action. We start at time $t = 0$ and initialize the parameters : $\alpha = 0.2$, $\varepsilon = 0.2$ and $\gamma = 0.9$. We also need to initialize the policy values for each player. We usually give equal probability to each possible action. In this situation we want to demonstrate the different aspects of the updating technique so we will give each player different starting probabilities for their actions. The attacker will choose to go *down* and *right* with probability 0% and 100%, respectively. The defender will choose to go *up* and *left* with probability 100% and 0%, respectively.

In state s, each player will choose an action following the exploration rule which means that the defender has a probability of choosing an action at random of 20%. The same apply for the attacker. For this example, the attacker will choose an action randomly and the defender will choose its action as per $\pi\left(s, a_{Def}\right)$ action. The attacker will choose randomly to go *down* and the defender will choose from its policy to go *up* because it has 100% probability to choose *up*.

In state s' the players will move in different cells. The attacker will receive a reward of 10 and the defender a reward of -10. The attacker updating equation would look like: $Q_1^{Att}\left[s, down, up\right] = (1 - 0.2) * Q_0^{Att}\left[s, down, up\right] + 0.2 * (R_{Att} + 0.9 * V\left[s'\right])$.

We know that the Q-value Q_0^{Att} and Q_0^{Def} equal 1 from the initiation. The same goes for the state-values $V^{Att}[s']$ and $V^{Def}[s']$. The new Q-value for the attacker is $Q_1^{Att}[s, down, up] = 0.8 * 1 + 0.2 * (10 + 0.9 * 1) = 2.98$. We can do the same thing with the defender but with a reward of -10 which gives: $Q_1^{Def}[s, down, up] = 0.8 * 1 + 0.2 * (-10 + 0.9 * 1) = -1.02$.

After updating the Q-table for each player, we need to find the new policy values $\pi(s, a_{Att})$ and $\pi(s, a_{Def})$ by using linear programming with the equation $V(s) = \max_{\pi \in PD(A_{Att})} \min_{a_{Def} \in A_{Def}} \sum_{a_{Att} \in A_{Att}} Q(s, a_{Att}, a_{Def}) \pi_{a_{Att}}$. We found that $\pi(s, a_{Att}) = [0.92, 0.08]$ and $\pi(s, a_{Def}) = [1, 0]$. This means that the attacker would choose to go *down* and *right* with a probability of 92% and 8%, respectively. The defender keeps the same policy and will choose to go *up* 100% of the time. This result is not the best policy for this game, but it will change over time. After each iteration, the agents will update their Q-values with more rewards and it will change the values of the policies found by linear programming. In our example, the policy that we are looking for is to choose either action 50% of the time.

The Minimax-Q learning algorithm is interesting because it takes the single agent Q-learning and adapts it for a multiagent environment. It also uses the linear programming to maximize its rewards by minimizing its opponents rewards. In a zero-sum stochastic game, the Minimax-Q learning algorithm will converge and find the Nash equilibrium strategy. It was not proven to converge in general-sum games and it is a limitation. A general-sum game environment can give more flexibility because the rewards do not need to respect $R_1 = -R_2$ where R_1 and R_2 are the rewards for Player 1 and Player 2. The next algorithm that we will discuss has been proven to converge to a Nash equilibrium for general-sum stochastic games.

3.3 Nash Q-Learning Algorithm

This algorithm was developed by Hu and Wellman [10] for multi-agent general-sum stochastic games. It uses the principle of the Nash equilibrium [7] where "each player effectively holds a correct expectation about the other players' behaviours, and acts rationally with respect to this expectation"[8]. In this situation "rationally" means that the agent will have a strategy that is a best response for the other players' strategies. Also, any change in strategy would put the agent in a worst position. The Nash-Q algorithm is derived from the single agent Q-learning developed by Watkins and Dayan in 1992 [2] and adapted to a multiagent environment.

$$Q_{t+1}(s_t, a_t) = (1 - \alpha_t) Q_t(s_t, a_t) + \alpha_t \left[r_t + \gamma \max_a Q_t(s_{t+1}, a) \right] \tag{3.1}$$

It starts from single agent Q-learning where the agent builds a Q-table and tries to maximize its payoff. Because it is a single agent algorithm, the agent only takes into consideration its own payoff. Q-learning can be adapted for a multi-agent general-sum stochastic game environment where there are two or more agents. In a stochastic game, the agent payoff is based on the joint action of all the agents and the current state [8].

Nash Q-Learning is more complex than multiagent Q-learning because each player needs to keep track of the other players actions and rewards. Instead of Q-values, this algorithm keeps Q-tables with Nash Q-values. The authors define the Nash Q-value as "the expected sum of discounted rewards when all agents follow specified Nash equilibrium strategies from the next period on" [8]. The authors used this notation because each agent will try to reach a Nash Equilibrium strategy to maximize its payoff. This means that each agent will reach an optimal strategy by observing all the other agents' actions and rewards.

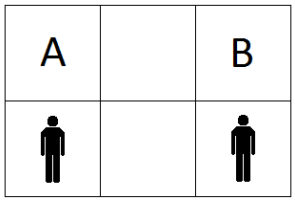
The updating method is relatively similar to the Q-Learning method. The main

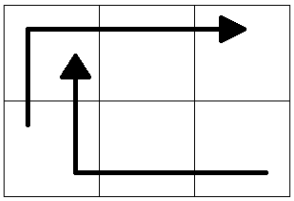
difference is that the update is based on the future Nash Equilibrium payoff considering all the other agents' actions, instead of the agent's own maximum payoff [8]. The Nash-Q function is defined by the author as Equation (3.2) [8].

$$Q_i^* \left(s, a^1, \ldots, a^n\right) = r^i \left(s, a^1, \ldots, a^n\right) + \gamma \sum_{s' \in S} p \left(s' | s, a^1, \ldots, a^n\right) v^i \left(s', \pi_*^1, \ldots, \pi_*^n\right) \tag{3.2}$$

The Nash equilibrium strategy in the equation is written as $(\pi_*^1, \ldots, \pi_*^n)$ which can be confusing because each symbol π represents more then just the next state policy. It represents all the policies from the next state until the end state which form a Nash equilbrium strategy. This strategy is any series of actions where all the agents maximize their rewards. An example of a Nash equilibrium for a two-player game is shown in Figure 3.2. The goal of the player in the lower left is to reach the top right cell and the player in the lower right needs to reach the top left cell. In this situation, Figure 3.2 b) shows one possible Nash equilibrium strategy. The Nash equilibrium strategy is very important in the Nash-Q learning algorithm. To return to Equation (3.2), we also have $r^i (s, a^1, \cdots, a^n)$ which is agent i's reward in state s and under joint action $(a^1, \cdots, a^n)$. This is a standard reward function where the reward is based on the state and the action of each player. The last part of the equation given by the term $v^i (s', \pi_*^1, \ldots, \pi_*^n)$ represents agent i's total discounted reward over infinite periods starting from state s' given that each agent follows the equilibrium strategies as explained earlier.

The Nash-Q function is used to update the Q-values in the Q-tables for each agent. From the Nash-Q function we can say that the Nash Q-values are the sum for agent i of its current reward and its future rewards when all the agents follow a joint Nash Equilibrium strategy [8]. In other words it means that the learning agent will update its Nash Q-value depending on the joint strategy of all the players and not only its own expected payoff. "The updating rule is based on the expectation that agents





a) Grid Game with 2 players and 2 goals b) Possible Nash equilibrium strategy

Figure 3.2: WOLF-PHC in Grid Game 1 with Constant Learning Rate α for Player 2

would take their equilibrium actions in each state." [8]

To be able to find the Nash equilibrium strategy the agent has to build a Q-table for each player and also needs to keep its own Q-table up to date. In each state, the agent has to keep track of every other agent's actions and rewards. With this information, it will be able to update its Q-tables and determine the Nash Equilibrium for future states. The agent has to calculate which action represents the equilibrium strategy. It will look at the next state expected payoff value (Nash Q-value) and determine the Nash equilibrium action. The Nash equilibrium required for the algorithm will be calculated from the two matrices created from the Nash Q-values of each player. These matrices represent the expected payoff of the next state for each possible action. To understand this method, we need to see the algorithm for Nash Q-Learning and then we can explain the process of choosing the equilibrium strategy. Algorithm 2 presents a more detailed look at Nash Q-learning.

In state s, the learning agent chooses an action a from its action space A and transitions to state s'. He will receive a reward for that action. He will also keep track of the actions and rewards for the other players. Equation (3.3) describes the updating technique used by Nash-Q.

$$Q_{t+1}^{j}\left(s,a^{1},\cdots,a^{n}\right)=\left(1-\alpha_{t}\right)Q_{t}^{j}\left(s,a^{1},\cdots,a^{n}\right)+\alpha_{t}\left[r_{t}^{j}+\gamma NashQ_{t}^{j}\left(s'\right)\right] \quad (3.3)$$

Algorithm 2 Nash Q-Learning

Initialization

Let $t = 0$ and initial state s_0

Let each learning agent be indexed by i

For all $s \in S$ and $a^j \in A^j$ where $j = 1, \ldots, n$

Let $Q_t^j(s, a^1, \ldots, a^n) = 0$

Loop

Choose action a_t^i

Observe $r_t^1, \cdots, r_t^n$, and $s_{t+1} = s'$

Update Q_t^j for $j = 1, \cdots, n$

$Q_{t+1}^j(s, a^1, \cdots, a^n) = (1 - \alpha_t) Q_t^j(s, a^1, \cdots, a^n) + \alpha_t \left[r_t^j + \gamma NashQ_t^j(s')\right]$

Let $t = t + 1$

When the agent updates its Q-table, the agent needs to find the equilibrium strategy for state s' and the value of the expected payoff if the players follow that equilibrium from that point on, which is represented by $NashQ_t^j(s')$. The agent will build the matrix game for state s' with the reward matrix of both players as seen in Table 3.2. This is the same as in the last chapter for a matrix game where each player has a reward matrix depending on the actions of each player. We are using a two player game with two actions per player.

Player 1	$\begin{bmatrix} Q^1(s', 1, 1) & Q^1(s', 1, 2) \\ Q^1(s', 2, 1) & Q^1(s', 2, 2) \end{bmatrix}$
Player 2	$\begin{bmatrix} Q^2(s', 1, 1) & Q^2(s', 1, 2) \\ Q^2(s', 2, 1) & Q^2(s', 2, 2) \end{bmatrix}$

Table 3.2: Reward matrices for player 1 and 2 in state s'

The Nash Equilibrium will correspond to a joint action for player 1 and 2. The algorithm replaces the value of $NashQ_t^j(s')$ by the value of the Q-values corresponding to that joint action. The goal is to always give a higher valued reward to the joint action that represents the Nash equilibrium action.

One of the problems with the Nash equilibrium is that there might be more than one equilibrium. If two agents choose a different Nash equilibrium when learning, they will learn two different equilibrium strategies and may not be able to optimize

their reward. The authors used the Lemke-Howson algorithm [5] to find the Nash equilibrium of the two matrices. We will discuss the Lemke-Howson algorithm in the next section.

The convergence of this Nash Q is based on three important assumptions from [8]. We are including them here to discuss the effects and the limitations that they impose on the algorithm.

Assumption 1 [8] *Every state $s \in S$ and action $a^k \in A^k$ for $k = 1, \ldots, n$, are visited infinitely often.*

Assumption 2 [8] *The learning rate α_t satisfies the following conditions for all $s, t, a^1, \ldots, a^n$:*

1. $0 \leq \alpha_t(s, a^1, \ldots, a^n) < 1, \sum_{t=0}^{\infty} \alpha_t(s, a^1, \ldots, a^n) = \infty, \sum_{t=0}^{\infty} [\alpha_t(s, a^1, \ldots, a^n)]^2 < \infty$, *and the latter two hold uniformly and with probability 1.*

2. $\alpha_t(s, a^1, \ldots, a^n) = 0$ *if* $(s, a^1, \ldots, a^n) \neq (s_t, a^1, \ldots, a^n)$. *This means that the agent will only update the Q-values for the present state and actions. It does not need to update every value in the Q-tables at every step.*

Assumption 3 [8] One of the following conditions holds during learning.

Condition A. *Every stage game $(Q_t^1(s), \ldots, Q_t^n(s))$, for all t and s, has a global optimal point, and agents' payoffs in this equilibrium are used to update their Q-functions.*

Condition B. *Every stage game $(Q_t^1(s), \ldots, Q_t^n(s))$, for all t and s, has a saddle point, and agents' payoffs in this equilibrium are used to update their Q-functions.*

In a matrix game, we consider that a joint strategy is a global optimal point if all the players receive their highest payoff in that particular state [8]. It can also be called a coordination equilibrium [11]. We consider that a joint strategy is a saddle point if it is a Nash Equilibrium and each agent would receive a higher payoff when

Example 1	Up	Left
Up	**10,8**	-1,3
Right	4,1	-2,-4

a) Global optimal point

Example 2	Up	Left
Up	**4,4**	-1,-5
Right	-5,-2	**2,2**

b) Saddle point

Example 3	Up	Left
Up	**10,8**	-1,3
Right	4,1	**2,2**

c) Both

Table 3.3: Examples of bimatrix games for different Nash equilibriums

the other players chose a different strategy [8]. If one of the players chose a different action, it will hurt him and help all the other players [11]. The saddle points are also called adversarial equilibrium [11]. We have an example of each in Table 3.3.

The Nash equilibrium actions of the three examples in Table 3.3 are $\langle up, up \rangle$ for Example 1, $\langle right, left \rangle$ for Example 2 and $\langle up, left \rangle$ for Example 3. In Example 3 there is another Nash equilibrium: $\langle right, left \rangle$. For this example, Player 1 is the row player and Player 2 is the column player. It is easy to see that the best choice for both players in Example 1 is $\langle up, up \rangle$ because if one of the player do not choose that joint action they will receive a lower reward. In Example 2, we have a saddle point and it is not so easy to determine which of the two possible joint actions is the Nash equilibrium. If Player 1 chooses action *up*, Player 2 will have to choose action *up* to maximize its payoff. The same apply if Player 2 chooses action *left*. In the Nash-Q algorithm, the Lemke-Howson algorithm is used to find the Nash equilibrium in a bimatrix game. This algorithm can find more than one Nash equilibrium and that is how we can find more than one Nash equilibrium in the second and third example. The Nash-Q algorithm assumption stipulates that if a state has either a global optimal point or a saddle point, the learner will converge to a Nash equilibrium strategy. Our third example shows that there is a chance that more than one Nash equilibrium are present. What does the agent do in that situation? It all depends on the algorithm used to find the Nash equilibriums. In this situation, the Lemke-Howson would choose $\langle up, up \rangle$ as the first Nash and $\langle right, left \rangle$ as the second Nash.

The different assumptions that we reviewed make Nash-Q learning a restrictive

algorithm. It was proven to converge within these assumptions, but it cannot be generalized for every general-sum game. In [11], Littman discussed the limitations of the Nash-Q Learning algorithm. The third assumption implies that the updates will take into account either the global optimal point or the saddle point, but as we pointed out earlier, there are occasions where both are present. It was shown in [8] and [12] that the algorithm does converge even if not all the assumptions are respected. This could mean that the algorithm might be worth investigating in the right environment.

3.3.1 Lemke-Howson algorithm

We will examine the Lemke-Howson algorithm because it was one of the main issues we had when implementing the Nash-Q algorithm. It is an algorithm to find Nash equilibrium in a two player game with a finite number of actions. It appeared for the first time in [13] in 1964. This method can be compared to the simplex method because it uses the same pivoting method to solve the linear equations.

We will repeat the notation that was used in the matrix games section to help in the explanation. We will define two $m \times n$ matrices referred as A and B. In both matrices, player 1 is the row player and player 2 is the column player. Let player 1 have m actions given by the set $M = \{1, \cdots, m\}$ and player 2 has n actions given by the set $N = \{m+1, \cdots, m+n\}$. Each value of matrix A is represented by a_{ij} and the same goes for B with b_{ij}, where i is the index of the action taken by player 1 and j is the index of the action taken by player 2. If player 1 chooses action i and player 2 chooses action j, then the expected reward for each player is a_{ij} and b_{ij} respectively [5]. Each possible action for player 1 and player 2 are represented as rows and columns respectively in the matrices. The policy x_i is the probability of player 1 to choose action i, where $i = 1, \cdots, m$ and $\sum_{i=1}^{m} x_i = 1$. The policy y_j is the probability of player 2 to choose action j, where $j = 1, \cdots, n$ and $\sum_{j=1}^{n} y_j = 1$.

Equations (3.4) represent the expected payoff for player 1 and player 2 respectively. The symbol x will represent the set of all the mixed-strategies for player 1 and y for player 2, i.e. $x_i \in X$ and $y_j \in Y$. It is assumed that all the values of A and B are positive, that A does not have any all-zero columns and that B does not have any all-zero rows [13]. It does not affect the generality of the algorithm because the addition of a large positive value to each element of A or B does not change the Nash equilibrium.

$$\sum_{i=1}^{m}\sum_{j=1}^{n} x_i a_{i,j} y_j \quad \text{and} \quad \sum_{i=1}^{m}\sum_{j=1}^{n} x_i b_{i,j} y_j \tag{3.4}$$

To understand this method, we need to describe what is a polytope. In linear programming, we call a polytope the feasible area delimited by the linear equalities and inequalities [14] [15] [16]. Let B_j denote the column of B corresponding to action j, and let Ai denote the row of A^i corresponding to action i. We define the polytopes as :

$$P_1 = \left\{x \in \mathbb{R}^M | (\forall i \in M : x_i \geq 0) \,\&\, \left(\forall j \in N : x^T B_j \leq 1\right)\right\} \tag{3.5}$$

$$P_2 = \left\{y \in \mathbb{R}^N | (\forall i \in N : y_i \geq 0) \,\&\, \left(\forall i \in M : A^i y \leq 1\right)\right\} \tag{3.6}$$

For this method there is no restriction for x or y to be stochastic. The only constraint is to be positive. The stochastic answer for x can be calculate by normalization as per Equation (3.7) [14]. The same can be applied to y.

$$\text{nrml}\,(x) := \left(\sum_i x_i\right)^{-1} x \tag{3.7}$$

We will use an example to demonstrate the Lemke-Holson algorithm. We will use a tableau technique that is used in linear programming. It is a different version because we need two tableaux to calculate the Nash equilibrium.

