Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL 2007)

The Effect of Bootstrapping in Multi-Automata Reinforcement Learning

Maarten Peeters, Katja Verbeeck and Ann Nowé
Vrije Universiteit Brussel
Computational Modeling Lab
Pleinlaan 2
1050 Brussel
Email: mjpeeter@vub.ac.be, kaverbee@vub.ac.be, ann.nowe@vub.ac.be

Abstract— **Learning Automata are shown to be an excellent tool for creating learning multi-agent systems. Most algorithms used in current automata research expect the environment to end in an explicit end-stage. In this end-stage the rewards are given to the learning automata (i.e. Monte Carlo updating). This is however unfeasible in sequential decision problems with infinite horizon where no such end-stage exists. In this paper we propose a new algorithm based on one-step returns that uses bootstrapping to find good equilibrium paths in multi-stage games.**

I. Introduction

The research on the learning behaviors of automata started with the work of Tsetlin [1] in the 1960's. In his research Tsetslin and co-workers, introduced finite action deterministic automata in stationary random environments. It was shown that under certain conditions the automata behaved asymptotically optimal. Further research [2]–[4] looked at more challenging problems such as non-deterministic environments and variable-structure, continuous action learning automata. Learning automata (LA) research led to many practical applications in the engineering field.

Recently, learning automata became also popular in the field of Multi-Agent Reinforcement Learning [5]. Early on, researchers looked at how multiple automata in a single environment could be interconnected and still find stable solutions [6]. One of the advantages of using learning automata in this field is that they operate without information concerning the number of other participants or their strategies, which is a problem for many multi-agent reinforcement learning techniques. In [7], results on learning automata games formed the basis for a new multi-agent reinforcement learning approach to learning single stage, repeated normal form games. Many real-world problems, however, are naturally translated into multi-stage problems. Therefore in this paper we are concerned with learning a sequence of games.

Thathachar and Ramakrishnan [8], [9] introduced the concept of a hierarchical LA in which the actions where distributed over a tree-structured hierarchy of LA. In such a hierarchy different actions have to be taken before an explicit end-stage is reached. Games between hierarchical learning automata agents can then be represented by multi-stage games or multi-agent Markov decision problems [10]. In [11] it was shown that hierarchical learning automata agents can solve (certain) multi-stage games with episodic tasks by using Monte Carlo rewards constructed along the path of the multi-stage game. However for now only episodic tasks could be considered.

Standard single agent reinforcement learning techniques, such as Q-learning [12], which are by nature constructed to solve sequential decision problems, use the mechanism of bootstrapping to handle non-episodic tasks. Bootstrapping means that values or estimates are learned on the basis of other estimates [6]. The use of estimates eliminates the need for episodic tasks where rewards are only given in explicit end-states.

Multi-agent learning approaches that bootstrap do exists; an overview of approaches based on Q-learning and through it on the Bellman equations is given in [13]. However in these approaches, agents cannot be independent anymore, they need to know the actions taken by other agents and their associated rewards to learn joint Q-values. Besides this, only weak convergence assurances are shown.

In this paper, we introduce bootstrapping for independent hierachical LA agents in multi-stage games. The learning automata in the hierarchy will be updated on the basis of estimates, and these estimates will be propagated from child to parent in the LA hierarchy. Just as in single agent learning, we can develop updates going from one-step back-up mechansims to the complete Monte Carlo update mechanism. Empirical results show that hierarchical LA agents based on one-step backups are capable of solving multi-stage games in a non-episodic way. Furthermore they do this with less communication compared to the Monte Carlo updating. The results also show a higher percentage of convergence to the optimal path in large multi-stage games and a faster convergence.

In the next section we repeat the concept of bootstrapping in classical single agent reinforcement learning. We continue in Section 3 with multi-stage games, the LA model and its properties. In Section 4 we explain how we can add boostrapping to our learning automata model. In Section 5 we discuss the effect of boostrapping on several multi-stage games, including games with a coordination problem and large random generated multi-stage games. In the last section we conclude.

