Multi-Agent Systems and Large State Spaces

Yann-Michaël De Hauwere, Peter Vrancx*, and Ann Nowé

Computational Modeling Lab - Vrije Universiteit Brussel
{ydehauwe,pvrancx,anowe}@vub.ac.be

Abstract. A major challenge in multi-agent reinforcement learning remains dealing with the large state spaces typically associated with realistic multi-agent systems. As the state space grows, agent policies become more and more complex and learning slows down. The presence of possibly redundant information is one of the causes of this issue. Current single-agent techniques are already very capable of learning optimal policies in large unknown environments. When multiple agents are present however, we are challenged by an increase of the state space, which is exponential in the number of agents. A solution to this problem lies in the use of Generalized Learning Automata (GLA). In this chapter we will first demonstrate how GLA can help take the correct actions in large unknown environments. Secondly, we introduce a general framework for multi-agent learning, where learning happens on two separate layers and agents learn when to observe each other. Within this framework we introduce a new algorithm, called 2observe, which uses a GLA-approach to distinguish between high risk states where the agents have to take each others presence into account and low risk states where they can act independently. Finally we apply this algorithm to a grid-world problem because of the similarities to some real-world problems, such as autonomous robot control.

Keywords: Multi-agent learning, reinforcement learning, large state spaces.

1 Introduction

Reinforcement learning (RL) has been shown to be a powerful tool for solving single agent Markov Decision Processes (MDPs). It allows a single agent to learn a policy that maximises a possibly delayed reward signal in an initially unknown stochastic stationary environment. While basic RL techniques are not suited for problems with very large state spaces, since they rely on a tabular representation for policies and enumerate all possible state-action pairs, several extensions have been proposed to reduce the complexity of learning. The use of temporally extended actions has recently been introduced as a possible solution [1,2]. Other

* Yann-Michaël De Hauwere and Peter Vrancx are both funded by a Ph.D grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen).

A. Håkansson et al. (Eds.): Agent and Multi-agent Technology, SCI 289, pp. 181–205.
springerlink.com

methods rely on function approximators, representing the agent's policy such as neural networks, decision trees and other regression techniques.

The non-stationary environment, agents experience and the uncertainty about the other agents' goal make the problem of large state spaces even more pertinent in multi-agent systems (MAS). Relatively little work has been done, however, on extending these RL techniques for large state spaces to MAS. One possible approach is to simply let each agent independently apply a single agent technique such as Q-learning, ignoring the other agents. This approach has limited applicability however, since in a multi-agent setting the typical convergence requirements of single-agent learning no longer hold. The other extreme is to let agents learn in a joint state-action space. In the cooperative case the problem can then be cast into a single agent MDP over the joint agent actions. This method assumes that centralised control is feasible and does not scale well since the joint actions are exponential in the number of agents. In the non-cooperative case typically one tries to learn an equilibrium between agent policies [3,4]. These systems need each agent to calculate equilibria between possible joint actions in every state and as such need each agent to retain estimates over all joint actions in all states. Other equilibrium learners, which do not learn in the joint action space exist [5,6], but even these systems still suffer from the large state spaces typical in realistic agent systems.

It is clear that some form of generalisation over the state space is necessary, to solve this problem. With this generalisation, a certain amount of accuracy must be traded in for acceptable learning times and memory considerations. In 1996, Boutilier already pointed out the interest and difficulties for using structured problem representations and generalisation techniques in multiagent environments [7] after having shown the usefulness of using Bayesian Networks (BN) to compactly represent the state transition function [8]. More recently, Guestrin et al. [9] introduced an algorithm for planning in cooperative MAS, using factored value functions and a simple message passing scheme among the agents to coordinate their actions.

However, all these approaches often assume that agent actively collaborate with each other or have full knowledge of the underlying transition and reward functions. Most research done so far towards learning to aggregate states in the RL problem has focused on learning the structure of the underlying problem [10,11,12]. Using this learned model, conventional techniques such as Dynamic Programming (DP) can be used to solve the problem.

Generalized Learning Automata (GLA) are capable of learning a generalisation over a large number of states, without needing to store large amounts of samples, perform computational intensive statistical tests or build a model of the environment. We will demonstrate the use of these GLA in large unknown environments and show that multiple GLAs can aggregate over states when more than one agent is present in the system. Furthermore we will introduce an alternative solution to the problem of large state spaces. Consider the problem of

a robot trying to learn a route to a goal location. In a stationary environment the robot can simply rely on basic sensor input to explore the environment and learn a path. Suppose now that other mobile robots are added to this system. The robot must now take care to reach its goal without colliding with the other robots. However, even if the robot is provided with the exact locations of the other robots at all times, it does not make sense to always condition its actions on the locations of the others. Always accounting for the other robots means that both the state space as well as the action space are exponential in the number robots present in the system. For most tasks this joint state-action space representation includes a lot of redundant information, which slows learning without adding any benefits. For example, when learning how to reach a goal without colliding, robots need to take into account each other's actions and locations, only when it is possible to collide. When there is no risk of collision there's no point in differentiating between states based solely on the locations of other agents. Therefore we introduce a framework where agents will learn when to take the other agents into account and when it is safe to ignore their presence. This means that the robot relies on single agent techniques combined with its original sensor inputs to learn an optimal path, and coordinates with other agents only when collisions are imminent. We will use the GLA to let the agents learn when to opt for which technique. In the experiments section we demonstrate how our approach delivers significant savings, in both memory requirements as well as convergence time, while still learning a good solution.

Throughout this chapter we will begin by giving the necessary background information in Reinforcement Learning (RL) in Section 2 and multi-agent learning in Section 3 before presenting our work with GLAs in Sections 4 and 5. We conclude the chapter with a short discussion and guidelines for future work.

2 Reinforcement Learning

In this section we will explain some of the basic concepts of Reinforcement Learning (RL) and will elaborate in detail on one of the most well-known algorithms in this field: Q-learning.

Reinforcement learning solves the problem an agent encounters when he has to find a solution to a given situation. *Without some feedback about what is good and what is bad, the agent will have no grounds for deciding which move to make* [13]. Because of the absence of a teacher, telling which actions the agent must take, he must discover on its own, which actions yield the highest reward or reinforcement.

By repeatedly chosing random actions in the different game situations, the agent will eventually be able to build a predictive model of the environment and thus forsee the outcome of its actions. Note that this model is not necessarily an explicit one.

Furthermore, the rewards can either be immediate, i.e. the reward (or penalty) is granted immediately after an agent performed his action, for instance in the

n-armed bandit problem[1], or the reward can be delayed for a number of actions like in Chess for example, where the reward is given at the end of the game. So it is possible that an agent must sacrifice short-term gains for larger long-term gains [14,15].

The main advantage of reinforcement learning is its independence from an external supervisor. This means that it is able to learn in non-stationary environments using random actions to explore the environment. The main goal of reinforcement learning is to maximise the received reward. To do so, a trade-off must be made between exploring new actions and exploiting the actions found in the past with positive outcome. This trade-off is commonly known as the exploration-exploitation dilemma. We will not elaborate any further on this, but refer to the book by Sutton and Barto [14].