Player 1	$\begin{bmatrix} 5 & 4 \\ 1 & 3 \end{bmatrix}$		Player 2	$\begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix}$

Table 3.4: Strictly Dominated Strategy for Player 1

Before building the tableau, we need to ensure that there are no strictly dominated strategies in the matrix game we are looking to solve. A strictly dominated strategy means that a player has a strategy where one action will always give more payoff than another. This means that if player 1 has two actions and action 1 gives him more rewards than action 2 he will always use action 1. We can see in Table 3.4 that player 1 has a dominated strategy compared to player 2. For any action of player 2, player 1 will choose action 1.

For the demonstration of the Lemke-Howson algorithm we will use the matrices shown in Table 3.5. Those two matrices are not strictly dominated because either player has an action that will always give them the maximum payoff.

Player 1:	$A = \begin{bmatrix} 2 & 5 \\ 4 & 3 \end{bmatrix}$		Player 2:	$B = \begin{bmatrix} 4 & 2 \\ 3 & 7 \end{bmatrix}$

Table 3.5: Payoff matrix for the Lemke-Howson example

The Lemke-Howson algorithm solves the system with the polytope from Equations (3.5) and (3.6). We will use a tableau method to solve the problem. It is similar to solving a linear equation with the Simplex algorithm. We start by finding the equations for the system. From the polytopes we define P_1 and P_2 by their limits: $A^i y \leq 1$, $x^T B_j \leq 1$, $x_i \geq 0$ and $y_j \geq 0$. Because we have inequalities, we need to add slack variables. These slack variables need to be added because we have inequalities. To solve a linear equation we transform the inequalities to equalities by adding a slack variable. That variable has to be positive. In this situation, $A_i y \leq 1$ becomes $Ay + r = \mathbf{1}$ by adding r_i as the slack variable and $x^T B_j \leq 1$ becomes $B^T x + s = \mathbf{1}$ by adding s_j. Our new system of equations is:

$$Ay + r = \mathbf{1},\ B^T x + s = \mathbf{1}, \text{ and where } x,\ y,\ r,\ s \text{ are non-negative.}$$

The basis in the initial tableaux is $\{r_i | i \in M\} \cup \{s_j | j \in N\}$ and we need to solve for them. The initial tableaux are $r = 1 - Ay$,

$$r_1 = 1 - 2y_3 - 5y_4 \tag{3.8}$$

$$r_2 = 1 - 4y_3 - 3y_4 \tag{3.9}$$

and $s = 1 - B^T x$,

$$s_3 = 1 - 4x_1 - 3x_2 \tag{3.10}$$

$$s_4 = 1 - 2x_1 - 7x_2 \tag{3.11}$$

We can draw the constraints on a graph to illustrate the polytopes by solving with the slack variables equal to zero. From this operation we get the equality constraint conditions for the polytope (x_1, x_2),

$$x_2 = \frac{1}{3} - \frac{4}{3}x_1 \tag{3.12}$$

$$x_2 = \frac{1}{7} - \frac{2}{7}x_1 \tag{3.13}$$

The same applies to the polytope (y_3, y_4),

$$y_4 = \frac{1}{5} - \frac{2}{5}y_3 \tag{3.14}$$

$$y_4 = \frac{1}{3} - \frac{4}{3}y_3 \tag{3.15}$$

We show polytope (x_1, x_2) in Figure 3.3.

At this point, we need to start solving the tableau by either starting with x or with y. For example, we will start with x_1. When using a tableau to solve linear programming we need to use a minimum ratio test. In the column for x_1, we will pivot the row with the lowest coefficient of x_1. In our example, we see that -4 is the

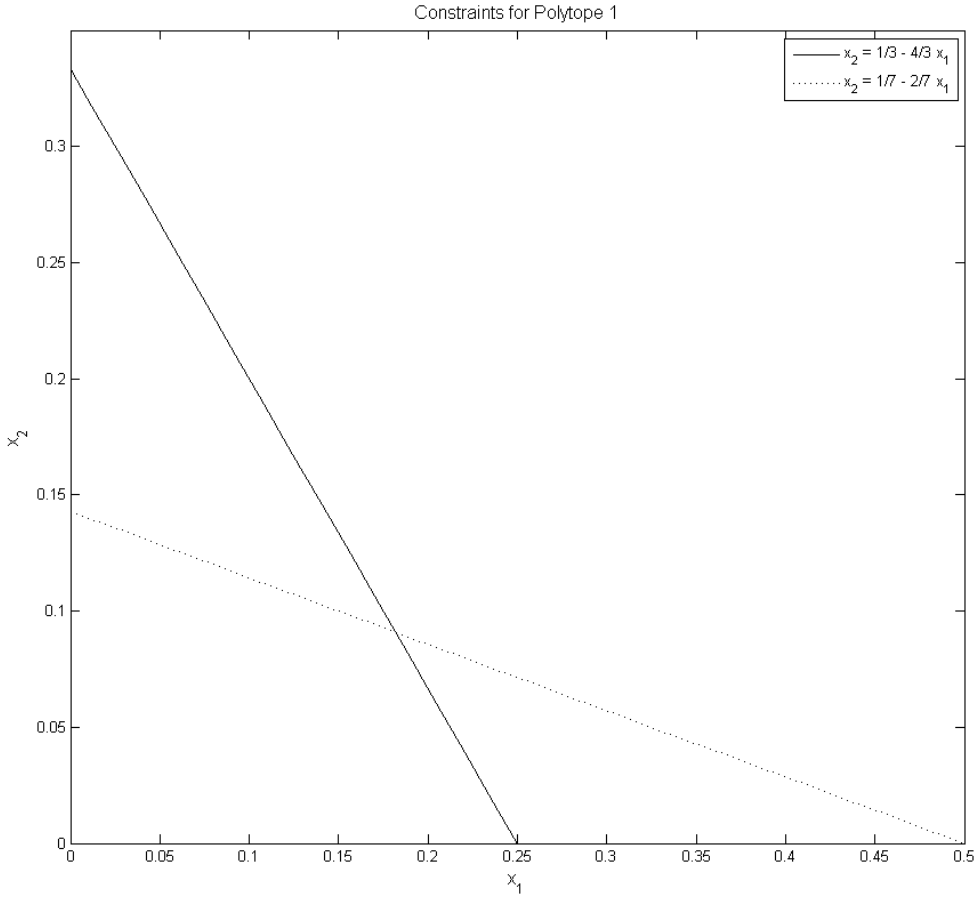


Figure 3.3: The polytope defined by P_1

lowest the coefficient which corresponds to the row of s_3. We will start by pivoting on $\langle x_1, s_3 \rangle$. Pivoting is used to find different basic feasible solutions that are adjacent. We are trying to reduce the size of the polytope region by changing the constraint from one feasible solution to another. The action of pivoting means that we choose a pivot element in the tableau. In this case we choose $\langle x_1, s_3 \rangle$. When we pivot, we find the equation for x_1 and replace it in the equation for s_4.

$$
\begin{aligned}
4x_1 &= 1 - s_3 - 3x_2 \\
x_1 &= \frac{1}{4} - \frac{1}{4}s_3 - \frac{3}{4}x_2 \qquad (3.16) \\
s_4 &= 1 - 2\left[\frac{1}{4} - \frac{1}{4}s_3 - \frac{3}{4}x_2\right] - 7x_2 \\
s_4 &= 1 - \frac{2}{4} + \frac{2}{4}s_3 + \frac{6}{4}x_2 - 7x_2 \\
s_4 &= \frac{1}{2} + \frac{1}{2}s_3 - \frac{11}{2}x_2 \qquad (3.17)
\end{aligned}
$$

We update the tableau with the new Equations (3.16) and (3.17):

$$
\begin{aligned}
x_1 &= \frac{1}{4} - \frac{1}{4}s_3 - \frac{3}{3}x_2 \\
s_4 &= \frac{1}{2} + \frac{1}{2}s_3 - \frac{11}{2}x_2
\end{aligned}
$$

As per [14], we know that the variable that we used in the first pivot determines the one that we will use in the next. The index of the slack variable determines which variable to use next. We used x_1 in the first pivot and it took the place of s_3. So for the next pivot, we start with y_3. The minimum ratio test gives us $\langle y_3, r_2 \rangle$ as the pivoting point because the coefficient -4 is smaller than -2. We find the equation for

y_3 and replace it in the equation of r_1.

$$
\begin{aligned}
4y_3 &= 1 - r_2 - 3y_4 \\
y_3 &= \frac{1}{4} - \frac{1}{4}r_2 - \frac{3}{4}y_4 \qquad (3.18) \\
r_1 &= 1 - 2\left[\frac{1}{4} - \frac{1}{4}r_2 - \frac{3}{4}y_4\right] - 5y_4 \\
r_1 &= 1 - \frac{2}{4} + \frac{2}{4}r_2 + \frac{6}{4}y_4 - 5y_4 \\
r_1 &= \frac{1}{2} + \frac{1}{2}r_2 - \frac{7}{2}y_4 \qquad (3.19)
\end{aligned}
$$

We update the tableau with the new Equations (3.18) and (3.19):

$$
\begin{aligned}
r_1 &= \frac{1}{2} + \frac{1}{2}r_2 - \frac{7}{2}y_4 \\
y_3 &= \frac{1}{4} - \frac{1}{4}r_2 - \frac{3}{4}y_4
\end{aligned}
$$

In the next step, we start with x_2 because we had r_2 in the last pivot. The minimum ratio test gives us $\langle x_2, s_4 \rangle$ as the pivoting point. We find the equation for x_2 and replace it in the equation of x_1.

$$
\begin{aligned}
\frac{11}{2}x_2 &= \frac{1}{2} + \frac{1}{2}s_3 - s_4 \\
x_2 &= \frac{2}{22} + \frac{2}{22}s_3 - \frac{2}{11}s_4 \qquad (3.20) \\
x_1 &= \frac{1}{4} - \frac{1}{4}s_3 - \frac{3}{4}\left[\frac{2}{22} + \frac{2}{22}s_3 - \frac{2}{11}s_4\right] \\
x_1 &= \frac{2}{11} - \frac{7}{22}s_3 + \frac{3}{22}s_4 \qquad (3.21)
\end{aligned}
$$

We update the tableau with the new Equations (3.20) and (3.21):

$$x_1 = \frac{2}{11} - \frac{7}{22}s_3 + \frac{3}{22}s_4$$
$$x_2 = \frac{2}{22} + \frac{2}{22}s_3 - \frac{2}{11}s_4$$

In the next step, we start with y_4 because we had s_4 in the last pivot. The minimum ratio test gives us $\langle y_4, r_1 \rangle$ as the pivoting point. We find the equation for y_4 and replace it in the equation of y_3.

$$\frac{7}{2}y_4 = \frac{1}{2} - r_1 + \frac{1}{2}r_2$$
$$y_4 = \frac{1}{7} - \frac{2}{7}r_1 + \frac{1}{7}r_2 \tag{3.22}$$
$$y_3 = \frac{1}{4} - \frac{1}{4}r_2 - \frac{3}{4}\left[\frac{1}{7} - \frac{2}{7}r_1 + \frac{1}{7}r_2\right]$$
$$y_3 = \frac{1}{7} + \frac{3}{14}r_1 - \frac{5}{14}r_2 \tag{3.23}$$

We update the tableau with the new Equations (3.22) and (3.23):

$$y_4 = \frac{1}{7} - \frac{2}{7}r_1 + \frac{1}{7}r_2$$
$$y_3 = \frac{1}{7} + \frac{3}{14}r_1 - \frac{5}{14}r_2$$

At this point, we have reached the point where we are looking at x_1 again, which means that the algorithm is over and we reached an answer. The new constraint equations are in Table 3.3.1.

To get the value for each of the x and y, we need to set the value of the slack

$$y_4 = \frac{1}{7} - \frac{2}{7}r_1 + \frac{1}{7}r_2$$
$$y_3 = \frac{1}{7} + \frac{3}{14}r_1 - \frac{5}{14}r_2$$
$$x_1 = \frac{2}{11} - \frac{7}{22}s_3 + \frac{3}{22}s_4$$
$$x_2 = \frac{2}{22} + \frac{2}{22}s_3 - \frac{2}{11}s_4$$

Table 3.6: Modified Constraint Equations

variables to zero. We get the following values :

$$r = (0,0), s = (0,0), x = \left(\frac{2}{11}, \frac{2}{22}\right), y = \left(\frac{1}{7}, \frac{1}{7}\right)$$

To get the stochastic values of x and y, we need to perform a normalisation as per Equation (3.7).

$$norm(x_1) = \frac{\frac{2}{11}}{\frac{2}{11} + \frac{2}{22}} = \frac{2}{3} \quad \text{and} \quad norm(x_2) = \frac{\frac{2}{22}}{\frac{2}{11} + \frac{2}{22}} = \frac{1}{3}$$
$$norm(y_3) = \frac{\frac{1}{7}}{\frac{1}{7} + \frac{1}{7}} = \frac{1}{2} \quad \text{and} \quad norm(y_4) = \frac{\frac{1}{7}}{\frac{1}{7} + \frac{1}{7}} = \frac{1}{2}$$

$$(\text{nrml}(x), \text{nrml}(y)) = \left(\left(\frac{2}{3}, \frac{1}{3}\right), \left(\frac{1}{2}, \frac{1}{2}\right)\right)$$

To verify our answers, given that $x_0 = \begin{bmatrix} \frac{2}{3} \\ \frac{1}{3} \end{bmatrix}$ and $y_0 = \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \end{bmatrix}$ then the payoff for to

player 1 is,

$$R_1 = x_0^T A y_0$$
$$= \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \end{bmatrix} \begin{bmatrix} 2 & 5 \\ 4 & 3 \end{bmatrix} \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \end{bmatrix} = 3.5$$

We do the same with player 2,

$$R_2 = x_0^T B y_0$$
$$= \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} 4 & 2 \\ 3 & 7 \end{bmatrix} \begin{bmatrix} \frac{2}{3} \\ \frac{1}{3} \end{bmatrix} = 3.833$$

We can see that the algorithm finds a Nash Equilibrium. The algorithm can find more than one Nash Equilibrium (if there exists more than one) by choosing a different starting variable. In this example, we started with x_1, but we could have started with x_2, y_3 or y_4. In Nash Q-learning, we get up to four different Nash equilibriums and we can choose which one we use.

3.4 Friend-or-Foe Q-Learning Algorithm

This algorithm was developed by Littman and tries to fix some of the convergence problems of Nash-Q Learning. As mentioned in section 3.3, the convergence of Nash-Q is restricted by assumptions 1, 2 and 3. The main concern lies within assumption 3, where every stage game needs to have either a global optimal point or a saddle point. These restrictions cannot be guaranteed during learning. To alleviate this restriction, this new algorithm is built to always converge by changing the update rules depending on the opponent. The learning agent has to identify the other agent as "friend" or "foe". That is the reason behind the name Friend-or-Foe Q-Learning

(FFQ). The author of the algorithm believes that this is an improvement over Nash-Q but that the application for general-sum game is still incomplete [17].

The FFQ is an adaptation of the Nash-Q algorithm that we described earlier in the text. The algorithm is built for the n-player game, but we will start with a two player game to understand this concept. One of the main differences between the Nash-Q and the FFQ is that the agent only keeps track of its own Q-table. From the equation of Nash-Q, Littman adapted the update performed by the agent in Equation (3.3) by replacing the state value $NashQ^i\left(s, Q_1, \ldots, Q_n\right)$ by

$$\max_{a_i \in A_i} Q_i\left[s, a_1, \ldots, a_n\right] \tag{3.24}$$

when the opponents are friends; and

$$\max_{\pi \in \Pi(A_i)} \min_{a_i \in A_i} \sum_{a_i \in A_i} \pi\left(a_i\right) Q_i\left[s, a_1, \ldots, a_n\right] \tag{3.25}$$

when the opponents are foes, where n is the number of agents and i is the learning agent [17].

Equation (3.24) is the Q-Learning algorithm adapted for multiple agents and Equation (3.25) is the minimax-Q algorithm from Littman [9]. These equations represent a situation where all the agents are either friend or foe. As defined by Littman [17], $NashQ_n\left(s, Q_1, \ldots, Q_n\right)$ for n-player games where X_1 through X_k are the actions available for the k friends of player i and Y_1 through Y_l are the actions available to the l foes is

$$NashQ_i\left(s, Q_1, \ldots, Q_n\right) =$$

$$\max_{\pi \in \Pi(X_1 \times \ldots \times X_k)} \min_{y_1, \ldots, y_l \in Y_1, \ldots, Y_l} \sum_{x_1, \ldots, x_k \in X_1, \ldots, X_k} \pi\left(x_1\right) \cdots \pi\left(x_k\right) Q_i\left[s, x_{1,\ldots,x_k, y_1, \ldots, y_l}\right] \tag{3.26}$$

which replace $NashQ^i$ in Algorithm 3. We can describe this as two groups of people:

i's friends and i's foes. The friends will work together to maximize i's payoff. The foes will work together against i to minimize its payoff.

Algorithm 3 Friend-or-Foe Q-Learning

Initialization

$\forall\ s \in S$, $a_1 \in A_1$ and $a_2 \in A_2$

Let $Q(s, a_1, a_2) = 0$

$\forall\ s \in S$

Let $V(s) = 0$

$\forall\ s \in S, a_1 \in A_1$

Let $\pi(s, a_1) = 1/|A_1|$

Let $\alpha = 1$

Loop

In state s

Choose a random action from A_1 with probability ε

If not a random action, choose action a_1 with probability $\pi(s, a_1)$

Learning

In state s'

The agent observes the reward R related to action a_1 and opponent's action a_2 in state s

Update Q-Table of player 1 with equation

$Q(s, a_1, a_2) = (1 - \alpha) * Q(s, a_1, a_2) + \alpha(R + \gamma * V[s'])$

Use linear programming to find the values of $\pi(s, a_1)$ and $V(s)$ with the equation

$V(s) = \max_{a_1 \in A_1, a_2 \in A_2} Q_1[s, a_1, a_2]$ if the opponent is a friend, and;

$V(s) = \max_{\pi \in \Pi(A_1)} \min_{a_2 \in A_2} \sum_{a_1 \in A_1} \pi(a_1) Q_1[s, a_1, a_2]$ if the opponent is a foe.