II. Bootstrapping in Single-Agent Learning

Reinforcement learning is the problem faced by an agent that learns behavior through trial-and-error interactions with a dynamic environment, see [6]. To obtain a lot of reward from the environment it is operating in, the agent should exploit actions that it has learned in the past and that were found to be good. However discovering better actions is only possible by trying out new ones, thus exploring. This trade-off is fundamental and in the stationary case convergence to the optimal policy[1] can be guaranteed by different classes of methods.

Single agent control problems in stationary environments are successfully modeled as Markov decision processes (MDPs). An MDP is defined by a set of states S, a set of actions A, a transition function[2] $T : S \times A \rightarrow P(S)$ that outputs a probability distribution on S and a reward function $R : S \times A \rightarrow P(\mathbb{R})$ which implicitly specifies the agent's task.

Without prior knowledge of the transition probabilities or rewards, an MDP can be solved online by the theory of Reinforcement Learning [6].

Common reinforcement learning methods, which can be found in [6], [14] are structured around estimating value functions. A value of a state or state-action pair, is the total amount of reward an agent can expect to accumulate over the future, starting from that state. One way to find the optimal policy is to find the optimal value function. If a perfect model of the environment as a Markov decision process is known, the optimal value function can be learned with an algorithm called value iteration. An adaptive version of this algorithm exists for situations were a model of the environment is not known in advance.

One feature used for distinguishing algorithms is by their degree of bootstrapping in their evaluation process. Following the generalized policy iteration framework of [6] a method for solving the RL problem can be described by 2 simultaneous interacting processes, namely policy improvement and policy evaluation. In the evaluation phase the algorithm tries to learn values for a state or state action-pair for the current policy. In the improvement phase the current policy is being improved with respect to the values learned. The evaluation process can be accomplished by bootstrapping, meaning that values or estimates are learned on the basis of other estimates.

For instance the Q-learning algorithm, which is an adaptive value iteration method (see [6], [15]) bootstraps its estimate for the state-action value $Q_{t+1}(s, a)$ at time $t+1$ upon its estimate for $Q_t(s', a')$ with s' the state where the learner arrives after taking action a in state s:

$$Q_{t+1}(s, a) \leftarrow (1 - \alpha)Q_t(s, a) + \alpha(r_t + \gamma\ max_{a'}Q_t(s', a')) \quad (1)$$

With α the usual step size parameter, γ a discount factor and r_t the immediate reinforcement.

[1]A policy is a mapping from states to actions. An optimal policy is a mapping which maximizes some long-run measure of reinforcement.

[2]This function models the probability of ending up in a next state when an agent takes an action in a certain state.

Non-bootstrapping evaluation methods as Monte Carlo methods update their estimates based on actual returns only. For instance the *every-visit Monte Carlo* method updates a state-action value $Q(s, a)$ at time $t + n$ (with n the time for one episode to finish) based on the actual return R_t and the previous value:

$$Q_{t+n}(s, a) \leftarrow (1 - \alpha)Q_t(s, a) + \alpha R_t$$

with

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{T-t-1} r_T$$

and t is the time at which (s, a) occurred.

Methods that learn their estimates in part on the basis of other estimates,(i.e. they bootstrap) are called Temporal Difference Learning Methods. The Q-learning algorithm (equation 1) can be classified as a TD(0) algorithm. The back-up for each state is based on just the next reward, an estimation of the remaining rewards is given by the value of the state one step later. One says that Q-learning is therefore a one-step TD method. However, one could also consider backups based on any intermediate number of rewards:

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma V_t(s_{t+n}) \quad (2)$$

In the limit, all real rewards up-until-termination are used and no bootstrapping is necessary, this is the Monte Carlo method. So, there is a spectrum ranging from using simple one-step returns to using full-backup returns. However, in general, backups can then also be done toward any average of n-step returns, as shown in Figure 1.