2.1 Markov Decision Processes

In a reinforcement learning system four elements can be distinguished next to the agent and the environment:

- a *policy* π_t. This is a function that maps states to actions, i.e., $\pi_t : S \rightarrow A$, where S is the set of states of the environment, and A is the set of actions available to the agent. $\pi_t(\sigma_t, a_t)$ denotes the probability that action a_t is chosen in state σ_t at time step t. This is the most important element of the reinforcement learning system because a policy alone is sufficient to determine the behaviour of an agent.
- a *reward function* R gives the quality of the actions chosen by the agent. It maps a state and an action to a real value (reward or penalty), i.e., $R : S \times A \rightarrow \mathbb{R}$. The reward signals are used by the agent for altering its policy.
- a *value function*, denoted $V(\sigma)$, is a mapping from a state to the reward an agent can expect to accumulate, when starting from that state σ. So this results in a long-term expected reward, whereas the reward function returns the immediate reward of a state. In reinforcement learning this is the function that agents will try to learn, as it is this function that will be used to obtain the highest reward.
- a *predictive model* of the environment. This will predict the behaviour of the environment (the next state and reward) when performing some action.

In the standard reinforcement learning model, depicted in Figure 1 proposed in [16], an agent interacts with its environment. This model consists of

- a discrete set of environment states S,
- a discrete set of actions from the agent A,
- a set of reinforcement signals R.

[1] In the n-armed bandit problem the agent is faced repeatedly with a choice among n different options, or actions. After each choice the agent receives a numerical reward chosen from a stationary probability distribution dependent on the action it selected. The objective is to maximize the expected total reward over some time period.

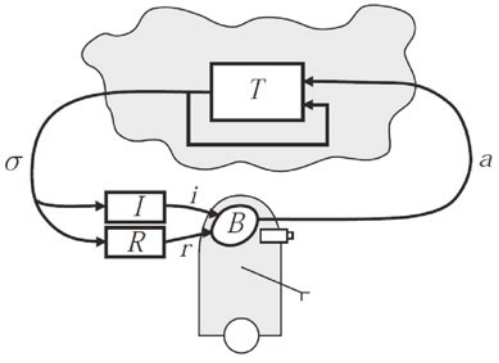


Fig. 1. The standard reinforcement learning model

On each time step the agent receives some information i about the current state s of the environment. Based on this information the agent chooses some action a. This action may change the state of the environment and the next state is communicated to the agent through a scalar reinforcement signal r. The agent will adapt his behaviour B in order to maximise the long-run measure of reinforcement.

A model for problems that are characterised by delayed rewards can be given as a *Markov Decision Process* (MDP). This brief overview is based on [14] and [16].

Definition 1. *A **Markov Decision Process** is a 4-tuple (S, A, T, R), where:*

- *S is the state space,*
- *A is the action space,*
- *T is the state transition function, $T : S \times A \rightarrow \Pi(S)$, mapping states and actions on probabilities of making a transition to a next state. We will use $T(\sigma, a, \sigma')$ to represent the probability of reaching state σ' from σ after performing action a.*
- *R is the reward function, $R : S \times A \rightarrow \mathbb{R}$, mapping states and actions to a real numerical value.*

A Markov Decision Process has the following Markov property:

Definition 2. *A system is said to possess the **Markov property** if the system's state transitions are independent of any previous environment states or actions taken, more formally:*

$$T(\sigma_{t+1}|\sigma_t, a_t, \sigma_{t-1}, a_{t-1}, ..., \sigma_1, a_1) = T(\sigma_{t+1}|\sigma_t, a_t).$$

This property ensures that an agent can behave optimally by only observing its current state. It doesn't matter which actions were taken before, so no history needs to be stored. An agent behaving optimal, is often achieved through optimal value functions, which are defined in the following section.

2.2 Learning in MDPs

The optimal value of a state is defined as follows:

Definition 3. *The **optimal value of a state** σ_t is the expected infinite discounted sum of rewards the agent will receive when it starts in that state and follows the optimal policy from there, i.e.* $V^*(\sigma_t) = max_\pi E\left(\sum_{k=0}^{\infty} \gamma^k r_{t+1+k}\right)$.

An optimal policy is a solution to MDP's. It specifies which action should be chosen for every state. A policy is denoted as π, and the recommended action for state σ is $\pi(\sigma)$. An optimal policy is a policy that yields the highest expected reward. We use π^* to denote an optimal policy.

Values can be seen as predictions for future rewards. Often, actions are chosen based on these values, so one of the most important aspects of finding a solution to a reinforcement learning problem, is finding a method for estimating future rewards. In the following paragraphs two algorithms will be presented for calculating optimal policies. Both algorithms find their origin in Dynamic Programming (DP) techniques[2] as these are well known solutions for solving MDPs.

The idea of Dynamic Programming is to convert the famous Bellman equations (1957), given below in Equation 1, into update rules for improving approximations of the desired value function.

$$V^{\pi}(\sigma) = E\left[\sum_{k=0}^{\infty} \gamma^k r_{t+1+k} | \pi, \sigma_t = \sigma\right] \tag{1}$$

The optimal value function V^*, with π^* the optimal policy, are the solutions of the Bellman Optimality Equation (2) and are given in Equations 3 and 4.

$$\begin{aligned} V^*(\sigma) &= \max_{\pi} E\left(\sum_{k=0}^{\infty} \gamma^k r_{t+1+k}\right) && (2)\\ &= \max_{a} E\left(r_{t+1} + \gamma V^*(\sigma_{t+1}) | \sigma_t = \sigma, a_t = a\right) \\ &= \max_{a}\left[R(\sigma, a) + \gamma \sum_{\sigma' \in \mathcal{S}} T(\sigma, a, \sigma') V^*(\sigma')\right], \forall \sigma \in \mathcal{S} && (3)\\ \pi^* &= arg\max_{a}\left(R(\sigma, a) + \gamma \sum_{\sigma' \in \mathcal{S}} T(\sigma, a, \sigma') V^*(\sigma')\right) && (4) \end{aligned}$$

For an arbitrary policy π, the state-value function V_π can be defined as:

$$V_{\pi}(\sigma) = E\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+1+k}\right\}$$

[2] Dynamic Programming is the brainchild of the American Mathematician Richard Ernest Bellman, who described the way of solving problems where you need to find the best decisions one after another.

$$= E\left\{r_{t+1} + \gamma V_\pi(\sigma_{t+1}) | \sigma_t = \sigma\right\}$$
$$= \sum_a \pi(\sigma, a) \left[R(\sigma, a) + \gamma \sum_{\sigma' \in \mathcal{S}} T(\sigma, a, \sigma') V_\pi(\sigma') \right]$$

By improving V_π iteratively, V^* can be approximated. $V_\pi(\sigma)$-function itself can be obtained using successive approximations called iterative policy evaluation:

$$V_{k+1}(\sigma) = \sum_a \pi(\sigma, a) \left[R(\sigma, a) + \gamma \sum_{\sigma' \in (S)} T(\sigma, a, \sigma') V_k(\sigma') \right] \quad (5)$$

The ***policy iteration*** algorithm will manipulate a policy directly in order to improve it, based on the state-values that correspond to the current policy π. The algorithm is given below:

Algorithm 1. Policy Iteration

choose an arbitrary policy π
repeat
  $\pi \leftarrow \pi'$
  compute the value function V_π using iterative policy evaluation (Equation 5):
  improve the policy π at each state and store it in: π'
    $\pi'(\sigma) \leftarrow \text{argmax}_a(R(\sigma, a) + \gamma \sum_{\sigma' in \mathcal{S}} T(\sigma, a, \sigma') V_\pi(\sigma'))$
until $\pi = \pi'$

To improve the policy, the best action a in a state σ is identified, based on the current state values. So the policy π is improved in state σ by updating $\pi(\sigma)$ into the action that maximises the right hand side of Equation 5, resulting in a better policy π'. We thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \rightarrow V_{\pi 0} \rightarrow \pi_1 \rightarrow V_{\pi 1} \rightarrow ... \rightarrow \pi^* \rightarrow V^*$$

A finite MDP has only a finite number of distinct policies, at most $|A|^{|S|}$, and the sequence improves at each step, so this algorithm terminates in at most an exponential number of iterations. This means, that this process is bound to result in an optimal policy and an optimal value function, when no further improvements are possible.