$\alpha = \alpha$ * decay

End Loop

FFQ has been designed to alleviate the flaws of Nash-Q Learning when confronted with the coordination and adversarial equilibrium. Littman [11] discussed this by pointing out that there is a possibility that both equilibriums exist at the same time. This can create problems in Nash-Q because it is not designed to decide which one to choose. In FFQ there is a selection mechanism and it can alleviate this problem.

3.5 Win or Learn Fast-Policy Hill Climbing (WOLF-PHC) Learning Algorithm

All the algorithms that were discussed earlier are adaptations of the single agent Q-learning algorithm for the multiagent environment. The Minimax-Q algorithm approaches the problem by maximizing its rewards by choosing the action that would give the best reward when considering the opponent's actions. The Nash-Q algorithm tries to find the Nash equilibrium strategy by calculating the Nash equilibrium in each state. Finally, the Friend-or-Foe-Q algorithm takes from both Minimax-Q and Nash-Q and creates an algorithm that uses a different updating method depending on the opponent (friend or foe). During our research we saw many more reinforcement learning algorithms that could be useful for our experiments. We decided to implement the WOLF-PHC algorithm because it presents another technique which is based on gradient ascent to update its policy. A gradient ascent algorithm will start with a suboptimal solution and will improve over time by changing its policy by small increments. As an example, we can compare with the Q-learning algorithm. The Q-learning algorithm always chooses the action that has the highest payoff when it is not exploring. It is a good way to maximize your payoff. However, the agent might get stuck in a suboptimal policy until he explores enough to get him to choose a different action. For example, let us assume that the agent Q-values are all zero. In the initial state of the game, the agent chooses action a, which gives it a reward of 1. The next time the agent is in the same state it will choose the same action again unless it tries to explore even if action b would give him a reward of 5. By using a gradient algorithm, the agent modifies its policy by small increments. In our example, the agent would have changed its probability for action a by δ. If $\delta < 1$ then the probability of action b of the agent does not become zero when updating its policy. If δ is small, the agent will update its policy by small increments which means

$$\text{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \Bigg\| \text{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Table 3.7: Player 1 and Player 2 payoff matrices

it will have less chance of getting into a suboptimal strategy.

To understand the concept of gradient ascent, we will explain the method in more detail. Lets take a 2 player general-sum game where each player can choose between two actions. We will use the definition of the matrix game that we used in Chapter 2 Section 5. We define two 2x2 matrices referred as A and B. In both matrices Player 1 is the row player and Player 2 is the column player. Each value of matrix A is represented by a_{ij} and the same goes for B with b_{ij}, where i is the index of the action taken by Player 1 and j is the index of the action taken by Player 2. If Player 1 chooses action i and Player 2 chooses action j, then the expected reward for each player is a_{ij} and b_{ij} respectively [5]. Each possible action for Player 1 and Player 2 are represented as rows and columns respectively in the matrices. Player 1 and Player 2 follow mixed strategies and can choose each action stochastically. In our example we will assume that α represent the probability that Player 1 chooses action 1 and $(1-\alpha)$ is the probability that Player 1 chooses action 2. The same goes for Player 2 policies where β is the probability that he chooses action 1 and $(1-\beta)$ is the probability that he chooses action 2. The players payoff matrices are shown in Table 3.7 where A is Player 1's payoff matrix and B is Player 2's payoff matrix.

We can now write Player 1 and Player 2 expected payoff for strategy (α, β) as per Equations (3.27) and (3.28).

$$V_a(\alpha,\beta) = a_{11}(\alpha\beta) + a_{22}((1-\alpha)(1-\beta)) + a_{12}(\alpha(1-\beta)) + a_{21}((1-\alpha)\beta) \quad (3.27)$$

$$V_b(\alpha,\beta) = b_{11}(\alpha\beta) + b_{22}((1-\alpha)(1-\beta)) + b_{12}(\alpha(1-\beta)) + b_{21}((1-\alpha)\beta) \quad (3.28)$$

When the players change policy it affects their respective payoff. We can calculate that effect by finding the partial derivative of its expected payoff with respect to its mixed strategy. We find the following values for each partial derivative:

$$\frac{\partial V_a(\alpha,\beta)}{\partial \alpha} = \beta u - (a_{22} - a_{12}) \tag{3.29}$$

where $u = (a_{11} + a_{22}) - (a_{21} + a_{12})$ and

$$\frac{\partial V_b(\alpha,\beta)}{\partial \alpha} = \beta \acute{u} - (b_{22} - b_{12}) \tag{3.30}$$

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

In a gradient ascent algorithm, each player adjusts their strategy in the direction of their gradient with a step size η as to maximize their payoff. They do it at every time step as per the following updating equations:

$$\alpha_{k+1} = \alpha_k + \eta \frac{\partial V_A(\alpha_k,\beta_k)}{\partial \alpha_k} \tag{3.31}$$

and

$$\beta_{k+1} = \beta_k + \eta \frac{\partial V_B(\alpha_k,\beta_k)}{\partial \beta_k}. \tag{3.32}$$

The time step is assumed to be $\eta \in [0,1]$. We will see later how the gradient ascent method is used by the WOLF-PHC algorithm.

The reasons behind the implementation of this algorithm was to have a learning algorithm that would be rational and that would converge. We can define rationality by stating that if the other players' strategies converge to stationary policies then the learning algorithm will converge to a policy that is the best response to their policies [18]. An agent is rational if it can find the best response policies against an opponent that has an unchanging policy over time. This property is the basis for many algorithms. The goal of a learning agent is often to maximize its own reward while

minimizing its opponent's. The other property is convergence to a stationary policy. We can see the link between the two properties. A rational agent will converge to a best response policy and if that policy becomes stationary then we have convergence. The authors of WOLF-PHC wanted to design an algorithm that would meet both properties. The focus of their research was for the self-play environment. They only tested their algorithm against itself. With this assumption, we can see that if they have a learning agent that is rational and converges to a stationary policy then they should have converged to an optimal policy which is the Nash equilibrium.

The WOLF-PHC learning algorithm was developed by Bowling and Veloso [18] in 2001. It is a combination of the Policy Hill Climbing (PHC) algorithm and the Win or Learn Fast (WOLF) algorithm. The PHC algorithm is a Q-Learning algorithm with a different method of updating its policy. In PHC there is a second learning rate $\delta \in [0, 1]$ which affects the probability of the highest value action [18]. After updating the Q-table, the algorithm checks which action would provide the highest reward and adds δ to that action probability. It is followed by a rectification of the policy's value to ensure a legal probability distribution. If $\delta = 0$, the algorithm is the same as Q-Learning and chooses the highest value action at every move. From Bowling's conclusion, PHC is rational and will converge to an optimal policy in situations where the opponents play stationary strategies [18].

The WOLF-PHC learning algorithm is an adaptation from the PHC algorithm. As the name points out, the algorithm will change the learning rate depending on the situation. The agent will learn fast when loosing and will learn slowly when winning. We define winning by comparing the agent's current policy's expected payoff with the expected payoff of the average policy over time [18].

We can adapt the gradient ascent equations in (3.31) and (3.32) for the WOLF-PHC by adding a learning rate l that will change depending on if the player is winning or losing. We get the following updating equations for Player 1:

$$\alpha_{k+1} = \alpha_k + \eta\, l_k^a\, \frac{\partial V_a\left(\alpha_k, \beta_k\right)}{\partial \alpha_k} \tag{3.33}$$

where

$$l_k^a = \begin{cases} l_{min} & \text{if } V_a\left(\alpha_k, \beta_k\right) > V_a\left(\alpha^*, \beta_k\right) \quad \text{Winning} \\ l_{max} & \text{Otherwise} \qquad\qquad\qquad\qquad\quad \text{Losing} \end{cases} \tag{3.34}$$

and the following updating equation for Player 2:

$$\beta_{k+1} = \beta_k + \eta\, l_k^b\, \frac{\partial V_b\left(\alpha_k, \beta_k\right)}{\partial \beta_k} \tag{3.35}$$

where

$$l_k^b = \begin{cases} l_{min} & \text{if } V_b\left(\alpha_k, \beta_k\right) > V_b\left(\alpha_k, \beta^*\right) \quad \text{Winning} \\ l_{max} & \text{Otherwise} \qquad\qquad\qquad\qquad\quad \text{Losing} \end{cases} \tag{3.36}$$

$V_a\left(\alpha^*, \beta_k\right)$ and $V_b\left(\alpha_k, \beta^*\right)$ are the expected payoff with the average policy over time. The WOLF-PHC algorithm used the theory of the gradient ascent when updating its policy. We will examine in more detail the algorithm that was designed by Bowling and Veloso [18].

Algorithm 4 Win or Learn Fast-Policy Hill Climbing

Initialization

Let $s \in S$ and $a \in A_i$ for player i

Let α, δ_l and δ_w be learning rates, where $\delta_l > \delta_w$.

Let $Q(s,a) \leftarrow 0, \quad \pi(s,a) \leftarrow \frac{1}{|A_i|}, \quad C(s) \leftarrow 0.$

Loop

Choose a random action from A with probability ε

If not a random action, choose action a with probability $\pi(s,a)$

Update Q-tables

$Q(s,a) \leftarrow (1-\alpha) Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') \right)$

Update estimate of average policy, $\bar{\pi}$,

$C(s) \leftarrow C(s) + 1$

$\forall a' \in A_i, \quad \bar{\pi}(s,a') \leftarrow \bar{\pi}(s,a') + \frac{1}{C(s)} \left(\pi(s,a') - \bar{\pi}(s,a') \right).$

Update $\pi(s,a)$ and constraint it to legal probability distribution,

$$\pi(s,a) \leftarrow \pi(s,a) + \begin{cases} \delta & a = \operatorname{argmax}_{a'} Q(s,a') \\ \frac{-\delta}{|A_i|-1} & \text{otherwise} \end{cases},$$

where,

$$\delta = \begin{cases} \delta_w & \text{if } \sum_a \pi(s,a) Q(s,a) > \sum_a \bar{\pi}(s,a) Q(s,a) \\ \delta_l & \text{otherwise} \end{cases}.$$

Algorithm 4 needs three different learning rates: α, δ_l and δ_w, where $\delta_l > \delta_w$. The Q-value updates as per usual with learning rate α. The policy is updated using δ_l or δ_w, when the agent is losing or winning, respectively. As we can see in Algorithm 4, the δ used to update the policy is determined by the following conditions:

$$\sum_a \pi(s,a) Q(s,a) > \sum_a \bar{\pi}(s,a) Q(s,a) \tag{3.37}$$

where $\pi(s,a)$ is the current policy and $\bar{\pi}(s,a)$ the average policy over time in state s. The agent chooses δ as per Equation (3.38).

$$\delta = \begin{cases} \delta_w & \text{if } \sum_a \pi(s,a) Q(s,a) > \sum_a \bar{\pi}(s,a) Q(s,a) \\ \delta_l & \text{otherwise} \end{cases} \tag{3.38}$$

In every state the agent updates its average policy $\bar{\pi}$ by comparing it with the current

one π as per Equation (3.39).

$$\bar{\pi}(s, a') \leftarrow \bar{\pi}(s, a') + \frac{1}{C(s)} (\pi(s, a') - \bar{\pi}(s, a')) \tag{3.39}$$

where $C(s)$ is a state counter, $\bar{\pi}(s, a')$ is the average policy over time and $\pi(s, a')$ is the current policy. The algorithm determines that the agent is winning when the expected payoff of the current policy is bigger than the average one (Equation (3.37)). The policy is then updated by the correct δ and constrained to the normal probability distribution $\pi \in [0, 1]$.

The authors Bowling and Veloso showed by empirical methods that the new algorithm is rational and converges in stochastic games. The values of the learning rate can be changed to modify the behaviour of the agent. By changing the ratio δ_l/δ_w, the agent will go from a more conservative player (low ratio) to a more aggressive player (high ratio).

We will show the different simulations that we did with the Minimax-Q algorithm, the Nash-Q algorithm, and the WOLF-PHC algorithm in the next chapter. We will discuss the differences and the similarities between each of them.

Chapter 4

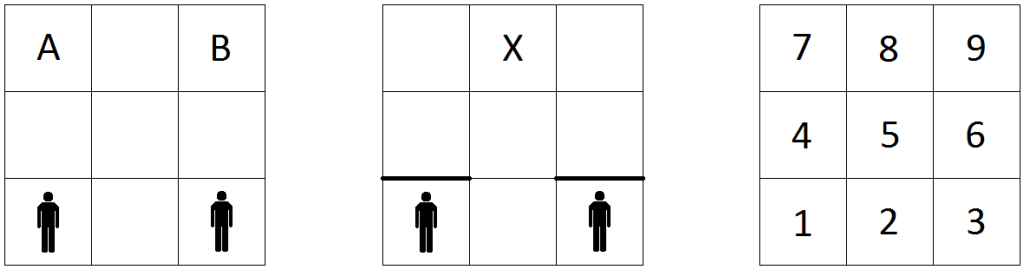
Experimental Results

4.1 Introduction

In this chapter we will discuss the results of our simulation of three of the algorithms from the previous chapter: Minimax-Q, Nash-Q and WOLF-PHC. In Section 4.2, we will describe each of the games we used to test the algorithms, followed in section 4.3 by a detailed explanation of the implementation for each algorithm which includes a performance test. Section 4.4 is a comparison between each algorithm. Section 4.5 is a comparison and discussion of the different learning rates used in our simulations. We finish in Section 4.6 with a simulation of the WOLF-PHC algorithm in a larger grid game to test its ability to adapt to different environment.

4.2 Experimental Games

Reinforcement learning commonly uses games to test various algorithms. We use two kinds of games in this thesis: matrix games and stochastic grid games. The principle behind these games has been explained in Chapter 2. We choose the following games because they represent the benchmark of the learning algorithm that we studied. They are often referred in the literature and accepted as valid test games.



a) Grid game 1 b) Grid game 2 c) Cells numbers

Figure 4.1: Grid Game 1 and 2 starting positions

We used only one matrix game called the Matching Pennies game. The Matching Pennies game is a simple two player zero-sum matrix game. Each player shows one side of a penny: heads or tails. Player 1 wins if both players show the same side. Player 2 wins if both players show a different side. The reward is 1 for the winner and -1 for the loser. Table 2.2 in Chapter 2 Section 5 represents the reward function for the game.

We are also using the grid games from [10] [12] [8]. They are called Grid Game 1 and Grid Game 2. They are two player general-sum stochastic games which means, as we discussed earlier, that each state s of the games is a matrix game. Figure 4.1 a) and b) show the different starting positions. We identified each cell in the game as per Figure 4.1 c).

The first grid game is a 3x3 grid game with two players and two goals as per Figure 4.1 a). Each cell is assigned a number from one to nine. The starting positions for player 1 and player 2 are the bottom left (cell 1) and bottom right (cell 3) corner of the grid, respectively. The goal for each player is in the opposite corner of the grid [top right (cell 9) and left (cell 7) corner]. Each player has four available actions: Up, Down, Left and Right. The game has been programmed in a way that the players will not try to get out of the grid. This reduces the number of state-actions in the game and it follows the method from [8]. Two players that try to move into the same grid

will be bounced back unless it is a goal for one of the players. When bounced back from this action, the players will receive a negative reward. The game ends when either one or both players reach their goal and a positive reward is given. The goal of the game is to minimize the number of moves each player makes before reaching their goal.

The reward function for this game is as follow:

1. When the two players try to move into the same cell they both receive a reward of -1.

2. When one or both players move into their goal cell, they receive a reward of 100.

3. When both players try to move into the goal cell of one of the players, the players are not bounced back; The game ends and the player who is in their goal cell receives a reward of 100.

The second grid game from [8] has the same dimensions and the same rules except that there is only one goal for both players. Also, as per Figure 4.1 b), the players have to go through a barrier when moving up from their starting position. That barrier has a 50% chance of blocking the movement. The reward function stays the same as per the previous game. Also, if both players reach the goal at the same time, they both get the goal reward. We used the same technique as the previous game and in [8]. This is off-line training with random start positions.

We recreated the same learning technique as per [8]. Hu and Wellman used off-line learning with Nash-Q learning. When reviewing the Nash-Q algorithm theory and results from [12], we saw that the author was using the term offline and online learning. We originally thought it meant that the agent had to learn on its own before playing against another player but it only meant that the agent used an explore-only technique. In reinforcement learning, it is accepted that a good exploit-explore

technique is necessary to maximize the expected payoff of a learning agent. In this case, the algorithm has to explore all the time to ensure that the policy found is the Nash strategy. We will discuss this later in the chapter during the Nash-Q algorithm implementation. For the purpose of this thesis, we will not use the online or offline term but refer to an explore-only technique. This brings us to the other type of learning techniques: exploit-only and exploit-explore. When an agent has to choose its next action, it can either chose its most rewarding action, or a random action, or a combination of the two. An agent that choses to exploit-only will look at which actions will provide the best reward. It is also called a greedy policy. It is used to maximize the reward to the agent, but it does not mean that the agent is following its optimal policy. The explore-only option is used to try all the actions randomly and to gather information to find the optimal policy. It will not maximize the rewards but the agent may find the optimal policy. The last type is a mix of both exploit-only and explore-only. The agent will choose the best action with $(1 - \varepsilon)$ probability and explore with ε probability. The exploit-explore technique is recognized to be very effective in reinforcement learning. The agent will maximize its rewards while still searching for the optimal strategy.

We used a standard method for learning during our experimentation. We have three different terms related to the game play: step, episode, and game. A step consists of each agent choosing an action, receiving the reward and updating their Q-tables. An episode represents the number of steps where the agents go from the starting position to the goal position. A game consists of a predetermined number of episodes. In our experimentation we use 5000 episodes as our benchmark.