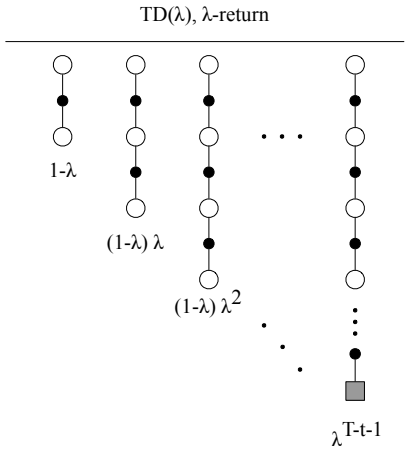

Fig. 1. The backup diagram for $TD(\lambda)$ [6]. If $\lambda = 0$ all the weight is put on the first reward (= immediate reward). However if $\lambda = 1$, the backup corresponds to the Monte Carlo backup.

Methods using these complex backup schemes are denoted as TD(λ) methods, where parameter λ weights the contributions of the n-step returns $R_t^{(n)}$.

$$R_t^{\lambda} = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)}$$

In a sense, TD(λ) methods form a bridge from simple one-step TD(0) methods to Monte Carlo methods. What is presented here is the so-called theoretical or forward view of TD(λ) methods. An equivalent backward view exists, which allows for an on-line implementation.

Advantages of TD methods over Monte Carlo methods include the fact that TD methods can be naturally implemented in an on-line, fully incremental fashion. With Monte Carlo methods only off-line updating is possible, since only at the end of an episode the actual full return is known. This makes the latter method also unsuited for contuining tasks. Moreover, TD methods learn from each transition, which can sometimes speed-up learning time.

Convergence results for the control problem exists in the extreme bootstrapping cases, meaning that estimates can only be based on current estimates. An elegant proof of convergence for Q-learning is given in [15] in case the problem is Markovian. Monte Carlo methods have asymptotic convergence guarantees for the evaluation problem however not for the control problem.

III. Multi-agent Learning in Multi-Stage Games

A. Multi-Stage Game

A multi-stage game is a game where the participating agents have to take a sequence of actions. An MDP can be extended to the multi-agent case, a Multi-agent Markov decision process (MMDP). Formally we can define an MMDP as a five-tuple, $M = \langle \mathbb{A}, \{A_i\}_{\forall i \in \mathbb{A}}, \mathbb{S}, T, R \rangle$ where:

- $\mathbb{A}$ is the set of agents participating in the game,
- $\{A_i\}_{i \in \mathbb{A}}$ is the sets of actions available to agent i,
- S is the set of states (same as defined with an MDP),
- $T(s, \vec{a}, s')$ is the transition function stating the probability that a joint-action $\vec{a}$ will lead the agents from state s to state s',
- and $R : S \rightarrow \mathbb{R}$ is the reward function denoting the reward the agents get for entering a certain state.

In the remainder of this paper we limited ourselves to tree-structured multi-stage games. This means that there are no loops possible between the game stages and once branches are separated their paths will never be joined again.

An example of such a tree-structured multi-stage game can be seen in Figure 2. In this particular example, the MMDP consists of 6 states. The game starts in state s_1. Here, both agents have to take an action resulting in the joint-action (a_i^1, b_j^1). Based on this joint-action the game continues to either state s_2 or s_3 (both states give a numerical feedback of 0). In this second stage, again, both agents must choose an action a_k^2 and b_l^2. If the agents are in state s_3 no matter how the joint-action (a_k^2, b_l^2) looks like, the agent will always end up in state s_6 resulting in a payoff of 0.75 (chance of receiving reward 1) for both agents. However if the agents ended up in state s_2 after the first stage then the agents have to deal with a coordination problem. If the agents can coordinate on either joint-actions (a_1^2, b_1^2) or (a_2^2, b_2^2) they both receive a pay-off of 1.0 (chance of receiving reward 1). Miscoordination on the other hand will be penalized heavily with a penalty of 0.0 (chance of receiving reward 1). When the agents end up in state s_4, s_5 or s_6 the game ends. Based on the rewards r_m obtained in process of reaching an end state a weighted reward can be formed: $r_{total} = \theta_1 r_1 + \cdots + \theta_s r_s$. The weights can be adjusted to completely in- or exclude a reward at a certain stage. In our example it would be useful to exclude the reward given at the first stage since this is always 0.0. After the reward is given, the agents will be transported back to the startstate s_1 and the game can restart.