The basic idea behind the ***value-iteration*** algorithm is to find the optimal value function $V^*(\sigma)$. The difference with policy iteration is that in this algorithm, no explicit policy evaluation will take place at each time step. This policy evaluation process is truncated, without losing convergence guarantees. This one-step truncated policy evaluation is given in Equation 6. The value iteration algorithm is given in Algorithm 2.

$$V_{k+1}(\sigma) = \max_a E\{r_{t+1} + \gamma V_k(\sigma_{t+1}|\sigma_t = \sigma, a_t = a\}$$
$$= \max_a \left[R(\sigma, a) + \gamma \sum_{\sigma' \in S} T(\sigma, a, \sigma') V_k(\sigma') \right] \quad (6)$$

Algorithm 2. Value Iteration

initialise $V(\sigma)$ arbitrarily
repeat
  for all $\sigma \in S$ **do**
    for all $a \in A$ **do**
      $Q(\sigma, a) \leftarrow R(\sigma, a) + \gamma \sum_{\sigma' \in \mathcal{S}} T(\sigma, a, \sigma') V_k(\sigma')$
      $V(\sigma) \leftarrow \max_a Q(\sigma, a)$
    end for
  end for
until Policy is good enough

As can be seen from the pseudocode, it is not strictly determined when to stop the outer loop. However this can be handled by the following condition: *If the maximum difference between two successive value functions is less than* ϵ*, then the value of the policy differs from the value function of the optimal policy by no more than* $2\epsilon\gamma/(1-\gamma)$. This provides an effective stopping criterion for the algorithm.

It is very common to reinforcement learning problems that the transition function or the reward function are unknown to the reinforcement learner. As such, it is hard to learn the optimal policy. So the problem here is, which evaluation function should be used. Let us define the evaluation function Q so that its value is the maximum discounted cumulative reward that can be achieved starting from state σ and applying action a as the first action.

$$Q(\sigma, a) \equiv r(\sigma, a) + \gamma V^*(\delta(\sigma, a)) \quad (7)$$

Note that with this equation, we can define the optimal policy as

$$\pi^*(\sigma) = \operatorname*{argmax}_a Q(\sigma, a) \quad (8)$$

and

$$V^*(\sigma) = \max_{a'} Q(\sigma, a') \quad (9)$$

and thus we have the following recursive definition for Q:

$$Q(\sigma, a) = r(\sigma, a) + \gamma \max_{a'} Q(\delta(\sigma, a), a') \quad (10)$$

So in order to find the optimal policy, one has to find this Q-function. Watkins described an algorithm to iteratively approximate it. In the ***Q-learning*** algorithm [17] a large table consisting of state-action pairs is stored. Each entry

contains the value for $\hat{Q}(\sigma, a)$ which is the learner's current hypothesis about the actual value of $Q(\sigma, a)$. The $\hat{Q}$-values are updated accordingly to following update rule:

$$\hat{Q}(\sigma, a) \leftarrow (1 - \alpha_t)\hat{Q}(\sigma, a) + \alpha_t[r + \gamma \max_{a'} \hat{Q}(\sigma', a')] \tag{11}$$

where α_t is the learning rate at time step t. Which gives us more formally the following algorithm:

Algorithm 3. Q-Learning

For each $\langle \sigma, a \rangle$ pair, initialise the table entry for $\hat{Q}(\sigma, a)$ to zero.
Observe the current state σ
loop
- Select an action a and execute it
- Receive immediate reward r
- Observe the new state σ'
- Update the table entry accordingly to equation 11
- $\sigma \leftarrow \sigma'$

end loop

This algorithm was an inspiration for a whole range of new RL-algorithms and extensions. In the following section we will discuss some of the current work around RL for large state spaces before moving on to a multi-agent setting.

2.3 Reinforcement Learning in Large State Spaces

In the previous section we explained the basic Q-learning algorithm, as proposed by Watkins in 1992. The major drawback of this algorithm is that it relies on a tabular representation of all state-action pairs in the environment. It is clear that, as the environment grows due to increasing demands in detail and complexity, simply enumerating all these states is no longer feasible.

Several techniques have been proposed to deal with these ever increasing demands. These techniques typically rely on some type of generalisation method, which transfers knowledge between similar states (or state-action pairs). One way of doing this is by representing Q-values or policies by function approximators [18].

Other techniques use adaptive resolution methods. Here the learning agent uses statistic tests to determine when a greater granularity in state space representation is needed. One example of this kind of system is the G-learning algorithm [19], which uses decision trees to learn partitions of the state space. This technique assumes that a state is represented by a number of discrete valued variables. Starting out with a single state the technique keeps statistics on Q-values corresponding to the values of each variable. When it finds a significant difference in Q-values for different values of a variable, the decision tree is split based on this variable. The process is then repeated for the new leaves of the tree.

State information is often not presented in an optimal way and may contain a lot of redundant and/or useless information which slows the learning process down. A more efficient way is to represent the state information as a set of random variables $X = \{X_1, \ldots, X_n\}$, where every state variable X_i can take values in a finite domain $Dom(X_i)$ and every possible state in the system corresponds to a value assignment $x_i \in Dom(X_i)$ for every state variable, X_i. Such a system is called a Factored Markov Decision Process (FMDP) [8,7,20].

The transition function in such a system is described by a *Dynamic Bayesian Network (DBN)*. This is a two-layer directed acyclic graph where the nodes are $\{X_1, \ldots, X_n, X'_1, \ldots, X'_1\}$. In this graph, the parents of X'_i are denoted by $Parents_a(X'_i)$. With every node $X'_i \in G_a$, a Conditional Probability Distribution $CPD^a_{X_i}(X'_i|Parents_a(X'_i))$ is associated quantifying the DBN. This method benefits from the dependencies that exist (or don't exist) between the variables of the network.

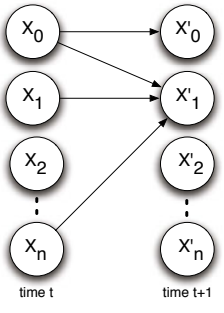


Fig. 2. DBN for action a_i

Figure 2 shows the dependencies between the variables at time t and time $t+1$ when action a_i is executed. According to this network, the Parents of X'_1 are X_0, X_1 and X_n.

When the algorithm is offered state information as such it can try to learn the state transition function and construct a model of the system, before applying planning algorithms on this model. This technique is commonly known as the DYNA-architecture [21] and has found its way to the FMDPs in [10]. Even without explicitely learning the model of the problem at hand, such a structured representation offers advantages to aggregate states and exploit the underlying structure of the system. One of the techniques that takes advantage of a structured representation are the GLA which we discuss in Section 4.