4.3 Algorithm Implementation

4.3.1 Minimax-Q Implementation

We implemented Minimax-Q with Matlab and tested it on a simple two action game called the Matching Pennies game. In this matrix game we only used the Minimax-Q algorithm to demonstrate a simple matrix game. We wanted to test the algorithm with a simple game to understand the details of the Minimax-Q algorithm. We tested the effect of the exploration variable ε. We ran the same game with three different values for ε : 0.1 , 0.2 and 0.3. We also changed the value of the learning rate. We tested learning rate values of $\alpha = 0.1$, 0.5 and 0.9. The learning rates did not change during the tests. We kept the number of games the same for all tests to compare how each value affects the learning. The test was done over 1000 games. We classified the results by how close to the perfect Nash Equilibrium each player's policy compared and by how many games each needed to reach an equilibrium. The expected policy value for each player action is 0.5 which means that the Nash equilibrium is to choose each action with a probability of 50%. Figure 4.2 shows the results for each of the learning rates α with an ε of 0.1. The theory [3] says that the lower learning rate should learn slower than the higher ones. Figure 4.2 c) shows that the learning rate of 0.9 does reach a policy very close to the optimal one during learning. Comparatively, the other two did not finish learning with policies close to the expected results. In Figure 4.2 a), we can see the effect of a smaller learning rate by the small changes to the policy compared to higher learning rates where the changes are bigger. We can also see the effect of a small exploration variable in Figure 4.2 b) and c) where it takes longer to try random actions and change the policy values.

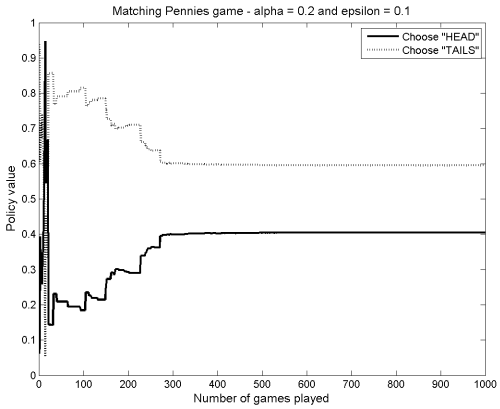
In Figure 4.3, we have the same learning rates but with a higher explore variable ε of 0.2. Two out of the three reached the optimal policy of 0.5 for both actions. The effect of the learning rates are similar to the last three where the lower learning rates

affect the policy by smaller steps than the higher ones. The different values of the explore variable made a significant effect to the learning because it allowed the agent to learn the optimal policy within the number of games allowed.

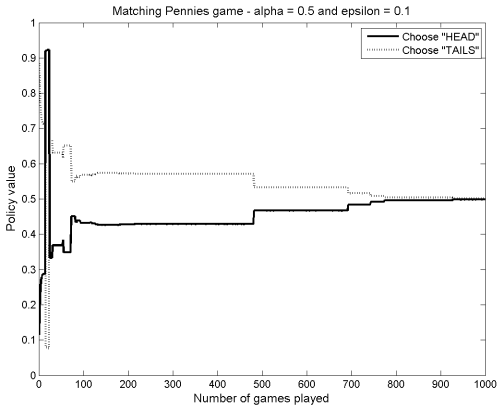
Finally, the results in Figure 4.4 represent the same learning rate but with an even higher explore variable $\varepsilon = 0.3$. We can see that the explore variable affects the learning greatly. The explore variable gives the learner a way to try actions that do not maximize its rewards at that time, but could in the future. We will see in more detail in Chapter 4 Section 4 why the learning rate is very important when implementing a learning algorithm.

Littman proved that the Minimax-Q algorithm converges in a multiagent environment [9] but only for zero-sum games. He also proved that the Friend-or-Foe Q-learning algorithm converges to a solution in a general-sum stochastic game but it does not always converge to a Nash equilibrium strategy. Foe-Q which we discussed earlier is the same as Minimax-Q and it means that the Minimax-Q algorithm does converge in some general-sum games but only when playing against foes. We wanted to simulate the algorithm in a general-sum environment. We used Grid Game 1 as the environment for this simulation. The Minimax-Q algorithm parameters were initialized as per [9]: $\alpha = 1$, $\varepsilon = 0.2$, $\gamma = 0.9$ and the number of episodes was set to 5000. The learning rate will diminish over time as per a decay variable $decay = 0.99862$ to reduce α to 0.001 after 5000 episodes.

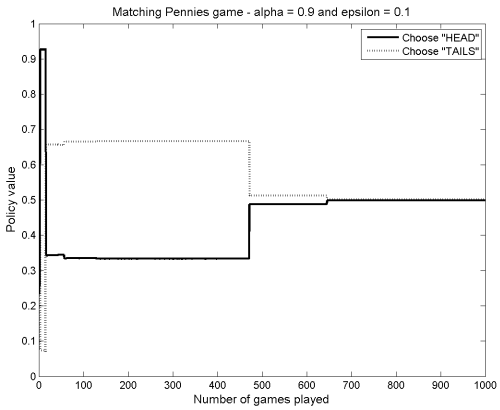
Each of the performance graph that we use in this thesis represent the cumulative average rewards divided by the number of steps that the agent perform. In each step, we divide the total amount of reward received by the total number of steps. We also calculate the cumulative reward over the number of episodes. At the end of the episode, we divide the total amount of reward received by the number of episodes. To ensure smooth graphs, we simulate ten games and average the cumulative average reward per step of the ten games.



a) $\alpha = 0.2$

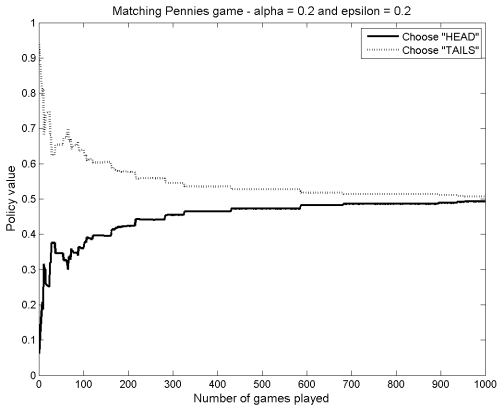


b) $\alpha = 0.5$

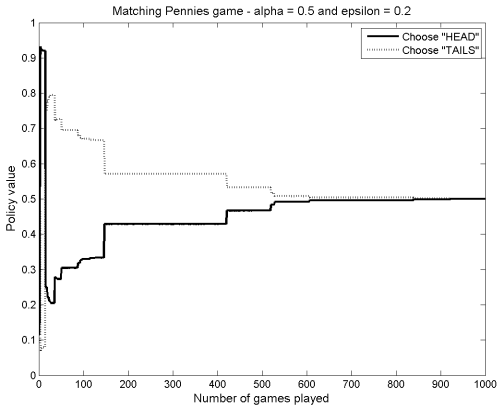


c) $\alpha = 0.9$

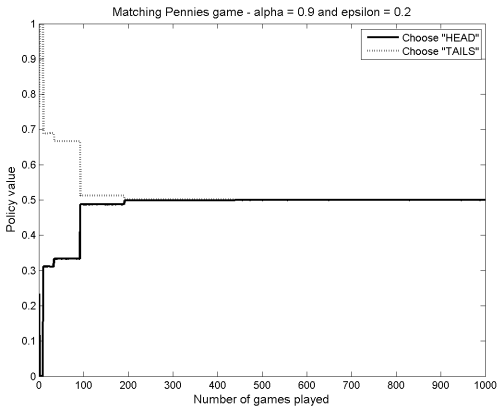
Figure 4.2: Minimax-Q Learning for Matching Pennies Game with $\varepsilon = 0.1$



a) $\alpha = 0.2$

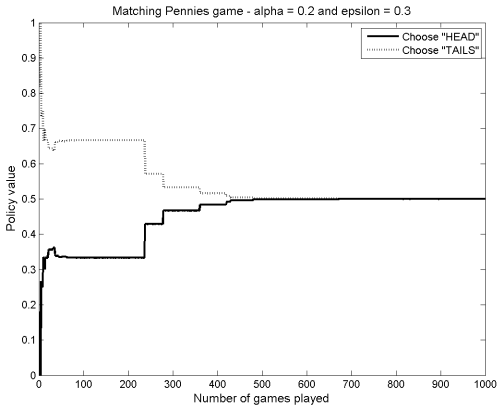


b) $\alpha = 0.5$

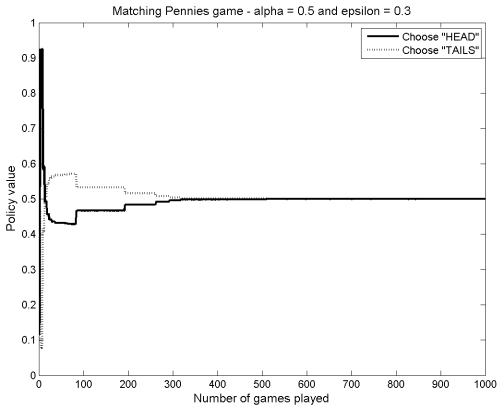


c) $\alpha = 0.9$

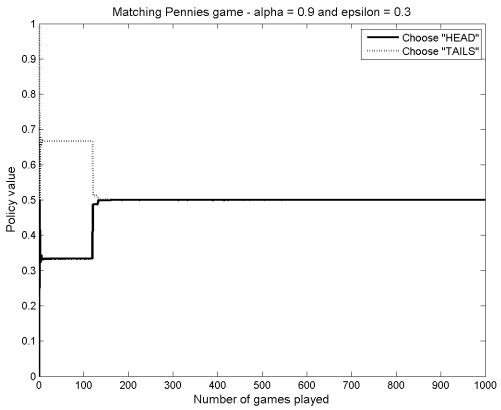
Figure 4.3: Minimax-Q Learning for Matching Pennies Game with $\varepsilon = 0.2$



a) $\alpha = 0.2$



b) $\alpha = 0.5$



c) $\alpha = 0.9$

Figure 4.4: Minimax-Q Learning for Matching Pennies Game with $\varepsilon = 0.3$

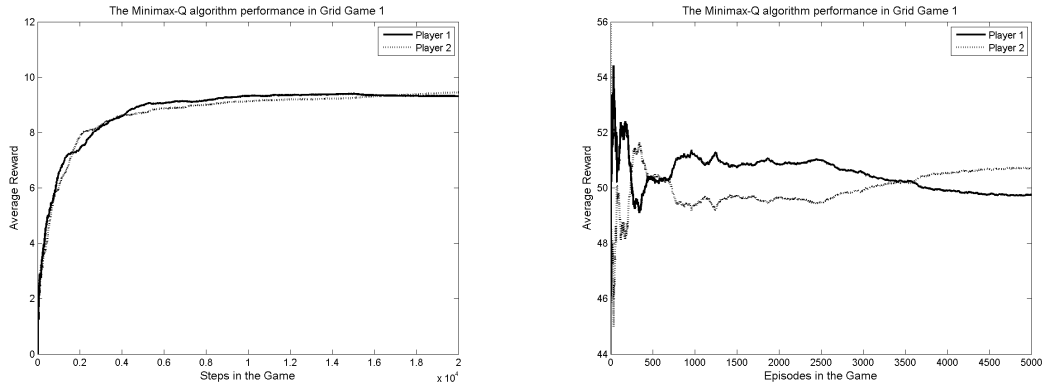


Figure 4.5: Minimax-Q algorithm performance in Grid Game 1

We simulated ten games between two Minimax-Q agents in Grid Game 1. Figure 4.5 shows the average reward over time. We can see that both players have similar performances. The agents' policy did not converge to a Nash equilibrium. In every game, Player 1 and Player 2 will move up to the top and then go down to the bottom until one of the players choose a random action. Depending on the random action, the opponent might have an opportunity to move toward its goal. The only way to have a winner is when they choose a random action. In most of the games there will be only one player that reaches its goal. That is why the average reward for each episode stays around 50 for both players. We can see that the players are still trying to maximize their reward but it is lower than if they had a Nash equilibrium strategy where both player reach their goal at every episode. Let us look at the Minimax-Q algorithm in Grid Game 2.

We decided to test Grid Game 2 to see if the results are as per the literature [17] [9] [11]. We simulated ten games with two Minimax-Q agents in Grid Game 2. The agents did not find the Nash equilibrium strategy in any of the games which was expected in a general-sum game.

We also checked the performance of both players over time. The results of the simulation are in Figure 4.6. We can see that Player 2 earned more reward over time, but they both converge to an average that is similar. We compared the individual

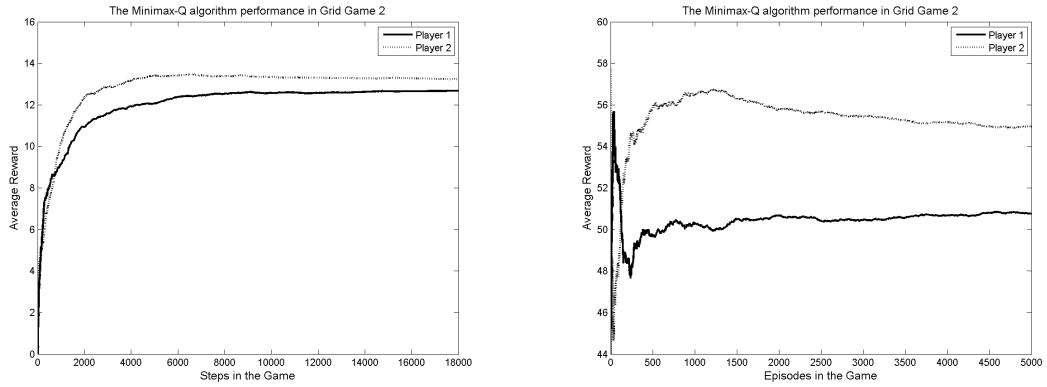


Figure 4.6: Minimax-Q algorithm performance in Grid Game 2

performances of each algorithm in Grid Game 1 and 2. From our simulations of Grid Game 2, the Minimax-Q algorithm did well if we only look at the average reward over time. We see that it finished learning with an average reward over the number of steps of approximately 12. Comparatively, the Nash-Q algorithm had approximatively 8.5 - 8.7 (Figure 4.9) and the WOLF-PHC had approximatively 11 - 14 (Figure 4.12). We can see that even if it did not learn the proper strategy, it still did better than the Nash-Q algorithm and almost as good as the WOLF-PHC. The Minimax-Q algorithm also did better than the Nash-Q algorithm in Grid Game 1, but it still obtained a much lower average rewards over time than the WOLF-PHC algorithm.

We can see that even if the Minimax-Q does not converge to a Nash equilibrium strategy, it still has a good performance when played against itself. We will test the Minimax-Q algorithm against the other two algorithms in the next section. We will also discuss the difference between the algorithms.

4.3.2 Nash-Q Implementation

Our goal with the Nash-Q algorithm is to reproduce the results obtained in the literature by Hu and Wellman. Their results were very conclusive in Grid Game 1 and 2 and showed that the algorithm reached a Nash equilibrium strategy all of the time when playing against itself. We wanted to recreate the situation in our

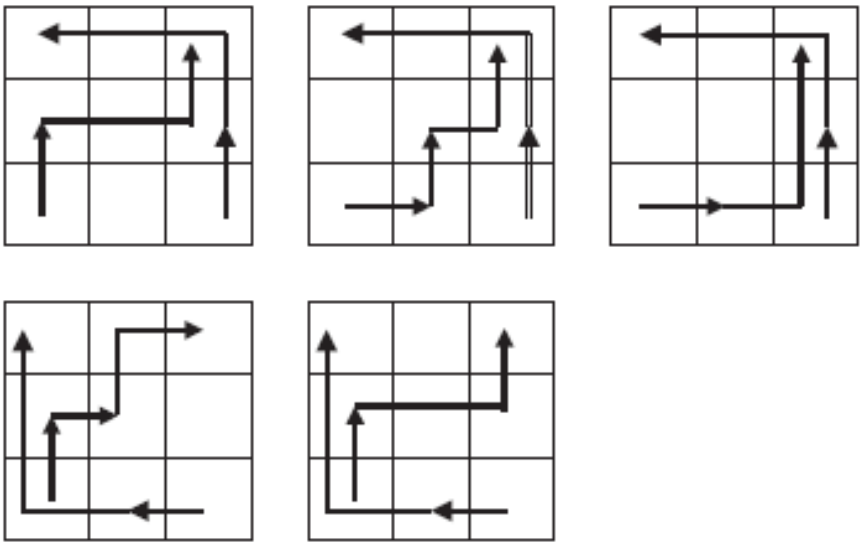


Figure 4.7: Nash equilibrium strategies in Grid Game 1

simulations. Hu and Wellman showed that both agents learned a Nash equilibrium strategy 100% of the time. The high percentage results is a very good indicator that the algorithm works very well in this environment. We wanted to explore possible uses of the algorithm.

There is a specific number of Nash equilibrium strategies in Grid Game 1. Figure 4.7 [8] shows half of the possible Nash equilibria for Grid Game 1. The symmetric variant represent the other half.

The implementation of the algorithm was complex and required very precise adjustments to the code to ensure good results. As explained in Section 3.3.1, the Lemke-Howson algorithm was used to find the Nash equilibrium of the bimatrix games. We adapted a JAVA version of the Lemke-Howson algorithm implemented by Hu and Welman for the Matlab environment. The algorithm is designed to find more than one Nash equilibrium when it exists. It is based on the work from [5]. The inputs are the two payoff matrices for both players and the output are the Nash equilibria. It gives the Nash equilibrium policies for each player in that particular bimatrix game.

We went through multiple versions of our implementation of the algorithm. In

	Up	Down	Left	Right
Up	96,95	0,0	92,92	0,0
Down	0,0	0,0	0,0	0,0
Left	0,0	0,0	0,0	0,0
Right	85,85	0,0	89,85	0,0

Table 4.1: Grid Game 1: Nash Q-values in state (1,3) for all actions

each implementation, we kept the parameters from [8]. The authors 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. As explained in section 4.3.1, α is inversely proportional with the number of times each state-tuples (s, a^1, a^2) are visited. We considered 5000 games a reasonable number of games because after some testing we calculated that each state would be visited an average of 93 times.

We started the implementation with a more general approach to the Nash-Q algorithm. In our first implementation, our agents were always able to choose any of the four actions. This meant that if the agent choose to go into a wall, he 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. We collected the information after a simulation and put it in Table 4.1. By using the whole table, the algorithm calculates more than it needs because many pairs of actions had a Q-value of zero.

We were not able to replicate the results from the literature. We decided to implement with less generality. 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 bounds. It was not an issue to implement this part because of the limited number of cells and the explore-only learning technique that the agent used. This change positively affected the results and we got a success rate of about 25%, which means that we got a Nash equilibrium strategy in 25% of the games. This was still far from the success rate noted from [12]. In our last implementation, the Lemke-Howson

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

Table 4.2: Grid Game 1: Nash Q-values in state (1,3)

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 next state. Table 4.2 shows an example with the initial state $s\,(1,3)$. We were able to achieve 100% success in getting a Nash equilibrium strategies for two Nash-Q learners. Table 4.2 shows the value of the Nash Q-values in the starting state $s\,(1,3)$.