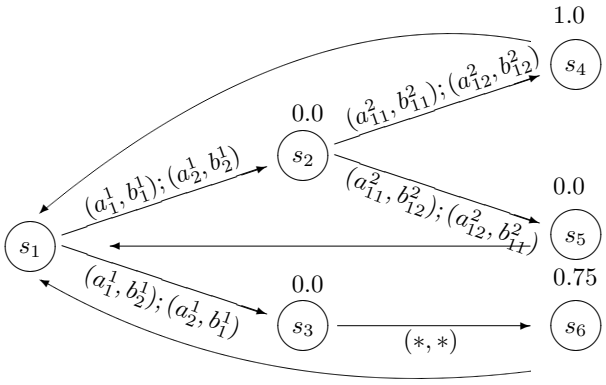

Fig. 2. An example of a simple MMPD, the Opt-In Opt-Out game [10]

We can view a multi-stage game as a sequence of single state games. The reward matrices for the 2 stages of the game of Figure 2 are given in Figure 3. Note that in this multistage decision problem, we have a coordination problem at the second stage. Also note that the second matrix is divided into 4 separate sub-matrices. We do this to depict that when the agents reach the second stage, they don't play the complete game, only a part of it. Which part is decided by the actions in the first stage. For instance if at the first stage, both agents select their first actions, in the second stage the sub-matrix in the upper-left corner is played. If the agents in the first stage both play their second action, the sub-matrix in the lower-right corner is activated in the second stage and so on.

$$M^1 = \begin{pmatrix} 0.0 & 0.0 \\ 0.0 & 0.0 \end{pmatrix} \quad M^2 = \left(\begin{array}{cc|cc} 1.0 & 0.0 & 0.75 & 0.75 \\ 0.0 & 1.0 & 0.75 & 0.75 \\ \hline 0.75 & 0.75 & 1.0 & 0.0 \\ 0.75 & 0.75 & 0.0 & 1.0 \end{array}\right)$$

Fig. 3. Reward matrices of the Opt-In Opt-Out multistage decision problem from Figure 2 with the rewards scaled to the interval $[0, 1]$.

B. Learning Automata Model

A learning automaton is an independent entity that is situated in a random environment and is capable of taking actions autonomously. The random environment is responsible for generating a scalar value indicating the quality of the action taken by the learning automaton. This scalar value, which we call reward, is then fed back into the learning automaton.

Let us first describe the environment formally. An environment is a triple $\langle A, \mathbf{c}, \mathbf{R} \rangle$ with A the possible sets of inputs

into the environment, $\mathbf{c} = [c(1) \dots c(n)]$ the penalty vector storing a chance of success (= $c(i)$) for each action i, and $\mathbf{R}$ the set of possible scalar rewards the environment can generate (based on $a_t \in A$ the action taken at time step t and $c(a_t)$ the probability of success for that action). Depending on the output set, we can identify three different environments. If $\mathbf{R} \in \{0, 1\}$, we say the environment is of the P-model. If the input is taken from a finite number of values in the closed interval $[0, 1]$, the environment is called a Q-model. And finally, if the reward value is an arbitrary number in the interval $[0, 1]$ it is a S-model. In this paper, we use the Q-model. Our results however, can be transfered to any of the two other models without loss of generality.