3 Multi-Agent Reinforcement Learning

In the previous two sections we explained some of the work done in RL and large state spaces. This section will focus on environments where multiple agents are present. Due to the presence of these agents, a large state space is inherently common to a MAS.

3.1 Markov Games

In the previous section we explained how a MDP is one of the key elements to solving RL problems. In MAS we can not use this model anymore. The Markov property fails, because there are different agents, changing the dynamics of the world. An extension of the single agent Markov decision problems (MDP's) to the multi-agent case can be defined by Markov Games [22]. In a Markov Game, actions are the joint result of multiple agents choosing an action seperately. Formally the Markov Game can be represented by the 5-tuple: $M = \langle \mathbb{A}, \{A\}_{\forall i \in \mathbb{A}}, \mathbb{S}, T, \mathbb{R} \rangle$, where:

- $\mathbb{A}$ is the set of agents participating in the game,
- $\{A_i\}_{\forall i \in \mathbb{A}}$ is the set of actions available to agent i,
- $\mathbb{S}$ is the set of states
- $T(s, \overrightarrow{a}, s)$ is the transition function stating the probability that a joint-action a will lead the agents from state s to state s',
- and $R_i : S \times A_1 \times \ldots A_{|\mathbb{A}|} \rightarrow \mathbb{R}$ is the reward function denoting the reward agent i gets for performing a joint action in the current state.

Note that the reward function R_i is now individual to each agent i. Different agents can receive different rewards for the same state transition. Since each agent i has its own individual reward function, defining a solution concept becomes more complex. Due to the existence of different reward functions, it is in general impossible to find an optimal policy for all agents. Instead, equilibrium points are sought. In an equilibrium, no agent can improve its reward by changing its policy if all other agents keep their policy fixed.

A special case of the general Markov game framework is given by the Multi-agent Markov Decision Process (MMDP) [23]. In this case, the Markov game is purely cooperative and all agents share the same reward function. This specialisation allows us to define the optimal policy as the joint agent policy, which maximises the payoff of all agents.

Because the agents share the same transition and reward function, one can think of the collection of agents being a single super agent with joint actions at its disposal and whose goal is to learn the optimal policy for the joint MDP. Since the agents' individual action choices may be jointly suboptimal, the added problem in MMDP's is for the agents to learn to coordinate their actions so that joint optimality is achieved. This model suffers from two major problems. Since all agents receive the same reward signal, it is not possible to apply this technique in situations where the agents have conflicting interest. The other drawback of this technique is that the states and actions of the MMDP increase exponentially in the number of agents, and thus it is not suitable for systems that already suffer from a large state space.

In Table 1 we present of a comparison of the state space complexity of some of the commonly used techniques in multi-agent learning.We compare a set of independent Q-learners to the MMDP-framework and the multi-agent extensions of Q-Learning, Nash and Correlated Q-Learning. The difference between these latter two and the MMDP lies in the fact that the agents still learn independently

when using the Q-Learning variants. The major drawback of these two techniques is that they require a full observability about the actions and rewards of all the agents. On top of this, every agent stores not only his own Q table, but also a Q table for every other agent in the system. Even tough these techniques are a vast improvement over the MMDP, which also requires all the agents to select a joint-action, computationaly these techniques are not an improvement at all.

Table 1. Space complexity of different MAS techniques in function of the number of states, actions and agents

	Independent Q-learning	MMDPs	Nash Q-Learning	Correlated Q-Learning
#states	linear	linear	linear	linear
#actions	polynomial	polynomial	polynomial	polynomial
#agents	n.a.	exponential	exponential	exponential

In the following sections we will describe our solution in detail, which is computationally much less intensive than Nash and Correlated Q-Learning and whose space complexity is the same as independent Q-Learners without the need of communication at every time step.

4 GLAs for Multi-Agent Learning

In this section we introduce our approach for scalable multi-agent learning. The method is similar to the ideas used in single agent reinforcement learning for large state space, in that we are trying to generalise policies over similar states. In multi-agent systems additional problems arise, however. Since system control is distributed over multiple agents with possibly conflicting goals, coordination between agents is required. Furthermore, in addition to the large state space typically associated with realistic agent systems, the action space now also grows exponentially in the number of agents. This motivates the need for generalisation in the action space as well as the state space.

As a basic tool for our approach we use simple pattern matching units called Generalized learning automata. These automata have the advantage that they are computationally simple, can be combined into larger networks to offer greater flexibility and have theoretical foundations [24].

In the first instance we use GLAs to learn similar regions in the state space. A policy can then be represented by a mapping from each region to an action, rather than a mapping from each individual state.

In the next section we extend this approach to include adaptive state space resolution and action generalisations. Now the GLAs learn regions in the state space, but rather than directly selecting an action, they choose between different strategies. In state space regions were conflicts are possible the GLAs select multi-agent techniques in order to coordinate between agents, in other regions single agent strategies with limited state space visibility are used and agents effectively ignore each other.

4.1 Generalized Learning Automata

A GLA is an associative reinforcement learning unit. The purpose of a GLA is to learn a mapping from given inputs or contexts to actions. At each time step the GLA receives an input which describes the current system state. Based on this input and its own internal state the unit then selects an action. This action serves as input to the environment, which in turn produces a response for the GLA. Based on this response the GLA then updates its internal state.

Formally, a GLA can be represented by a tuple (X, A, β, u, g, U), where X is the set of inputs to the GLA and $A = \{a_1, \ldots, a_r\}$ is the set of outputs or actions the GLA can produce. $\beta \in [0, 1]$ again denotes the feedback the automaton receives for an action. The real vector u represents the internal state of the unit. It is used in conjunction with the probability g to determine the action probabilities, given an input $x \in X$:

$$P\{a(t) = a|u, x\} = g(x, a, u) \tag{12}$$

where g has to satisfy following conditions:

$$\begin{aligned} g(x, a, u) &\geq 0 \; \forall x, a, u \\ \sum_a g(x, a, u) &= 1 \quad \forall x, u \end{aligned}$$

U is a learning algorithm which updates $\mathbf{u}$, based on the current value of $\mathbf{u}$, the given input, the selected action and response β. In this paper we use a modified version of the REINFORCE [25] update scheme. In vector notation this update scheme can be described as follows:

$$\begin{aligned} \mathbf{u}(t+1) = \mathbf{u}(t) + \lambda\beta(t)\tfrac{\delta ln\, g}{\delta \mathbf{u}}(\mathbf{x}(t), a(t), \mathbf{h}(\mathbf{u}(t))) \\ +\lambda\mathbf{K}(\mathbf{h}(\mathbf{u}(t)) - \mathbf{u}(t)) \end{aligned} \tag{13}$$

where $\mathbf{h}(\mathbf{u}) = [h_1(u_1), h_2(u_2), \ldots h_r(u_r)]$, with each h_i defined as:

$$h_i(\eta) = \begin{cases} L_i & \eta \geq L_i \\ 0 & |\eta| \leq L_i \\ -L_i & \eta \leq -Li \end{cases} \tag{14}$$

In this update scheme λ is the learning rate and $L_i, K_i > 0$ are constants. The update scheme can be explained as follows. The first term added to the parameters is a gradient following term, which allows the system to locally optimise the action probabilities. The next term uses the $h_i(u)$ functions to keep parameters u_i bounded within predetermined boundaries $[-L_i, L_i]$. This term is added since the original REINFORCE algorithm can give rise to unbounded behavior. In [26] it is shown, that the adapted algorithm described above, converges to local maxima of $f(\mathbf{u}) = E[\beta|\mathbf{u}]$, showing that the automata find a local maximum over the mappings that can be represented by the internal state in combination with the function g.