We decided to use the same method as Hu and Wellman to confirm their results. This means that the agent will choose a random action in every state. The starting position of the players change in every game. They start in a random position except their goal cell. This ensures that each state is visited often enough as per Assumption 1. As per [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 was visited often enough which satisfies Assumption 1. We found that if you remove the states where both players occupy 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 a wall), we 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 a version of the algorithm that used an exploit-explore technique. 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 a exploit-explore learning technique where the agent chooses a random action with a

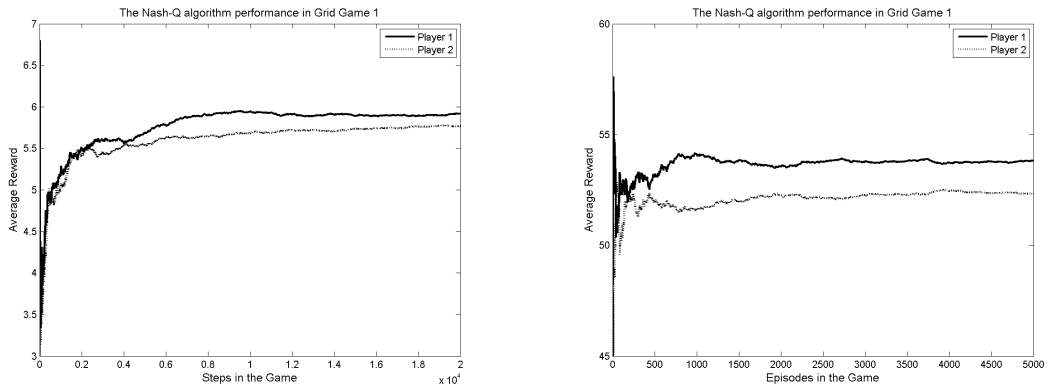


Figure 4.8: Nash-Q algorithm performance in Grid Game 1

probability of $1 - \varepsilon$ and the Nash equilibrium strategy with a probability of ε where $1 \geq \varepsilon \geq 0$. The value of ε varies during learning as per $\varepsilon(s) = \frac{1}{n_t(s)}$ where n_t is the number of times the game was in the state s [8]. The probability that the agent will choose a random action increases with time. This is not a common practice in a learning algorithm 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 a convergence to a Nash equilibrium strategy. The algorithm was design to use a explore-only technique which is not the standard for reinforcement learning algorithms. We will see in the comparison section the effect on performance against other algorithms like Minimax-Q and Wolf-PHC.

We have two sets of figures to illustrate the performance of the Nash-Q algorithm. The first set is the performance of the algorithm when playing against itself using the parameters mentioned earlier. The second set of figure shows a performance comparison when the learning technique of each player changes. In this first set, we have Figure 4.8 that shows the performance of the Nash-Q algorithm in Grid Game 1 and Figure 4.9 that shows its performance in Grid Game 2.

We used two kind of graph to illustrate the performance of the algorithm when playing against themselves. We have a graph with the cumulative average reward over the number of steps and a graph with the cumulative average reward over the

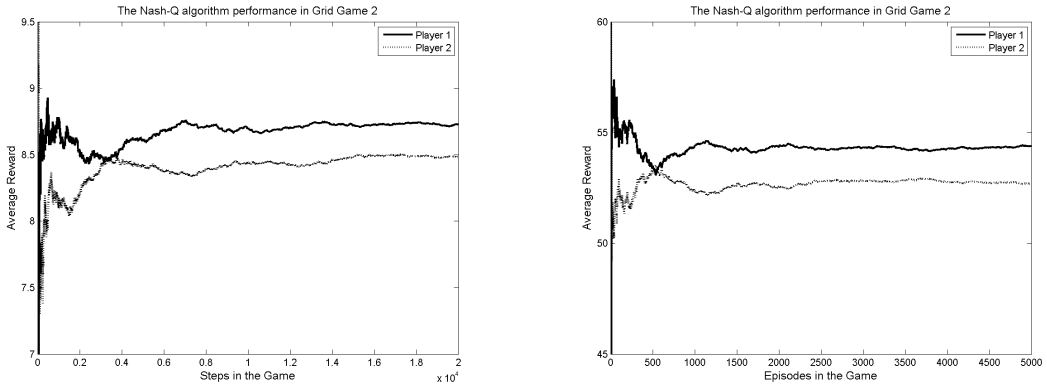


Figure 4.9: Nash-Q algorithm performance in Grid Game 2

number of episodes. We believe that it is important to show both graph to illustrate the differences between each of the algorithm. In this case, the Nash-Q algorithm uses more steps than the other algorithms and it shows in Figure 4.8 and Figure 4.9.

Our final results correspond perfectly with the literature when looking at two Nash-Q learners playing Grid Game 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 its policy. To ensure that the values do not reflect the results of only one game, we computed the average of the values over ten games. It also gives us a smoother curve on the graphs which makes it easier to read.

In the second set of figures, Figure 4.10 illustrates the performance of the algorithm in Grid Game 1 when both agents are Nash-Q learners. It shows the average rewards per step when the two agents use the three learning techniques: exploit-explore, explore-only, and exploit-only.

The results reflect our assumption that the exploit-explore technique would perform better over the course of the game. The results for the explore-only and the exploit-explore are very close because the algorithm was designed to choose actions

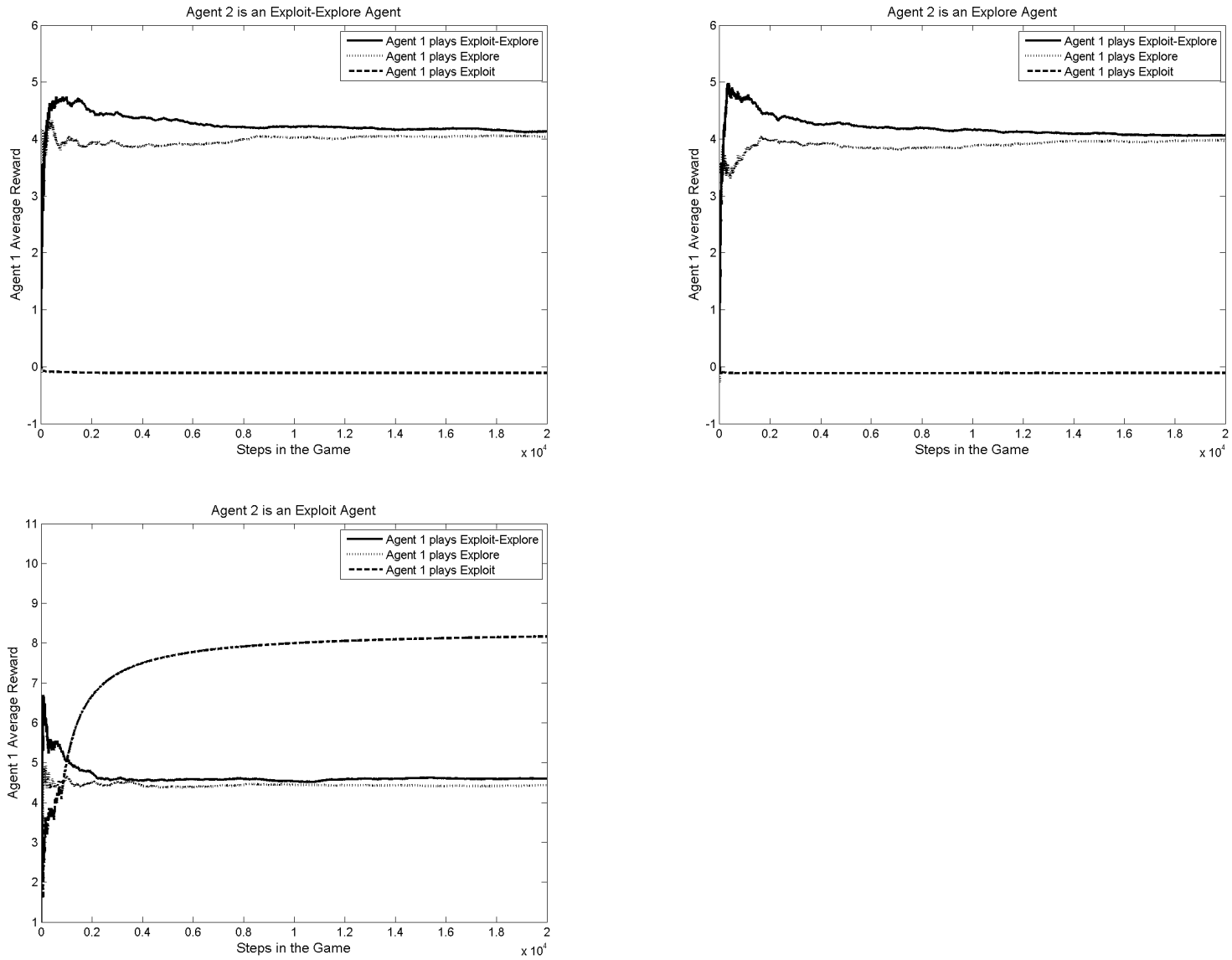


Figure 4.10: Performance with Nash-Q Learning

randomly. The modified exploit-explore version does give an advantage by choosing a greedy policy some of the time which increases its total payoff. The exploit version cannot work without modifying the code to allow some exploration at the beginning. Without any data in the Q-tables, the agent always chooses the same action and gets stuck in endless games.

The implementation of this algorithm was not easy because it cannot work with more general conditions. As explained earlier, we started with a more general implementation where there were no restrictions for the agent. We found that the algorithm does not allow for this kind of generality. The assumptions from the authors remove any kind of flexibility from the algorithm. The Nash-Q algorithm converges to a Nash equilibrium strategy only when all the assumptions and conditions enumerated in Chapter 3, Section 3 are met. The Nash-Q algorithm has a high level of complexity. Each learning agent needs to maintain n Q-tables where n is the number of agents in the environment. The learning agent updates $(Q^1, \ldots, Q^n)$ where each Q^j, $j = 1, \ldots, n$, is made of $Q^j(s, a^1, \ldots, a^n)$ for all states s and actions $a^1, \ldots, a^n$ [12]. We have a fixed number of states $|S|$ and a fixed number of actions $|A^i|$ for each agent i. We calculate that the number of entries in Q^k is $|S| \cdot |A|^n$. The total amount of space required for each agent to maintain n Q-tables is $n\,|S| \cdot |A|^n$ [12]. The space complexity of the algorithm is linear in the number of states, polynomial in the number of actions and exponential in the number of agents. It also needs to evaluate the Nash equilibrium at every step to update its Q-values [8]. The Lemke-Howson algorithm is the equivalent to the simplex algorithm in terms of computation and requires a large amount of processing power as we will discuss in the comparison section.

After implementing the Nash-Q algorithm, we determined that it was not as promising as characterized in [11] or [20]. The algorithm only works in a very specific environment. As mentioned in [11] and [20] and also with our observation during

implementation, the algorithm needs to find the Nash equilibrium at every step of the game. This can only be guaranteed if every state of the game has a global optimal point or a saddle point. We cannot guarantee those conditions for every state in general-sum stochastic games. There is also a possibility that multiple equilibriums exist in certain states. It is computationally heavy to find a Nash equilibrium at every step.

Another aspect of the algorithm that was not taken into account in the reviews is the fact that the algorithm is designed to be used with an explore-only learning method. This method is only good in a problem where we only seek to find a Nash equilibrium strategy of a specific grid game and do not care about the payoff during play. It also takes longer to learn because the agent does not get better over time. The major disadvantage of this method is that the Nash equilibrium strategy learned would have to be relearned if the environment changes. This would mean that the agent would have to go though another session of learning by choosing random actions.

Our implementation showed that the Nash-Q algorithm should not be used outside of a specific environment that respects the algorithm constraints. We agree with the review by [11] that other learning methods would be better suited for a more general environment. We will show in Section 4.3.3 that the WOLF-PHC algorithm would be a better choice.

4.3.3 WOLF-PHC Implementation

Our last algorithm studied in this thesis was the WOLF-PHC learning algorithm. Its implementation was the easiest of the three. The absence of linear programming in the algorithm favours a very fast and adaptable learning process. The WOLF-PHC algorithm is characterized by its variable policy updates. We used a ratio of $\delta_l/\delta_w = 0.5$, where $\delta_l = 2$ and $\delta_w = 4$. This algorithm is characterized by its ability to be used without conditions or assumptions. We tested different ratios of the policy

	Up	Left
Up	71,71	71,65
Right	69,71	69,65

a) Restricted version

	Up	Left
Up	72,71	72,67
Right	64,71	64,67

b) Unrestricted version

Table 4.3: Grid Game 1: WOLF-PHC Q-tables for Player 1 and 2 in initial state s(1,3)

update δ_l/δ_w, exploration variable ε and learning rate α to see if it could still find the Nash equilibrium strategy. We tested both grid games with the WOLF-PHC algorithm as the agent's learning algorithm. We tested both games 20 times and the agents found their Nash equilibrium strategy 100% of the time. We show the Q-values for both players in the initial state of Grid Game 1 in Table 4.3.

The two sets of Q-values are from two different versions of Grid Game 1. We used the same game parameters as the Nash-Q learner to get the first set and we used a more general approach to collect the second set. The main difference between the two implementations was that the learner could choose to move in any direction it wants, even into a wall during learning. In the Nash-Q learner, it was a requirement to not allow that to happen to ensure that the end results is a Nash equilibrium strategy. We can see that the values are similar in both sets. Also, the value for Player 1 *up* and *right* are the same for any action from player 2. This is because the WOLF-PHC algorithm is a single agent learner and it does not take into account the actions of the opponent. We calculated with the help of the Lemke-Howson algorithm the Nash equilibrium action in both the sets. The Nash equilibrium with these Q-values is both player choosing to go *up*. The WOLF-PHC algorithm does not take into consideration the Nash equilibrium when changing its policy like Nash-Q, but the end result is the same.

As with the Nash-Q algorithm, we have two sets of figures to illustrate the performance of the WOLF-PHC algorithm. The first set is the performance of the algorithm when playing against itself using the parameter mentioned earlier. The second set of

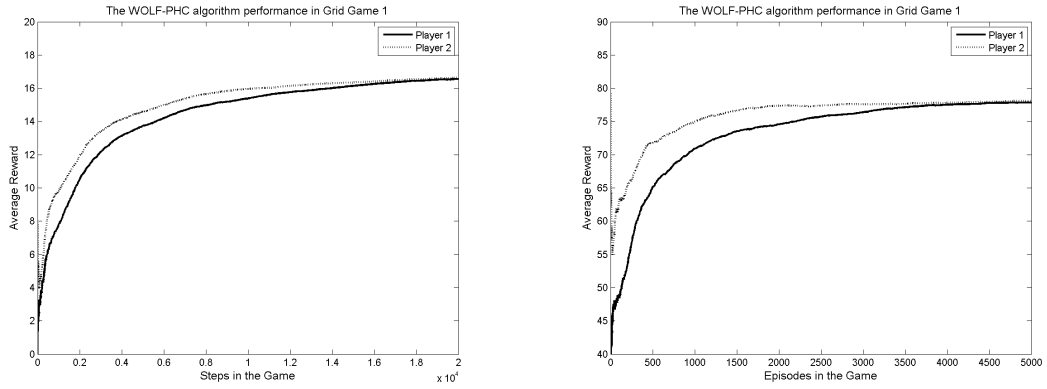


Figure 4.11: The WOLF-PHC algorithm performance in Grid Game 1

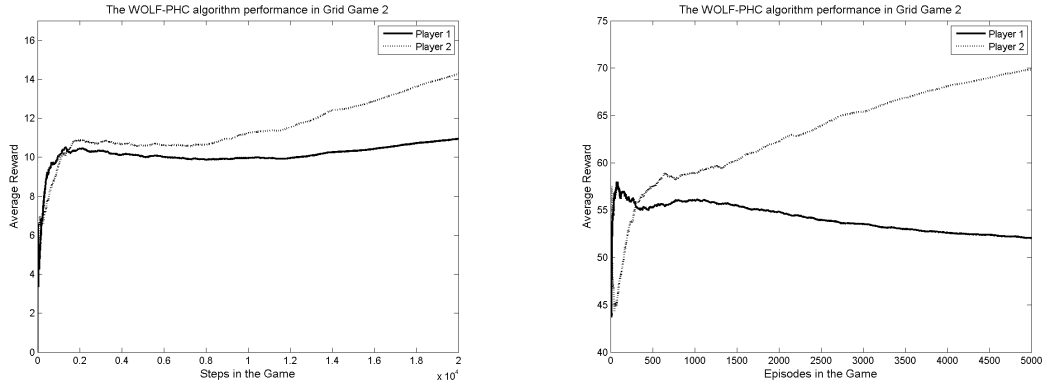
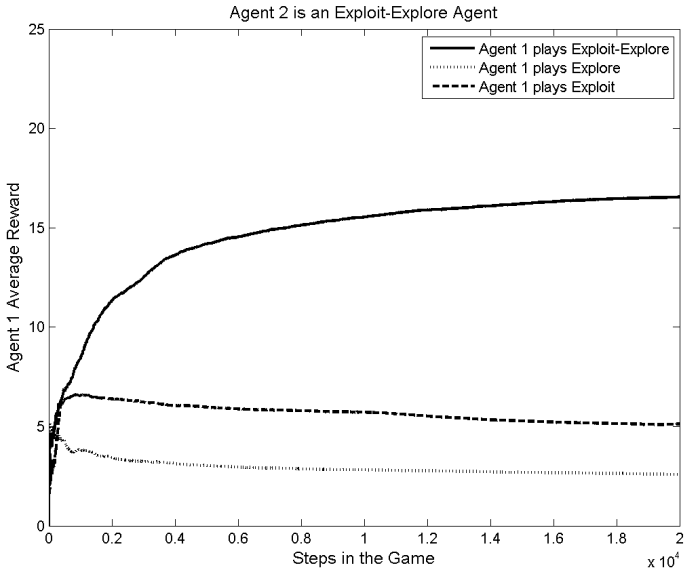


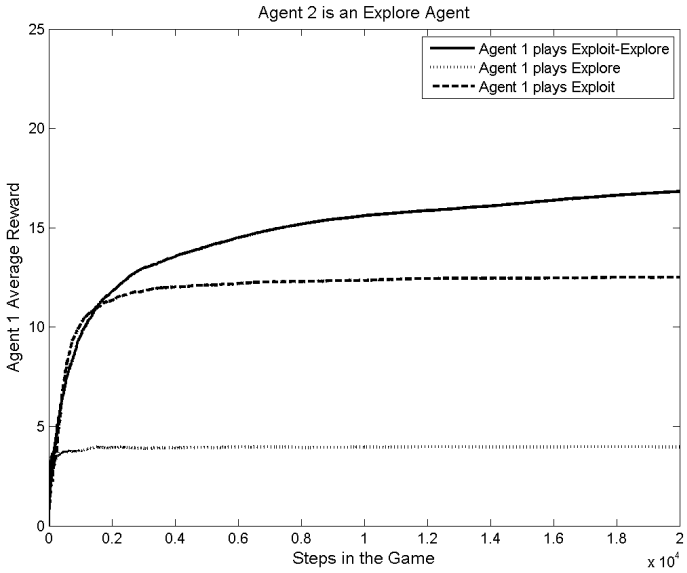
Figure 4.12: The WOLF algorithm performance in Grid Game 2

figures shows a performance comparison when the learning technique of each player changes. In this first set, we have Figure 4.11 that shows the performance of the WOLF-PHC algorithm in Grid Game 1 and Figure 4.12 that shows its performance in Grid Game 2.

In the second set of figures, Figure 4.13 illustrates the performance of the algorithm in Grid Game 1 when both agents are WOLF-PHC learners. As per the previous algorithm, it shows the average rewards per step when the two agents use the three learning techniques: exploit-explore, explore and exploit.

The results are not surprising in the fact that the exploit-explore technique is doing better than the other two. It is easy to see that a good exploration technique ensures good performances.





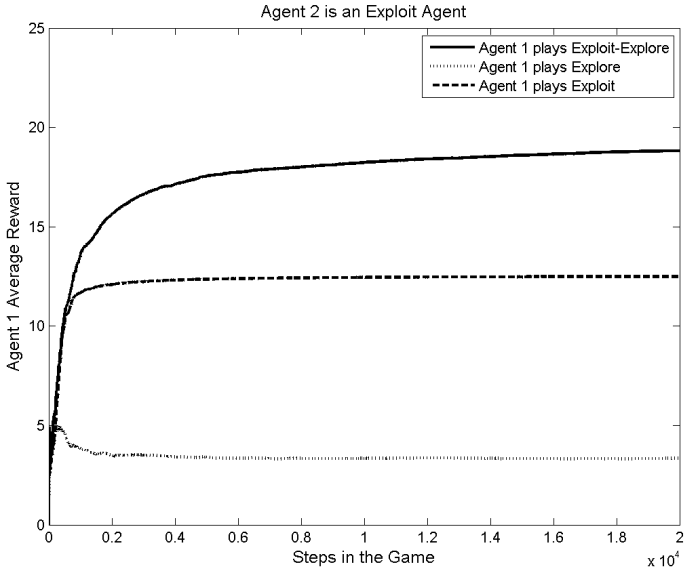


Figure 4.13: Performance with WOLF-PHC

4.4 Algorithm Comparison

Each of the algorithms seen in the last section has advantages and disadvantages. In this section, we compare each algorithm against the others. We will use the following comparison points: explore-only vs exploit-explore, zero-sum vs general-sum, algorithm complexity, computation time and performance. We will see that the algorithms were not all designed for the same purpose but we were able to make a sound comparison.

4.4.1 Explore-only vs Exploit-explore

As explained in Chapter 4 Section 2, Hu and Wellman used the term online and offline in their paper to describe explore-only and exploit-explore learning techniques. The Nash-Q algorithm needs to visit all the states in the environment often enough to make it converge to a Nash equilibrium strategy. For this reason, when the Nash-Q learner chooses its next action it uses an explore-only method. Also, the agent will start in a different state at every episode of the learning process. Both of these requirements are necessary for convergence. This type of algorithm needs to be in an environment that does not change with time, because it would need to learn a new Nash equilibrium strategy. When an agent chooses only random actions, it takes more steps in every episode because it does not get better with time. A good alternative would be to determine when the agent reached the Nash equilibrium strategy and then reduce exploration until there is a change to the environment or to the other player's strategy. This would increase the performance of the Nash-Q algorithm while the player uses the Nash equilibrium strategy. In our simulation, the Nash-Q algorithm does find a Nash equilibrium strategy to ensure that both players maximize their payoffs. The applications of an explore-only learner are not very common but we can see the potential when we need to find a Nash equilibrium strategy between two

	Grid Game 1	Grid Game 2
Minimax-Q	5.380	4.164
Nash-Q	12.565	8.551
WOLF-PHC	4.205	3.700

Table 4.4: Number of steps per episode for Grid Game 1 and Grid Game 2

players where the performance during learning is not important.

The authors, Hu and Wellman, did design a second version of the algorithm that always starts at the normal starting state and used an exploit-explore learning technique. As explained in a previous section, it is not the common exploit-explore technique where the agent chooses an action randomly with probability ε and a greedy action with probability $1 - \varepsilon$ where ε is small $\varepsilon = 0.2$. In this version of the Nash-Q algorithm, the agent chooses an action randomly with probability $1 - \varepsilon$ and a greedy action with probability ε. The variable ε also decreases with time. This means that the agent will choose actions more and more randomly as time goes. We will look at performance later on but we can see that it will be affected. The WOLF-PHC algorithm and the Minimax-Q algorithm use the standard explore-exploit method where the agent chooses a greedy action with a probability of $(1 - \varepsilon)$.

During our research we were surprised by the way Hu and Wellman used their algorithm. It is counter intuitive to have a learner choose actions more randomly with time. The end result is still the same as either Nash-Q or WOLF-PHC, they both reach a Nash equilibrium strategy which was the goal. The main difference between both learning processes would be the number of steps needed and the average reward over time. By choosing randomly at every step, the Nash-Q algorithm does not get better over time. Table 4.4 shows the average number of steps per episode for each algorithm in the different games.

4.4.2 Zero-sum vs General-sum

In this thesis we simulated both zero-sum and general-sum games. As explained in Chapter 2 Section 5, in a zero-sum game the reward function for Player 1 and Player 2 is $R_1 = -R_2$. In a general-sum game the players' reward function do not need to follow that rule. Therefore, the addition of all the rewards do not equal zero. General-sum games as the name says are more general and can be customised to many different situations. Zero-sum games are usually stage games where it is easier to ensure that $R_1 = -R_2$. The algorithm that we studied were all supposed to converge in either zero-sum or general-sum games. The Minimax-Q algorithm on its own does not converge in a general-sum game. Littman [17] designed another algorithm called the Friend-or-Foe algorithm which is an adaptation of the Nash-Q algorithm. It uses the Minimax-Q algorithm when playing against "foes" and the Q-Learning algorithm when playing against "friends". The Friend-or-Foe does not converge to Nash equilibrium all the time in general-sum games. The algorithm does converge to a result, but that result might not be a Nash equilibrium. From our simulations we found that the Minimax-Q algorithm does not converge to a Nash equilibrium in either Grid Game 1 or 2. The WOLF-PHC algorithm converges to a Nash equilibrium in both zero-sum and general-sum games. We did not test the Nash-Q algorithm for zero-sum game but it does converge to a Nash equilibrium in a general-sum game. We could not test the Nash-Q algorithm in our zero-sum matrix games because it was designed for stochastic games. The algorithm uses the Nash equilibrium of the next state when updating its Q-values and matrix games have only one state.

4.4.3 Algorithm Complexity and Computation Time

The complexity of an algorithm affects its usefulness and efficiency. In our situation, reinforcement learning uses different updating methods and some of them are more

complex than others. A complex algorithm might be more prone to errors and usually takes longer to compute. Another factor of complexity is the amount of data that the agent has to keep in memory. We will look at each of our algorithms and discuss their complexity.

The Minimax-Q algorithm uses the Q-learning updating technique adapted for a multiagent stochastic game. The agent needs to keep track of its own position and its opponents position. The main difference in the Minimax-Q algorithm is that it needs to find the action that minimizes its opponent's reward while maximizing its own. To achieve this, the Minimax-Q algorithm uses linear programming to find the state-value V of state s'. The two main issues with linear programming is that it is computationally heavy and there is a potential for errors. In our version, we use the Matlab function *lineprog* which has a long computation time. We did not find or recreated a simplex algorithm in Matlab, but it could have been a solution to the long computation time. The average computation time for one episode in Grid Game 2 is 101.45 milliseconds with MATLAB. All the simulation where done on a computer with an AMD Athlon$^{\text{TM}}$II x4 635 2.90 GHz on Windows 7$^{\text{TM}}$Professional 64 bits with 4 Gb of RAM. As a comparison the average time to compute one episode of the WOLF-PHC algorithm takes 0.75 milliseconds with MATLAB in Grid Game 1 or 0.66 milliseconds in Grid Game 2. This is a factor of approximately 154. In our simulation we are looking at 5000 episodes, but it could be much more in other situation. The computation time affect the time needed to go through each simulations. To ensure proper solutions we also needed to have error checking after each optimization. Table 4.5 shows the average computation time per episode for Grid Game 1 and 2 for our three algorithms.

The Nash-Q algorithm uses the same modified multiagent Q-learning updating technique as the Minimax-Q algorithm, but in this version the State-value $V(s')$ is replaced by $NashQ(s')$. The value of $NashQ$ is the expected reward in state s' if

	Grid Game 1	Grid Game 2
Minimax-Q	161.5	114.8
Nash-Q	16.32	10.9
WOLF-PHC	0.75	0.66

Table 4.5: Average time in milliseconds (ms) to compute one episode for Grid Game 1 and Grid Game 2 on MATLAB

the agent chooses the Nash equilibrium action from that point on. The agent has to calculate a Nash equilibrium at every step. To find the Nash equilibrium, we used the Lemke-Howson algorithm which requires as much computation as linear programming because it is similar to the simplex algorithm. Also, to be able to calculate the Nash equilibrium the agent needs to keep an updated Q-table for itself and one for its opponent. The agent has to keep track of all the actions and rewards its opponent gets over the course of learning. So instead of keeping track of $Q(s, A_1, A_2)$ where A_i is the action space for player i and s is the current state, the agent has to keep $Q^i(s, A_1, A_2)$ where i is the number of players. This doubles the amount of data to keep in memory. The Nash-Q algorithm was also very complex to code because of the restrictions that we discussed in the Nash-Q implementation section. To reiterate, the players could not try to move out of the grid which means we had to ensure that when the agent chooses an action it would not cause it to go out of bounds. This was necessary to obtain the expected results. The learning agents in this algorithm need to have information about the game before starting to play because their action space change in every states. Another restriction was for the calculation of the Nash equilibrium with the Lemke-Howson algorithm. We had to ensure that only the possible actions were taken into account in the bimatrix game. It was hard-coded in each state because every state was different. For example, in state $s(1, 3)$, Player 1 could go *up* or *left* and Player 2 could go *up* or *right*. When calculating the Nash equilibrium the bimatrix had to be 2X2 matrices with only the available actions. The average computation time for one episode in Grid Game 1 is 16.32 milliseconds with

MATLAB. There is a factor of approximately 22 between the Nash-Q algorithm and the WOLF-PHC algorithm. Our version of Minimax-Q uses the linear programming function of Matlab which is not fast compare to our Lemke-Howson function. This would be an improvement to have a better function for Minimax-Q.

The WOLF-PHC algorithm uses the basic Q-learning update method. Instead of using linear programming to update its policy, WOLF-PHC uses a learning rate that varies depending on if the agent is winning or losing. This means that on top of its own Q-table and the average policy, the WOLF-PHC algorithm does not use a lot of memory or computing power. Also it does not take into account the position of the opponent. This reduces even more the memory needed. The WOLF-PHC algorithm is easy to implement, requires low memory usage and has a low computational cost. This makes it a useful learning algorithm.

4.4.4 Performance

The performance of the three algorithms individually was discussed in their respective section but we wanted to test them against each other. We chose the average reward over time as the performance measurement. We also verified if they learned a Nash equilibrium strategy. When playing against themselves they all reached a Nash equilibrium strategy. We used diminishing values for α, ε, δ_w and δ_l because we found from our simulations and the literature [18] [8] [9] that it increases performance over time. The learning rate α will decrease by multiplying by a decay variable $decay = 0.99862$ which means that after 5000 episodes, the learning rate will be equal to $\alpha = 0.001$. The learning rate δ_w for the WOLF-PHC algorithm will decrease by a decay variable $decayD = 0.99895$ and will be equal to 0.001 after 5000 episodes. The learning rate δ_l will be updated to keep the ratio $\delta_l/\delta_w = 2$ which means that it will be equal to twice δ_w. The exploration variable ε will also decrease with time to reduce the amount of random actions toward the end of the learning simulation.

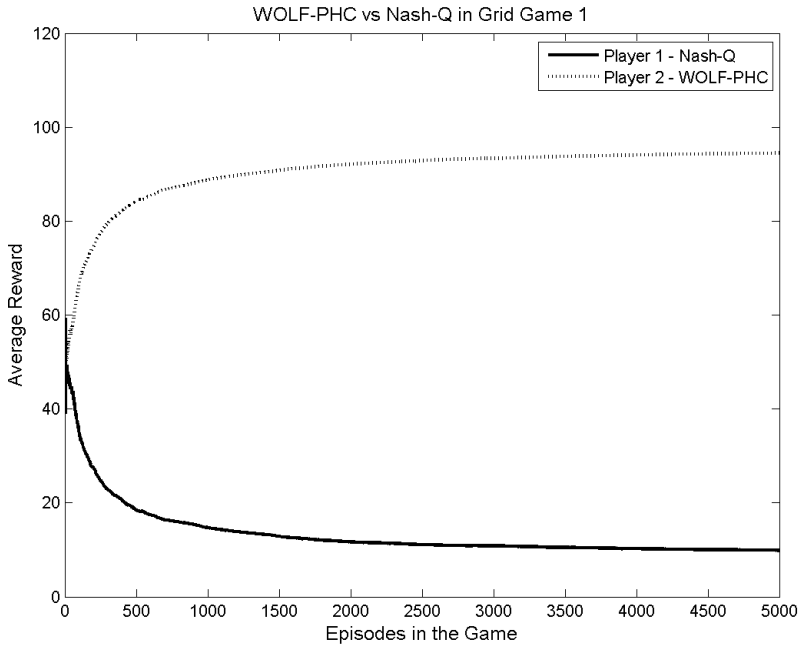


Figure 4.14: Performance comparison between Nash-Q and WOLF-PHC in Grid Game 1

It will decrease by multiplying it by a decay variable $decayE = 0.99895$ and reach a value of 0.001 after 5000 episodes.

We started with the Nash-Q algorithm and the WOLF-PHC algorithm. We used Grid Game 1 as the benchmark for this simulation. We discussed earlier that the Nash-Q algorithm has a disadvantage in a performance test like the one we are using because of its unorthodox exploration technique. The Nash-Q agent will choose actions more randomly with time instead of the opposite like the WOLF-PHC algorithm. We anticipated that the Nash-Q agent would get less rewards over time. The results of the simulation are plotted in Figure 4.14.

As predicted the Nash-Q algorithm received less rewards over time than the WOLF-PHC algorithm. With an explore-only learning technique, the Nash-Q algorithm cannot compare with the exploit-explore learning technique of WOLF-PHC when looking at the average reward per episode. In this situation, the WOLF-PHC algorithm will find its Nash equilibrium strategy and choose the greedy actions from

that point on to maximize its rewards. The probability of choosing a random action will diminish over time. The Nash-Q algorithm will find its Nash equilibrium strategy but it will not be able to use the information to increase its payoff because of its learning technique. In this simulation of ten games, they both found the Nash equilibrium during learning.

The next simulation is between Minimax-Q and WOLF-PHC in Grid Game 2. We are using Grid Game 2 as the environment because the Minimax-Q algorithm was proven to be unable to reach an equilibrium in Grid Game 1. It was also shown that it could not find the Nash equilibrium strategy in Grid Game 2. In this situation, we did not have any insights into the results because both used the standard exploit-explore learning technique. In Grid Game 2, there are two Nash equilibriums: Player 1 goes *right* and Player 2 goes *up* in the starting state or Player 1 goes *up* and Player 2 goes *left* in the starting state. This is caused by the barriers that we can see in Figure 4.1 b). The best strategy is to go by the middle cell and then go to the goal cell from there, but if both agents try to go in the middle they will be sent back to their original positions. To reach a Nash equilibrium, one of the agents will have to "sacrifice" itself by going through the barrier and will get less rewards over time. The results of the simulation can be found in Figure 4.15.

The Minimax-Q algorithm did not find the Nash equilibrium when playing against WOLF-PHC and its average reward over time is lower than its opponent's. We looked at the policy for the Minimax-Q agent in the initial state and we found that it converged to a solution that did not maximize its reward and made it try to move into the wall 33 % of the time. The WOLF-PHC algorithm was able to find its best strategy by choosing to go in the middle cell and was able to receive a high average reward over time.

Our final comparison simulation was between Nash-Q and Minimax-Q in Grid Game 2. We choose Grid Game 2 again because of the reasons mentioned above. From

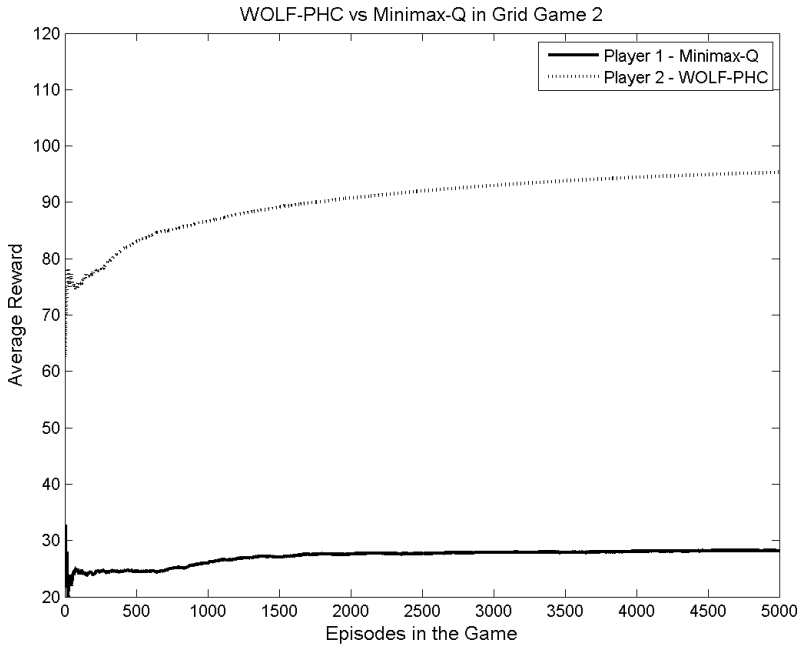


Figure 4.15: Performance comparison between Minimax-Q and WOLF-PHC in Grid Game 2

the results in the simulation between Minimax-Q and WOLF-PHC we expected to see the same kind of results by the Minimax-Q algorithm. The results of the simulation are in Figure 4.16.