Originally these systems were proposed for classification problems, in which the context vectors represent features of objects to be classified and the GLA output represents class labels. We propose to use the same techniques in factored MMDPs. In such a system each agent internally uses a set of GLA to learn the different regions in the state space where different actions are optimal.

We use the following set-up for the GLA. With every action $a_i \in A$ the automaton can perform, it associates a vector $\mathbf{u_i}$. This results in an internal state vector $\mathbf{u} = [\mathbf{u_1}^\tau \dots \mathbf{u_r}^\tau]$ (where τ denotes the transpose). With this state vector we use the Boltzmann distribution as probability generating function:

$$g(x, a_i, \mathbf{u}) = \frac{e^{\frac{x^\tau u_i(a_i)}{T}}}{\sum_j e^{\frac{x^\tau u_j(a_i)}{T}}} \tag{15}$$

with T a parameter that represents the temperature. Of course, since this function is fixed in advance and the environment in general is not known, we have no guarantee that the GLA can represent the optimal mapping. For instance, when using the function given in Equation 15 with a 2-action GLA, the internal state vector represents a hyperplane. This plane separates context vectors which give a higher probability to action 1 from those which action 2. If the sets of context vectors where different actions are optimal, are not linearly separable the GLA cannot learn an optimal mapping.

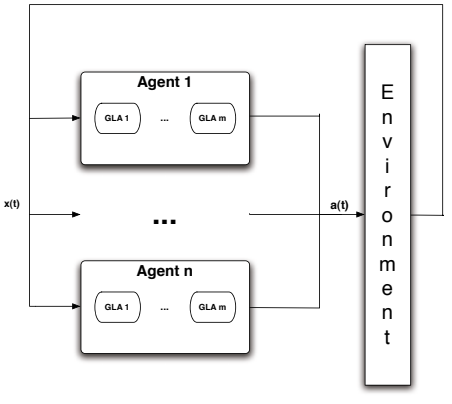


Fig. 3. Learning set-up. Each agent receives factored state representation as input. GLA decide action to be performed.

To allow a learner to better represent the desired mapping from context vectors to actions, we can utilise systems composed of multiple GLA units. For instance the output of multiple 2-action GLAs can be combined to allow learners to build a piecewise linear approximation of regions in the space of context vectors. In general, we can use systems which are composed of feedforward structured networks of GLA. In these networks, automata on one level use actions

of the automata on the previous level as inputs. If the feedforward condition is satisfied, meaning that the input of a LA does not depend on its own output, convergence to local optima can still be established [24].

Figure 3 shows the general agent learning set-up. Each time step t a vector $\mathbf{x}(\mathbf{t})$ giving a factored representation of the current system state is generated. This vector is given to each individual agent as input. The agents internally use a set of GLA to select an action corresponding to the current state. The joint action $\mathbf{a}(\mathbf{t})$of all agents serves as input to the environment, which responds with a feedback $\beta(t)$ that agents use to update the GLA. One of the main advantages of this approach is that convergence guarantees exist for general feedforward GLA structures. In the common interest problems under study in this paper, a group of agents each internally using one or more GLA can be viewed as a single large network of GLA, thus ensuring convergence to a local optimum.

What follows are a demonstration of the capabilities of GLA in a number of relatively simple experiments. Our basic experimental set-up is shown in Figure 4. Two agents A and B move on a line between $[-1, 1]$. Each time step both agents select action *left (L)* or *right (R)*, move and then receive a reward based on their original joint location and the joint action they chose. Each agent then updates using only the reward signal and the joint location, without any knowledge of the action selected by the other agent.

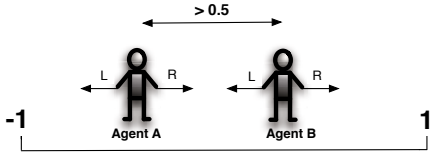


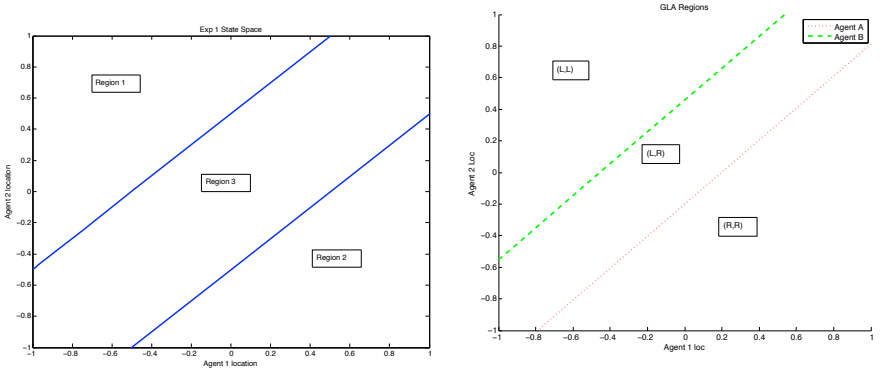
Fig. 4. Experimental set-up. Two agents move around on a line between positions -1 and 1. Each time step both agents take a step left or right.

4.2 Experimental Results

Agents using a single GLA. In this experiment the state space is divided in three regions, as shown in Figure 5(a). In region 1 Agent A is left of Agent B. In the second, Agent A is to the right of Agent B. The third region encapsulates all the states where the absolute value of the distance between the two agents is less than 0.5. Each agent has two possible actions, i.e. *Left* or *Right*. The reward scheme is as follows:

1. Region 1: A reward of $+1$ is given, when both agents choose action *Left*, 0 otherwise.
2. Region 2: A reward of $+1$ is given, when both agents choose action *Right*, 0 otherwise.
3. Region 3: A reward of $+1$ is given, when both agents move apart from each other, 0 otherwise.

For this experiment each agent uses a single GLA with 2 actions corresponding to the agent actions L and R. Each time step we give both agents an input vector $\mathbf{x} = [x1\ x2\ 1]$, where $x1$ is the position of agent A and $x2$ is the position of agent B. The GLA use a vector $\mathbf{u_i} = [u_{i1}\, u_{i2}\, u_{i3}]$ for each action i. The learning process of a GLA can then be seen as moving the line $(\mathbf{u_1} - \mathbf{u_2})^\intercal \mathbf{x}$ which separates regions in the state space where the GLA prefers action 1(L) from those where it prefers action 2(R). Typical results obtained with this system can be seen in Figure 5(b). This result was obtained running the experiment for 100.000 iterations. Each iteration consists of a single action choice and update for both agents. After each move and subsequent learning update, the agents were reset to new random positions and the game was restarted. This was done to avoid the undersampling problem which occurs easily when dealing witch such large state spaces.



(*a*) (*b*)

Fig. 5. Experiment 1. (a) State space regions for experiment 1. (b) Typical result learnt by GLA. Lines separate regions where agents prefer different actions. Joint actions with highest probability are given in each region. Parameter settings where $\lambda = 0.01$,$K_i = L_i = 1$, $T = 0.5$.

Since GLA take context vectors as input, it is possible to present the state information in different forms to the agent. Figure 6 shows a comparison of the average reward obtained, with three distinct ways of information. We compared the use of the joint location described above, to an absolute distance metric $(AbsoluteValue(Pos(AgentA) - Pos(AgentB)))$ and a deictic distance metric $(Pos(AgentA) - Pos(AgentB))$. This experiment was run without tuning of the exploration of the Boltzmann action selection method, so these values are not necessarily measures for optimal performance of the GLA, but rather serve as a criterion to compare the influence of the information given in the context vectors.

The absolute distance metric clearly performs the worst due to the inability of making a distinction between different positions of the other agent. When presenting the agent with a deictic information to the position of the other agent, it outperforms agents using a joint location based state information. We

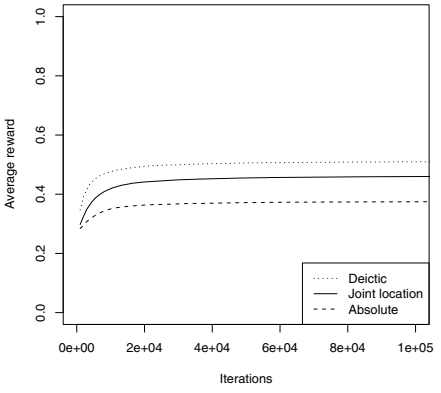


Fig. 6. Comparison of the influence of the state information given to the GLA

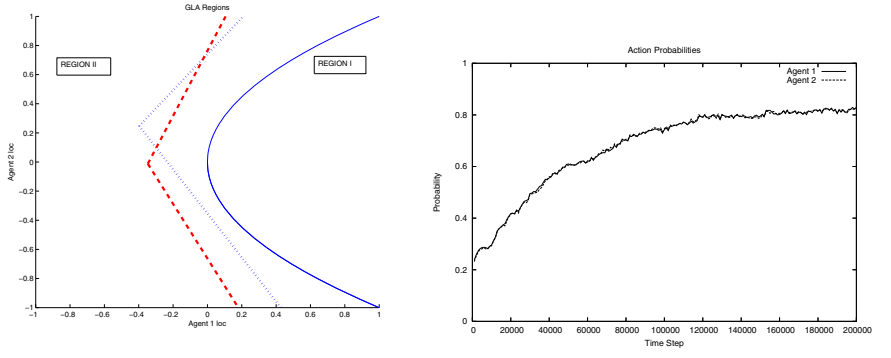
performed this experiment to show that, even though the same information is used, presenting it in different forms to the agent clearly benefits the learning results.

Agents Using Multiple GLA. In the second experiment we examine a situation where the different regions in the state space are not linearly separable. In such a case the agents cannot exactly represent the optimal mapping, but rather have to approximate it using hyperplanes. We use the same set-up as in the previous experiment, but now we consider two regions, as given in Figure 7(a). In region I, given by the inside of the parabola action (L, L) is optimal with a reward of 0.9. When the joint location of the agents falls outside the parabola, however, action (R, R) is optimal with reward 0.5. In both cases all other joint actions have a pay-off of 0.1.

Both agents use a system consisting of 2 GLA, connected by an AND operation. Both GLA have 2 actions: 0 and 1. If the automata both choose 1 the agents performs its first action L else it performs action R. Figure 7(a) shows 2 typical results for the boundaries that the agents learn to approximate the parabola. Figure 7(b) shows for both agents the evolution of probability of the optimal action L in region I. The probabilities in this plot where obtained by generating 100 points in the region with uniform probability and calculating the average probability over these points.

While it can be seen from the results in Figure 7 that the agents are able to approximate the desired regions, this experiment also demonstrated the limits of our approach. As was mentioned in the previous section the GLA are only guaranteed to converge to a local optimum. This means that the agent can get stuck in suboptimal solutions. Such a situation was observed when the reward for the optimal action in region II is increased. In this case it is possible for the agents to get stuck in a situation where they both always prefer the optimal action for region II and neither agent has a good approximation of the region inside the parabola. Since the rewards of both agents are based on their joint

action, no agent can get out of this situation on its own. The agents can only improve their pay-off by switching actions inside region I together. In such a situation with multiple local optima, the final result obtained by the agents is depended on their initialisation.



(a) (b)

Fig. 7. Experimental results for the second experiment. (a) Typical results for approximations for parabola learnt by agents. (b) Probabilities of optimal action in region I for both agents (average over 100 runs). Parameter settings where $\lambda = 0.005$,$K_i = L_i = 1$, $T = 0.5$.

5 Decoupling the Learning Processes

As demonstrated in the previous sections, learning in multi-agent systems is a cumbersome task. Choices have to be made wether to observe agents, or to communicate or even both. Most of these techniques make a very black and white decision for these choices, i.e. they either always observe all other agents or they never do. In many tasks however it is often enough to observe the other agents only in very specific cases. We propose to decouple the multi-agent learning process in two separate layers. One layer will learn when it is necessary to observe the other agent and select the appropriate technique. The other layer contains a single agent learning technique, to be used when their is no risk of influence of the other agent, and a multi-agent technique, when the agents will influence each other. Figure 8 shows a graphical representation of this framework.

5.1 First Level

The first layer of this framework is the important one, as this one decides the performance of the entire algorithm. A Q-learner could be installed on this level, taking the joint location as input, and choosing between the two techniques on the second level. This would however hardly be an improvement over a standard MMDP. Kok and Vlassis introduced a somewhat similar technique, where they would learn single agent when there is no need to coordinate, and learn using a

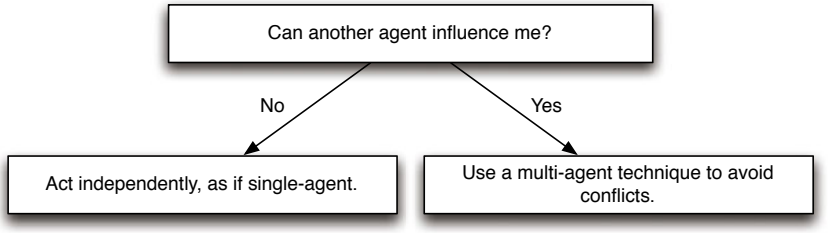


Fig. 8. Decoupling the learning process by learning when to take the other agent into account on one level, and acting on the second level

MMDP when they need to coordinate explicitely [27]. In this technique, called sparse tabular multi-agent Q-learning the list of states in which the agents had to be coordinated, needed to be defined explicitly beforehand. In our approach the main idea is that these states are learned as a function of the joint state. This gives us the benefit that we can generalise over states and just define danger zones as a function and not an explicit collection of individual states. It is for this reason that in our implementation of this framework we use *Generalized Learning Automata* (GLA).

Algorithm 4 contains the pseudo code of an algorithm that has been developed within this framework. It uses a GLA on the top level to decide if agents are within each others range of inference. If they are, a simple coordination mechanism is used for the next time step in order to avoid collisions. If the GLA judges that the agents will not interfere with each other in the next time step, both agents use Q-learning to select the action for the next time step.

Algorithm 4.