The Nash-Q agent was able to find the Nash equilibrium and maximized its rewards over time but the Minimax-Q agent was not successful. The results are very similar to the last simulation results in Figure 4.15. The Minimax-Q algorithm is not able to find the Nash equilibrium strategy to maximize its rewards.

The simulations in this section showed that not all the algorithms studied had good performance when played against each other. We can see that the WOLF-PHC algorithm has consistently out performed the other two. The results also showed the problems of an explore-only learning technique like the Nash-Q algorithm when it comes to performance. We also realised that the Minimax-Q algorithm does not converge to a Nash equilibrium when playing against the other two algorithms.

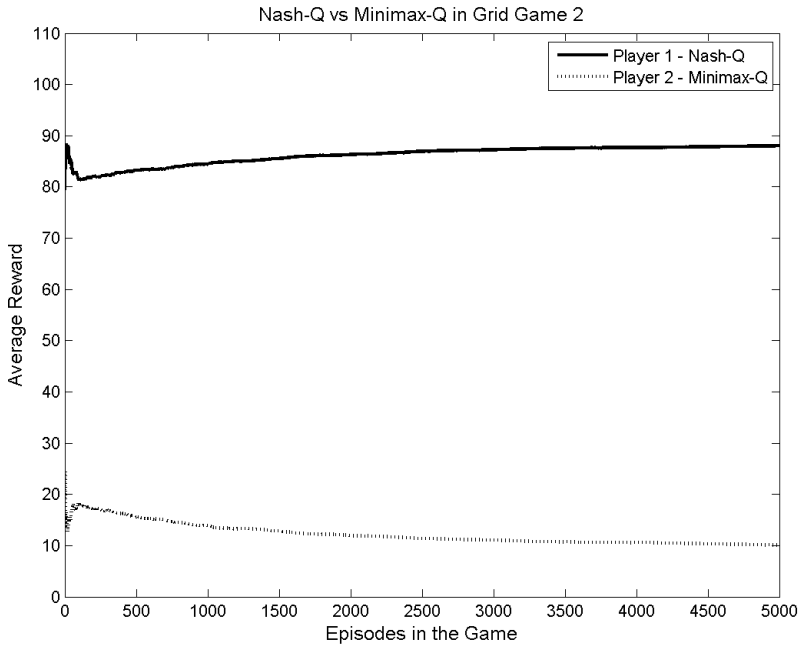


Figure 4.16: Performance comparison between Nash-Q and Minimax-Q in Grid Game 2

4.4.5 Summary

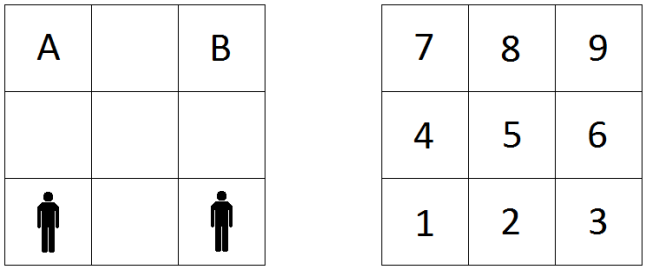
In this section we compared each of the three main algorithms of this thesis. We discussed the reasons that the Nash-Q algorithm needs to use an explore-only learning technique to ensure a convergence to a Nash equilibrium and the effects on its performance. The WOLF-PHC and the Minimax-Q algorithms use the standard exploit-explore technique to maximize their reward over time during learning. The Nash-Q algorithm is a good solution if the performance during play is not important and the end result is the only thing that matters. We explained that the Minimax-Q algorithm was not designed for general-sum games but that it was able to have a comparable performance to the other two algorithms. A general-sum game environment was shown to be more flexible than the zero-sum games. We demonstrated that each algorithm studied had different complexity and computation time. The WOLF-PHC algorithm was the one with the lowest computation time and was the least complex of the three algorithms we tested. The main reason why the Minimax-Q algorithm

takes much more time is because of the Matlab function which is not optimized for our example but for a much more general application. A solution would be to code a simplex algorithm to replace the Matlab function. Finally, we compared the performance of each algorithm when playing against each other. The fact that each algorithm was designed for different environments made the comparison more complex. We found that the Nash-Q and the WOLF-PHC algorithms find their Nash equilibrium even when playing against each other. The Minimax-Q is not designed for general-sum games which is why it was not able to perform well in our comparison.

This comparison section showed that the WOLF-PHC was a good learning algorithm that could adapt to different situations. We wanted to do some more simulation on the algorithm and test its boundaries. In the last section of this thesis we will experiment with the WOLF-PHC algorithm in different environments. In the next section we are looking into the learning rate α more closely and we are comparing the effect of a constant learning rate versus a diminishing one.

4.5 Learning Rate Adaptation

In this section we discuss the different learning rates α used during our experiment and we test them to see which one is best. As explained in Chapter 2, the learning rate $\alpha \in [0, 1]$ is very important. It affects the way the agent learns by dictating how much of the new information to keep. A high learning rate will replace its Q-values almost completely every time it learns new information where a low learning rate reduces the amount of information the agent learns each time. A learning rate of zero would mean that the agent is not learning anything from the action it chooses. In a deterministic environment a learning rate $\alpha = 1$ would be sufficient because the rewards for any action in any state will not change. For example, a single agent in a non-changing and deterministic environment will always get the same reward when



a) Grid game 1 b) Cell numbers

Figure 4.17: Grid Game 1 starting position with the cell's numbers

choosing action a in state s. In this case, we want the agent to learn as fast as it can. A learning rate of 1 will ensure that he gets the full reward at each step. A stochastic environment on the other hand will be very different by having a possibility that the reward from action a in state s could change over time. This is illustrated with Grid Game 1 in Figure 4.17. In the starting state, player 1 can go *up* or *right* and player 2 can go *up* or *left*. If player 1 chooses *up* and player 2 chooses *left*, the players will get a reward of 0. But if player 1 goes *right* and player 2 goes *left*, they will get both a reward of -1. During learning the players will be in the same state more than once which means that both situations will happen at least once. Player 2 would then receive two different rewards for the same action in the same state. In this situation, if the learning rate is 1, player 2 would never keep in memory that the same action could mean either a reward of 0 or -1.

For a stochastic environment we can use an incremental update rule

$$Q_{k+1} = Q_k + \alpha \left[r_{k+1} - Q_k \right], \tag{4.1}$$

where the learning rate is, $\alpha \in [0, 1]$, is constant and Q_k is the weighted average of

the k past rewards. We assume a constant learning rate α.

$$\begin{aligned} Q_k &= Q_{k-1} + \alpha \left[r_k - Q_{k-1} \right] \\ &= \alpha r_k + (1-\alpha) Q_{k-1} \\ &= \alpha r_k + (1-\alpha) \alpha r_{k-1} + (1-\alpha)^2 Q_{k-2} \\ &= \alpha r_k + (1-\alpha) \alpha r_{k-1} + (1-\alpha)^2 \alpha r_{k-2} + \ldots \\ &+ (1-\alpha)^{k-1} \alpha r_1 + (1-\alpha)^k Q_0 \\ &= (1-\alpha)^k Q_0 + \sum_{i=1}^{k} \alpha (1-\alpha)^{k-i} r_i \end{aligned} \tag{4.2}$$

From Equation (4.2), the sum of $(1-\alpha)^k + \sum_{i=1}^{k} \alpha (1-\alpha)^{k-i} = 1$ and it is called a weighted sum [3]. We know that $(1-\alpha) \leq 1$, which means that the weight of each individual reward is distributed over time. The weight of the reward r_i diminishes over time with each new reward.

As an example, we have $Q_0 = 0$ and a learning rate of $\alpha = 0.5$. After the first action, we get a reward of $r_1 = 10$ which means that $Q_1 = 0.5^1 * 0 + \sum_{i=1}^{1} 0.5 * 0.5^0 * 10 = 5$. The second time we visit the same state and use the same action, we get another reward $r_2 = 10$. Now the value of $Q_2 = 0.5^2 * 0 + 0.5 * 0.5^0 * 10 + 0.5 * 0.5^1 * 10 = 7.5$. As we can see the weight of the second reward was diminished. A constant learning rate provides adaptability to the learner against opponents that change strategy over time. For a continuous environment where the agent has to be able to deal with changes over time a constant learning rate can provide a good solution.

The learning rate can also change from one step to another. The learning rate will usually start high and finish low. As such the rewards will have a bigger effect at the beginning than at the end. It will be amplified when put into the weighted sum equation where later rewards will have an even lower weight. A variable learning rate will converge better than a constant learning rate as long as it follows the conditions

dictated by the stochastic approximation theory for convergence [3]. In a stochastic environment with a variable learning rate, the conditions on α are:

$$\sum_{k=1}^{\infty} \alpha_k = \infty \tag{4.3}$$

$$\sum_{k=1}^{\infty} \alpha_k^2 < \infty. \tag{4.4}$$

Equation (4.3) ensures that each update is large enough to overcome any initial conditions or other maxima [3]. Equation (4.4) ensures convergence when the learning steps become very small.

We experimented with three types of learning rates: constant, decaying each step, and the inverse of the number of visits to the state-action tuple. We will describe each type and show the results obtained in Matlab. We used the WOLF-PHC algorithm to test the different types of learning rates because we considered it better suited for the experiment. It is faster and needs a lot less computational power than the two others for the same results. We used Grid Game 1 as the benchmark and both players were WOLF-PHC learners. Grid Game 1 is stochastic but the grid is deterministic which gives both players the same chance of winning the game by using a Nash equilibrium strategy. To compare each learning rate we used the performance of the algorithm over 5000 episodes. The average reward per step is calculated by adding all the rewards obtained by each player and dividing it by the number of steps. We can also calculate the performance per episode. We will use the resulting graphs to demonstrate how each learning rate affects the performance of the algorithm.

We started with constant learning rates. We chose $\alpha = 0.2$, 0.5 and 0.9. We wanted to test three different values of $\alpha \in [0, 1]$ that would represent a low, medium and high constant learning rate. From the condition in Equation 4.4, a constant learning rate does not meet the condition for convergence in a stochastic environment. However, in our tests we found that Player 1 and 2 do converge to a Nash

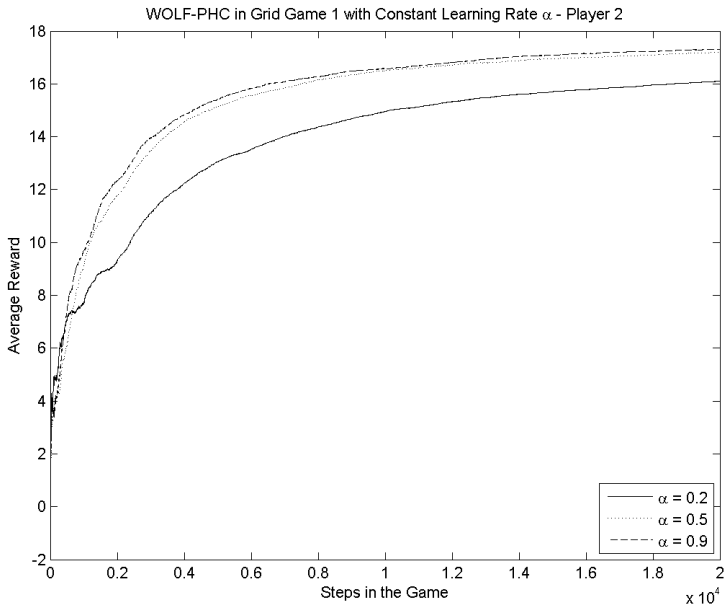


Figure 4.18: WOLF-PHC in Grid Game 1 with Constant Learning Rate α for Player 2

equilibrium strategy. Our expectation for this test was that the higher learning rate would maximize the reward by learning the equilibrium faster. The results are in Figure 4.18. We can see that the best results for the constant learning rate is with $\alpha = 0.9$.

Our interpretation of the results confirmed that the higher learning rate would perform better during the learning process. Figure 4.18 also shows that after approximately 20,000 steps, the performance of $\alpha = 0.5$ and $\alpha = 0.9$ converge to a similar value. The third learning rate $\alpha = 0.2$ seems to be converging too but at a lower value than the other two. We will consider that the best of these learning rates is $\alpha = 0.9$ because it had a higher reward average over the whole simulation.

The next test was done with variable learning rates. The first one is decaying over time. It means that α is multiplied by a *decay* variable at every step of the game. We represent it as: $\alpha = 0.9997^k$ where k is the number of steps. A step consists of each agent choosing an action, taking the action, receiving the reward and updating their Q-tables. The decay variable was set to a value that would decrease α to a

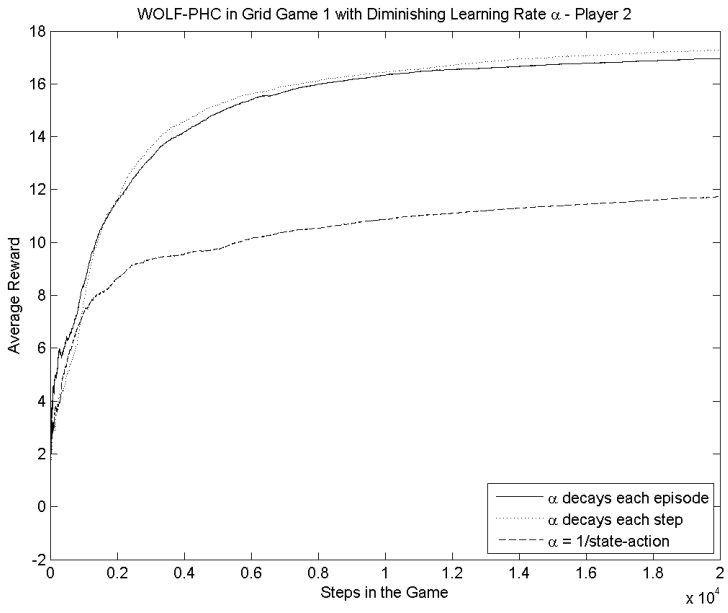


Figure 4.19: WOLF-PHC in Grid Game 1 with Diminishing Learning Rate α for Player 2

negligible value after 22000 steps ($0.9997^{22000} = 0.0014$). The second learning rate is decaying at every episode of the game and it is represented as: $\alpha = 0.999^k$ where k is the number of episodes. It follows the same decaying rules as the previous rate but decays every episode instead of every step. An episode represents the number of steps where the agents go from the starting position to the goal position. In this case, each game has 5000 episodes ($0.999^{5000} = 0.0067$). The third rate is represented by : $\alpha\left(s, a^1, a^2\right) = \frac{1}{n_t(s,a^1,a^2)}$, where s is the present state, a^1 and a^2 are the actions of player 1 and 2, and n_t is the number of times that the state-action tuple has been visited. This means that α will decrease every time the players choose action a^1 and a^2 in state s.

The results for these three rates are very interesting because each type of learning rate has a different effect on the performance of the algorithm. The learning rates decaying at every step or at every episode reached a high reward average over the course of the simulation. The state-action learning rate had the worst performance

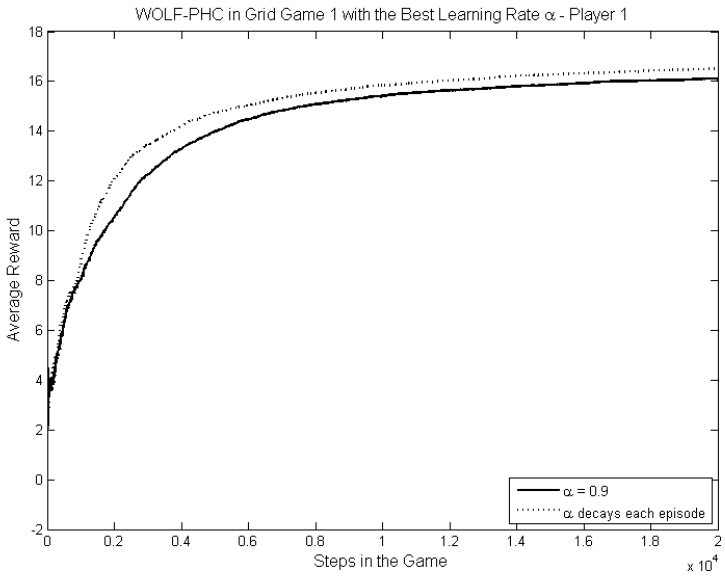


Figure 4.20: WOLF-PHC in Grid Game 1 with the Best Learning Rate α for Player 1

over time. The main difference between each of the rates is that the first two decrease slower than the latter. This could mean that the policy learned by the agent does not maximize its rewards and it cannot learn a better policy because its learning rate is too low.

We plotted the two best results from the previous simulations in Figure 4.20. We kept $\alpha = 0.9$ and $\alpha = 0.999^k$ with k as the number of episodes.