For each agent and every $\langle \sigma, a \rangle$ pair, initialise the table entry for $\hat{Q}(\sigma, a)$ to zero.
Observe the current state σ and the distance between the agents δ
loop
  if both agents decide to coordinate, based on δ **then**
    Agents coordinate on action to take to avoid a collision
  else
    Agents select action a, based on their action selection strategy
  end if
  • Execute the chosen actions
  • Receive immediate reward r_q and r_c, where r_q is the reward for the actions the agents chose and q_c is the reward for their coordination selection.
  • Observe the new state σ'
  • Update the Q-table entry accordingly to equation 11
  • $\sigma \leftarrow \sigma'$
  • Update the GLAs with r_c
end loop

The main advantage of using GLA is that the first layer of the algorithm can learn to determine which technique on the second level must be chosen, without explicitly storing estimates or samples of visited state-action pairs. All necessary information is encoded in the parameters of the GLA, which are typically much smaller in number than the states about which information needs to be kept. The possibility of combining GLA into a larger network which can learn more complex distinctions, also gives us the flexibility to adapt the complexity of the first layer to the needs of the problem at hand.

5.2 Second Level

At the second level of the learning algorithm two possible methods can be used, depending on the outcome of the first layer of the algorithm. If this first layer determines that other agents can safely be ignored a single agent technique is used, else agents use another technique that takes the other agents into account.

In this paper the single agent technique we use is an independent Q-learner. When it is deemed safe to do so agents act as if alone in the system, completely ignoring all other agents. Here we also assume that the states of our problem are represented by a set of state variables, some of which describe the local states of other agents (see Section 5.3 for an example). When learning in single agent mode, these superfluous variables are ignored, further reducing the size of the state-action space.

The multi-agent technique we adopt is a simple form of coordination through communication. If agents decide to coordinate their actions they first observe if a collision will occur if both agents would just choose their preferred action. If this is the case, one agent is selected randomly to perform its action, while the other selects a random other action. If no collision will occur, both agents can play their preferred action. Many other techniques are of course possible. Depending on the setting agents could try to bargain, learn an equilibrium policy or try to exploit each other.

In the following section we give some results of this algorithm and compare it to other RL algorithms.

5.3 Experimental Results

We test our approach by applying it in a gridworld game and test its performance against single agent Q-learning with independent agents and within the MMDP framework, using the joint location as the state. Even tough the problem seems an easy one, it contains all the difficulties of much harder problems and is widely used in the RL community [14]. Our gridworld represents a very simplistic version of a warehouse, where agents moving on tracks, have to get orders from the different aisles. Some of these aisles have only one track, so agents have to pass through them sequentially. Figure 9 shows a graphical representation of a gridworld where such problems are present. The two agents, in the upper and lower left corners, both have to reach the goal, G, which is surrounded by walls.

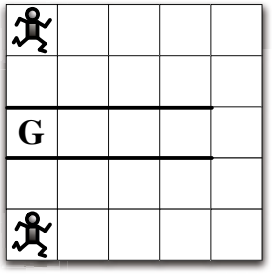


Fig. 9. Simple multi-agent gridworld problem

They have four actions at their disposal (N,E,S,W) for going up, right, down and left respectively. If both agents would use the shortest path to the goal, without considering the other agents, they would collide at the entrance of the passageway to the goal.

Before giving the results of the different techniques, we analyse the state-action spaces used by the different approaches. The independent Q-learners do not take into account any information about the other agents learn resulting in a state space consisting of only 25 states with 4 actions each. The joint state learners learn in a state space represented by the joint locations of the agents resulting in $(25)^2 = 625$ states, but select their actions independently, so they have 4 actions each. The MMDP learner also learns in the joint state space of the agents but with 16 actions (all possible combinations of the 4 individual actions). Our approach uses the same state action space as the independent Q-learners, unless the first level GLAs indicate, that a conflict is possible. In this case the agent communicate in order to select a safe action for next time step.

Since GLAs are able to learn regions in the state space, the actual size of their state space is not relevant.

All experiments were run with a learning rate of 0.05 for the Q-learners and Q-values were initialised to zero. An ϵ-greedy action selection strategy was used, where ϵ was set to 0.9. The GLA have a learning rate of 0.01, use a boltzmann action selection strategy and were initialised randomly. All experiments were run for 200.000 iterations, where an iteration is the time needed for both agents to reach the goal, starting from their initial positions. If an agent reached the goal, it receives a reward of +100. For the MMDP learner the reward of +100 was given when both agents reach the goal position, but once an agent is in the goal, its actions no longer matter, since the goal is an absorbing state. If an agent collided with another agent, it was penalised by −10. Bumping into a wall was also penalised by −1. For every other move, the reward was zero. For every collision, wether it was with a wall or with another agent, the agent is bounced back to its original position. The GLA were rewarded individually according to Figure 10.

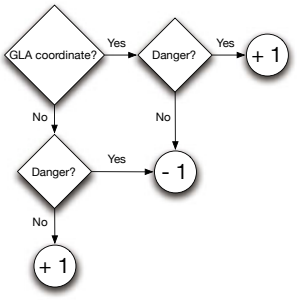
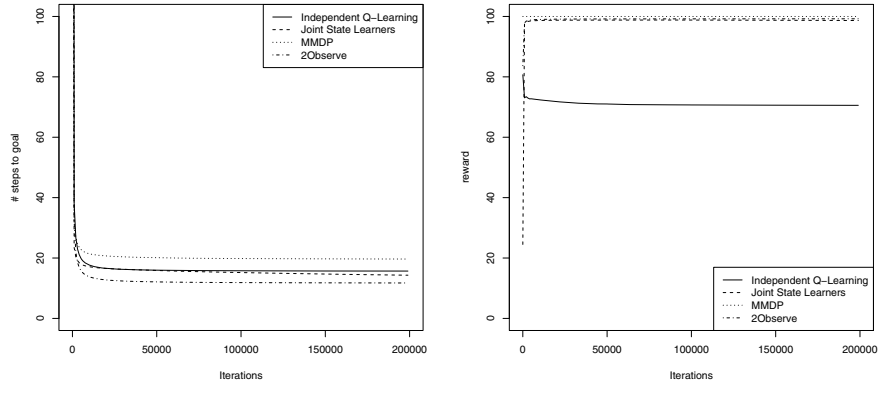


Fig. 10. Rewards for the GLA of the 2observe algorithm



(*a*) (*b*)

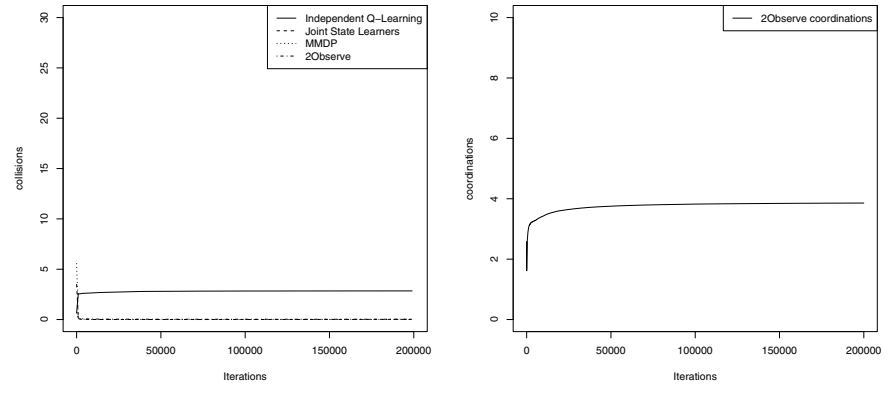
Fig. 11. (a) Average number of steps from start position to goal position and (b) the average reward of both agents for the different techniques

If on top of this, both agents decided to coordinate when there was no need, they are penalised according to following rule: $0-(\delta/10)$ where δ is the distance between both agents.

Figure 11(a) shows the average number of steps both agents needed to reach the goal, as well as the average reward they collected in Figure 11(b). Our technique converges not only to a better solution in terms of the number of steps to the goal, but also converges faster than both other techniques. When looking in terms of the reward, we see that our technique performs almost as good as the MMDP and much better than the independent Q-Learners. The reason for the quite poor performance of the latter can be seen in Figure 12(a).

Both multi-agent approaches find a solution with zero collisions relatively fast, whereas the independent learners never do. They converge to an average of three collisions per iteration. This means that they can only converge to a maximum reward of 70. In Figure 12(b) we illustrate that our approach converges to an average of approximately four coordination actions per iterations. This means

that our agents only communicate explicitely four times per iteration, about which action to choose.



(*a*) (*b*)

Fig. 12. (a) Average number of collisions and (b) the average number of coordinations per iteration

6 Discussion and Future Work

When dealing with large state spaces and the presence of multiple agents, the learning task becomes difficult. In this chapter we introduced two solutions for these problems. First we demonstrated how GLAs can be used to aggregate states in unknown environments and showed the importance of the way the state information was presented to the agent. Second, we introduced a general framework for learning in multi-agent environments based on a separation of the learning process in two levels. The top level will learn when agent's need to observe each others' presence and activate the appropriate technique on the second level. If no risk of interference is present, the agents can use a single agent technique, completely ignoring all the other agents in the environment. If the risk of interfering with each other is true, a multi-agent technique will be activated in order to deal with this increased complexity.

The main advantage of this framework is that it can be used as a foundation for using existing single-agent and multi-agent techniques, adapting the learning process wherever needed.

We implemented a concrete instantiation of this framework, called 2observe, which uses a generalized learning automaton on the top level, a Q-learner for the case where agents do not interfere with one another and a simple communication based coordination mechanism when the agents need to take each others' presence into account. We showed empirically that our technique was able to

reach a solution, almost as good as MMDP learners but without the need of full observation and in a much smaller state space.

The possibilities for future work are wide. Many techniques exist to incorporate in our framework. On the second level, the entire range of single agent and multi-agent techniques can be used. On the first level also many alternatives exist. We chose to use GLA in this paper due to their simplicity and low computational costs without the need to store previously seen samples. However, appropriate statistical tests could be used on this level to measure the influence two agents have on each other. Another interesting research track is to use the rewards given on the second level as feedback for the first level. This would mean that the GLA could learn from delayed rewards using a monte-carlo updating scheme. In this way, a wider range of problems can be solved and state information can be used even more wisely.

References

1. Sutton, R.S., Precup, D., Singh, S.P.: Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. Artificial Intelligence 112(1-2), 181–211 (1999)
2. Stolle, M., Precup, D.: Learning options in reinforcement learning. In: Koenig, S., Holte, R.C. (eds.) SARA 2002. LNCS (LNAI), vol. 2371, pp. 212–223. Springer, Heidelberg (2002)
3. Hu, J., Wellman, M.P.: Nash q-learning for general-sum stochastic games. J. Mach. Learn. Res. 4, 1039–1069 (2003)
4. Greenwald, A., Hall, K.: Correlated-q learning. In: AAAI Spring Symposium, pp. 242–249. AAAI Press, Menlo Park (2003)
5. Vrancx, P., Verbeeck, K., Nowe, A.: Decentralized learning in markov games. IEEE Transactions on Systems, Man and Cybernetics (Part B: Cybernetics) 38(4), 976–981 (2008)
6. Vrancx, P., Verbeeck, K., Nowé, A.: Optimal convergence in multi-agent mdps. In: Apolloni, B., Howlett, R.J., Jain, L. (eds.) KES 2007, Part III. LNCS (LNAI), vol. 4694, pp. 107–114. Springer, Heidelberg (2007)
7. Boutilier, C.: Planning, learning and coordination in multiagent decision processes. In: Theoretical Aspects of Rationality and Knowledge, pp. 195–201 (1996)
8. Boutilier, C., Dearden, R., Goldszmidt, M.: Exploiting structure in policy construction. In: Mellish, C. (ed.) Proceedings of the 14th International Joint Conference on Artificial Intelligence, pp. 1104–1111. Morgan Kaufmann, San Francisco (1995)
9. Guestrin, C., Koller, D., Parr, R.: Multiagent planning with factored mdps. In: 14th Neural Information Processing Systems, NIPS-14 (2001)
10. Degris, T., Sigaud, O., Wuillemin, P.H.: Learning the structure of factored markov decision processes in reinforcement learning problems. In: Proceedings of the 23rd International Conference on Machine learning, New York, NY, USA, pp. 257–264 (2006)
11. Strehl, A.L., Diuk, C., Littman, M.L.: Efficient structure learning in factored-state mdps. In: AAAI, pp. 645–650. AAAI Press, Menlo Park (2007)
12. Abbeel, P., Koller, D., Ng, A.Y.: Learning factor graphs in polynomial time and sample complexity. Journal of Machine Learning Research 7, 1743–1788 (2006)

13. Russel, S., Norvig, P.: Artificial Intelligence, a Modern Approach. Prentice-Hall, Englewood Cliffs (1995)
14. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998)
15. Mitchell, T.: Machine Learning. McGraw-Hill Companies, New York (1997)
16. Kaelbling, L.P., Littman, M.L., Moore, A.P.: Reinforcement learning: A survey. Journal of Artificial Intelligence Research 4, 237–285 (1996)
17. Watkins, C.: Learning from Delayed Rewards. PhD thesis, University of Cambridge (1989)
18. Lin, L.: Programming robots using reinforcement learning and teaching. In: Proceedings of AAAI, vol. 91, pp. 781–786 (1991)
19. Chapman, D., Kaelbling, L.: Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In: Proceedings of the 12th International Joint Conference on Artificial Intelligence, pp. 726–731 (1991)
20. Guestrin, C., Hauskrecht, M., Kveton, B.: Solving factored mdps with continuous and discrete variables. In: Proceedings of the 20th conference on Uncertainty in artificial intelligence, pp. 235–242 (2004)
21. Sutton, R.S.: Reinforcement learning architectures. In: Proceedings ISKIT 1992 International Symposium on Neural Information Processing (1992)
22. Shapley, L.: Stochastic Games. Proceedings of the National Academy of Sciences 39, 1095–1100 (1953)
23. Claus, C., Boutilier, C.: The dynamics of reinforcement learning in cooperative multiagent systems. In: Proceedings of the Fifteenth National Conference on Artificial Intelligence, pp. 746–752. AAAI Press, Menlo Park (1998)
24. Phansalkar, V., Thathachar, M.: Local and global optimization algorithms for generalized learning automata. Neural Computation 7, 950–973 (1995)
25. Williams, R.: Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. Reinforcement Learning 8, 229–256 (1992)
26. Thathachar, M., Sastry, P.: Networks of Learning Automata: Techniques for Online Stochastic Optimization. Kluwer Academic Pub., Dordrecht (2004)
27. Kok, J.R., Vlassis, N.: Sparse tabular multiagent Q-learning. In: Nowé, A., Tom Lenaerts, K.S. (eds.) Proceedings of the Annual Machine Learning Conference of Belgium and the Netherlands, Brussels, Belgium, pp. 65–71 (January 2004)