In our simulation with Grid Game 1 and using the WOLF-PHC algorithm, we can say that the two best learning rates will provide very good performances while learning. The main difference would be how quickly the agents get to their Nash equilibria. In this example, the agents can both find a Nash equilibrium to maximize their reward, but if we had a changing environment the constant learning might be able to out perform the others because it would always be able to update its Q-values. The differences between each of the learning rates tested are not dramatic. We could say that any of them can be used with the WOLF-PHC algorithm for Grid Game 1. However, as seen in the literature, each author uses a different learning rate to obtain

the desired results. Specifically, the Nash-Q learner needs to use a state-action based learning rate to be able to converge to a Nash equilibrium $\left(\alpha\left(s, a^1, a^2\right) = \frac{1}{n_t(s,a^1,a^2)}\right)$. The reason is that it needs to meet Assumption 2 to ensure that the algorithm will converge to a Nash equilibrium. The designer needs to know in what environment the agent will work when choosing a learning rate. Usually, the learning rate is chosen for the algorithm specifically as in the Nash-Q algorithm to ensure convergence. However, sometimes the environment will dictate the proper learning rate to use. In a non-deterministic environment the agent would need to adapt its strategy over time either to be flexible against an opponent that changes policy over time or a changing environment. A constant learning rate would do better because the agent will continue to learn at the same rate. In this case the environment dictates how to use the learning rate.

We determined that the choice of the learning rate is based either on the environment or on the algorithm. We discovered that the learning rate in Q-learning related algorithms is not the primary concern to the author of the algorithm that we studied. They applied the theory and used the learning rate that would meet their needs. We showed by simulations that a diminishing learning rate will outperform a constant learning rate while learning but they will eventually reach a similar average reward over time. A constant learning rate has advantages in some situations where we need to continue learning because the other agents or the environment changes over time. With the results from this section, we will look more closely into the WOLF-PHC algorithm when used in a different environment than the one tested earlier.

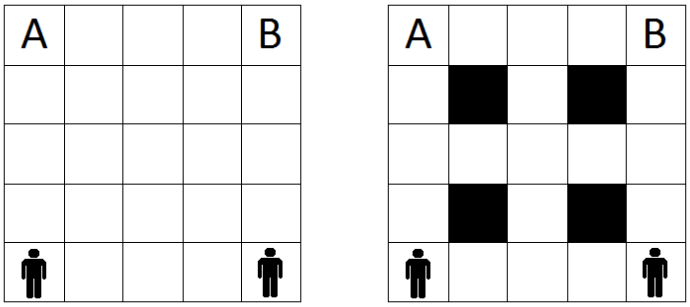
4.6 WOLF-PHC

During the course of our simulations, we discovered that the WOLF-PHC algorithm could adapt to a variety of environments. The fact that it is a single agent learner

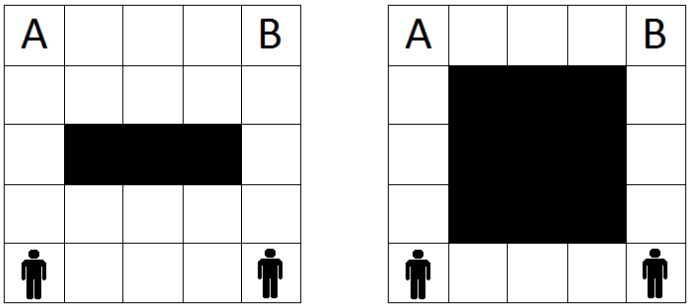
did not impede its efficiency and in fact helped to make it the best learner that we tested. We discussed the differences with the other algorithms earlier in the text but we wanted to have a section only for WOLF-PHC where we test the algorithm in other games. We will look into types of games that are similar to the two previous grid games but also some obstacle avoidance games.

We will describe the parameters of the learner for each game but we wanted to explain that we decided to test a more general version of WOLF-PHC than the one tested with Grid Game 1. We wanted the algorithm to be as flexible as possible and we needed to know if it could still converge in a more general approach. In each state of the games, the agent will be able to choose any of the actions available even if it leads him into a wall. Also, when choosing randomly for exploration the agent will have an equal chance to choose any of the actions available. As for the learning rate α, we decided to go for a diminishing rate at every episode as per the results in the last section. The number of episodes in a game is always known before the learning starts. We can then calculate a proper decaying value for the learner. The learning rate will start at a value of $\alpha = 1$ and it will decay at every episode until it reaches $\alpha = 0.001$ when learning is over. This value was chosen to render the learning rate negligible at the end of learning. The values of δ_w and δ_l will also diminish with every episode starting at $\delta_w = 0.2$ and $\delta_l = 0.4$. We will multiply δ_w with a decay variable $decayD$ at the end of every episode which will bring it to $\delta_w = 0.001$. We will update the value of δ_l by multiplying δ_w by the ratio $\delta_l/\delta_w = 2$. The value of $\gamma = 0.9$ is the same as in the previous simulations. We will also use the Q-Learning algorithm to do some comparisons in terms of performance to see how big a difference the variable policy updating does to learning.

The first game that we tested had the same rules as Grid Game 1 but it had a 5x5 grid as per Figure 4.21. We wanted to know if the agents could reach a Nash equilibrium strategy in this bigger game grid. The main difference with the



a) Grid Game 5X5 b) Four Corners Obstacles



c) Center Line Obstacles d) Square Obstacles

Figure 4.21: Grid Game 5X5 starting positions with different configurations

previous grid game is the number of states the agent can visit. We calculated that this environment has 552 possible states when we remove the states where both agents are on the same cell and we also remove the states where one of the two agents is on its goal cell. The greater number of states and steps needed to get to the goal affected the learning time significantly. Figure 4.22 shows the performance of the WOLF-PHC algorithm when playing against itself. We changed the number of episode per game to 50 000 to reach a point where we can see that the values are converging.

The results of the simulation were very conclusive and the learner found a Nash equilibrium strategy in each of the ten games played. There is more Nash equilibrium strategies in this new grid than in Grid Game 1 from Hu and Wellman. We were confident that WOLF-PHC would be able to adapt to a larger grid and find a Nash equilibrium strategy. This new larger grid gave us more flexibility to change the environment and test the effect on the agents.

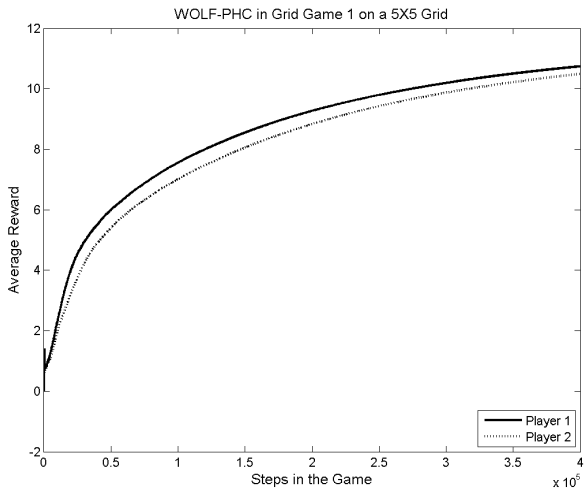


Figure 4.22: Performance of WOLF-PHC in 5X5 grid game

In our next simulation we decided to test the same grid but with pre-determined obstacles that the agents were not able to go through. We decided on three different grid configurations that are shown in Figure 4.21.

The first environment has four obstacles in cells (7, 9, 17 and 19) as per Figure 4.21 b). This configuration has multiple Nash equilibriums but we wanted to see the effect of new obstacles in the middle of the grid instead of the outer bound. WOLF-PHC found a Nash equilibrium strategy in every game that we tested. We demonstrated the effect on performance in Figure 4.23 when both agents are using the WOLF-PHC algorithm. The next two games tested are modified versions of the first. As per Figure 4.21 c) and d), the first grid game has obstacles in cells (12, 13 and 14) and the second has obstacles in cells (7, 8, 9, 12, 13, 14, 17, 18 and 19). We tested the second grid to reduce the number of paths to their goal states. The last one was to see the effect of a reduction on the number of Nash equilibrium strategies to two. We show its performance in Figure 4.24 and Figure 4.25.

The WOLF-PHC algorithm proved to be able to adapt to these new situations and was able to find the equilibrium in most of the games. It took longer for the algorithm to find the Nash equilibrium in the simulation with the square obstacle, but that

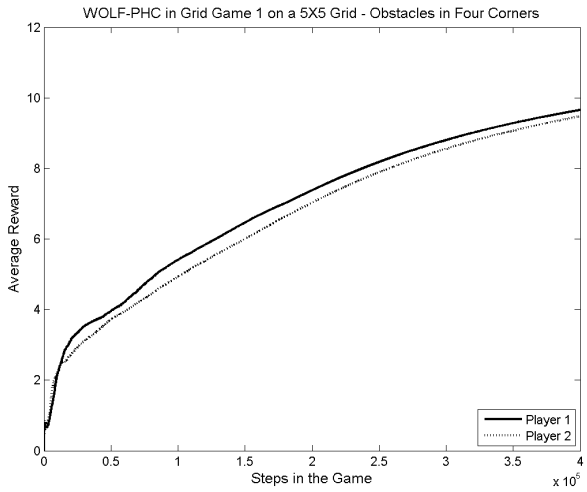


Figure 4.23: Performance of WOLF-PHC in 5X5 grid game with four obstacles

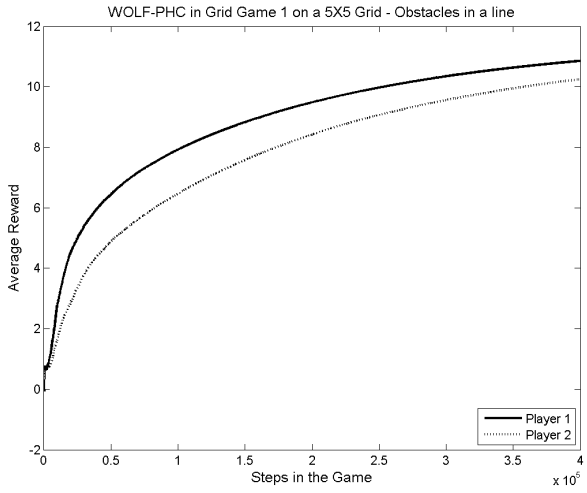


Figure 4.24: Performance of WOLF-PHC in 5X5 grid game with line in middle of the grid

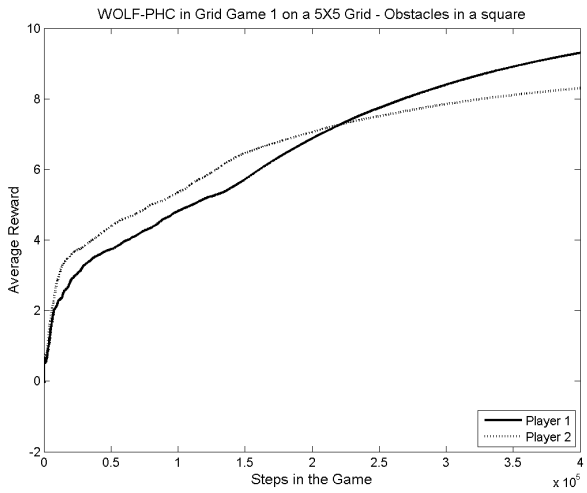


Figure 4.25: Performance of WOLF-PHC in 5X5 grid game with square obstacles

would be caused by the fact that there are only two possible Nash equilibriums. The performance of the WOLF-PHC algorithm in the last three simulations was similar to what we had with the Grid Game 1 implementation in earlier sections. Of course the average reward per steps is different because the number of steps necessary to complete the game has increased.

In general the WOLF-PHC algorithm has a lot of potential in multi-agent reinforcement learning even if it only takes into consideration its own actions. We showed that it is not a complex algorithm and that it did not use much computation power. It was also able to adapt to large environment without problem.

Chapter 5

Conclusion

In our research we reviewed in detail the evolution of Reinforcement Learning and its basic principles. We implemented three important algorithms used in multiagent reinforcement learning: The Minimax-Q algorithm, the Nash-Q algorithm and the WOLF-PHC algorithm. Our simulations were done in a general-sum stochastic grid game environment. We looked at five different aspects of reinforcement learning to compare the algorithms: Their learning method, their ability to converge to Nash equilibrium, their complexity, their computational cost and their performance during learning. We used two criteria for performance in our experiment: The average reward over time and the ability to find the Nash equilibrium strategy.

We were able to confirm that the Minimax-Q algorithm does not converge to a Nash equilibrium in a general-sum game which was already proven in [9], [17] and [11]. However, we found that the Minimax-Q algorithm does perform as well as the Nash-Q algorithm during learning. The Minimax-Q algorithm does not find the Nash equilibrium strategy but has an average reward over time during learning on par with the Nash-Q algorithm. Our version of the Minimax-Q algorithm employed a Matlab function to find the min-max values and its computation time was much longer than the other two. We would recommend a specific simplex algorithm when implementing

the Minimax-Q algorithm to minimise its computation cost.

The Nash-Q algorithm was able to find the Nash equilibrium strategy in a two-player general-sum stochastic grid game as per the literature [10], [19], [12], [27], and [8]. The Nash-Q algorithm used an explore-only method of learning, for example the learner only chooses random actions at every step of the game. A Nash-Q learner does not get better over time. Its goal is to find the Nash equilibrium strategy of the game and then that information can be used until there is a change in the environment or the opponent changes strategy. Also, the Nash-Q algorithm needs to find a Nash equilibrium at every step. As per [20], [11] and [23], the algorithm needs specific conditions to converge and it cannot be generalized. The empirical data shows that the algorithm does converge even if it does not meet its constraints. To be able to find the Nash equilibrium the learner also needs to keep track of its opponent's actions and rewards during play. This is only possible in a game where the leaner has access to the information. Our recommendation for the Nash-Q algorithm is that it is not a good choice for future work because of the learning method and the complexity of finding a Nash equilibrium at every step of learning.

The WOLF-PHC was able to out perform the two other algorithms and was able to find the Nash equilibrium in each grid game. The main difference with the other two algorithms was the fact that it is a single agent learner. It performed as per [18] and it was able to adapt to a larger game grid without problems. From our observation and implementation, we would recommend the WOLF-PHC algorithm to be used in general-sum stochastic games.

Bibliography

[1] C.J.C.H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989.

[2] C.J.C.H. Watkins and P. Dayan. *Q-learning*. Kluwer Academic Publishers.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[4] R.E. Bellman. *Dynamic Programming*. Dover Books on Mathematics. Dover, 2003. ISBN : 9780486428093.

[5] Richard W. Cottle, Jong-Shi Pang, and Richard E. Stone. *The Linear Complimentary Problem*. Computer Science and Scientific Computing. Academic Press, inc., San Diego, CA, USA, 1992.

[6] Xiaosong Lu. *Multi-Agent Reinforcement Learning in Games*. PhD thesis, Carleton University, December 2011.

[7] John Nash. Non-cooperative games. *The Annals of Mathematics*, 54(2):286–295, September 1951.

[8] Junling Hu and Michael P. Wellman. Nash q-learning for general-sum stochastic games. *Journal of Machine Learning Research 4*, pages 1039–1069, November 2003.

[9] Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 157–163. Morgan Kaufmann, 1994.

[10] Junling Hu and Michael P. Wellman. Multiagent reinforcement learning: Theoretical framework and an algorithm. In *Proceedings of the Fifteenth International Conference on Machine Learning*, ICML '98, pages 242–250, Madison, Wisconsin, USA, July 1998. Morgan Kaufmann.

[11] Michael L. Littman. Value-function reinforcement learning in markov games. *Journal of Cognitive Systems Research*, 2(1):55–66, 2001.

[12] Junling Hu and Michael P. Wellman. Experimental results on q-learning for general-sum stochastic games. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 407–414, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[13] C. E. Lemke and J. J. T. Howson. Equilibrium points of bimatrix games. *SIAM Journal on Applied Mathematics*, 12(2):413–423, 1964.

[14] David Pritchard. Game theory and algorithms. `http://www.cs.princeton.edu/~dp6/gta/`, March 2011.

[15] B. von Stengel. *Computing equilibria for two-person games*, volume 3, chapter 45, pages 1723–1759. North-Holland, Amsterdam, Holland.

[16] X. Chen and X. Deng. Settling the complexity of two-player nash equilibrium. Proceedings of the 47th IEEE Symposium on Foundations of Computer Science, pages 261–272, 2006.

[17] Michael L. Littman. Friend-or-foe q-learning in general-sum games. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01,

pages 322–328, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[18] Michael Bowling and Manuela Veloso. Rational and convergent learning in stochastic games. In *Proceedings of the 17th international joint conference on Artificial intelligence - Volume 2*, IJCAI'01, pages 1021–1026, Seattle, WA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN = 1-55860-812-5, 978-1-558-60812-2.

[19] Junling Hu. *Learning in Dynamic Noncooperative Multiagent Systems*. PhD thesis, University of Michigan, 1999.

[20] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. A comprehensive survey of multi-agent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 38(2):156–172, Mars 2008.

[21] Amy Greenwald and Keith Hall. Correlated-q learning. In *Association for the Advancement of Artificial Intelligence (AAAI) Spring Symposium*, pages 242–249. AAAI Press, 2003.

[22] Georgios Chalkiadakis. Multiagent reinforcement learning: Stochastic games with multiple players. Technical report, University of Toronto, March 2003.

[23] Michael Bowling and Manuela Veloso. An analysis of stochastic game theory for multiagent reinforcement learning. Technical report, Computer Science Department, Carnegie Mellon University, 2000.

[24] Shankar Sastry João P. Hespanha, Maria Prandini. Probabilistic pursuit-evasion games: A one-step nash approach. In *Proceedings of the 39th IEEE Conference on Decision and Control*, pages 2272–2277, Sydney, Australia, December 2000.

[25] Bernhard von Stengel. *Algorithmic Game Theory*. Cambridge University Press, New York, NY, USA, 2007.

[26] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[27] Junling Hu and Yilian Zhang. Online reinforcement learning in multiagent systems. University of Rochester.

[28] Badr Al Faiya. Learning in pursuit-evasion differential games using reinforcement fuzzy learning. Master's thesis, Carleton University, Ottawa, Ontario, February 2012.

[29] Rahul S. J. Savani. *Finding Nash Equilibria of Bimatrix Games*. PhD thesis, London School of Economics and Political Science, 2003.

[30] P.G. Balaji and D. Srinivasan. *An Introduction to Multi-Agent Systems*, pages 1–27. Springer-Verlag Berlin Heidelberg.

[31] Philip R. Cook. Limitations and extensions of the wolf-phc algorithm. Master's thesis, Brigham Young University, Provo, UT, December 2007.

[32] Yoav Shoham, Rob Powers, and Trond Grenager. Multi-agent reinforcement learning: a critical survey. Stanford University, May 2003.