A learning automaton can be expressed as a quadruple $\langle A, \mathbf{R}, \mathbf{p}, U \rangle$. $A = \{a(1), \dots, a(n)\}$ denotes the set of actions the learning automaton can take. $R_t(a_t(i)) \in \mathbf{R}$ is the input that is given to the LA to indicate the quality of the chosen action. Note that we deliberately reused the symbols A and $\mathbf{R}$ because the output from the environment is the input into the LA and vice versa.

The probabilities for selecting action $a_t(i)$ are stored in the vector $\mathbf{p_t} = [p_t(1), \dots, p_t(n)]$. The restrictions on $p_t(i)$ are the following:

$$\sum_{i=1}^{n} p_t(i) = 1 \quad \text{and} \quad 0 \leq p_t(i) \leq 1.$$

Thus all the probabilities sum up to 1 and each probability lays within the interval $[0, 1]$. Note that at the beginning of the game all action probabilities are chosen equal:

$$p_0(1) = p_0(1) = \dots = p_0(n) = \frac{1}{n}.$$

Each iteration the action probabilities get updated by the reinforcement obtained from the environment. For the experiments in this paper, we used the Linear Reward-Inaction [3] scheme. Let $a_t = a(i)$ be the action chosen at time step t. Then the action probability vector $\mathbf{p}$ is updated according to

$$p_{t+1}(i) = p_t(i) + \alpha r_t(1 - p_t(i)) \quad (3)$$

$$p_{t+1}(j) = p_t(j) - \alpha r_t p_t(j), \forall j \neq i. \quad (4)$$

with α the step size parameter. The L_{R-I} has been studied widely and has several nice properties such as ϵ-*optimality* and *absolute expediency*. For more details we refer to [2]–[4].

C. Hierarchies of Learning Automata

One problem often reoccurring in reinforcement learning is that the agents learn too slow. One way to tackle this problem was given by Thathachar and Sastry [16] and Ramakrishnan [9]. They constructed an agent by connecting multiple learning automata in a tree structured hierarchy and distributing the total action set between the automata at the bottom level (see Figure 4).

A hierarchical LA works as follows. The first automaton that is active is the root at the top of the hierarchy: LA. This automaton selects one of its n actions. If for example the automaton selects action 2, the learning automaton that will become active is learning automaton $LA2$. Then this active learning automaton is eligible for selecting an action. Based on this action, another learning automaton on the next level will become active. This process repeats itself until one of the learning automata at the bottom of the hierarchy is reached.

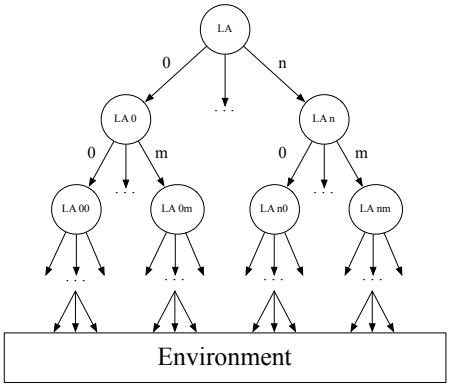

Fig. 4. An agent constructed in with a hierarchy of learning automata

Using a hierarchy of learning automata is especially useful when dealing with large actions spaces. One of the advantages is computational. Suppose that the number of actions at level l is n_l and there are r levels in the hierarchy. The total number of probabilities that get updated is $n_1 + n_2 + \dots + n_r$. If we had used a single automaton, this automaton would have had $n_1 \times n_2 \times \dots \times n_r$ actions and each all these actions would have to be updated each iteration.

D. An interaction between learning automata hierarchies

The interaction of the two hierarchical agents of Figure 5 goes as follows. At the top level (or in the first stage) Agent 1 and Agent 2 meet each other in the stochastic game. They both take an action using their top level learning automata LA A and LA B. Performing actions a_i by LA A and b_k by LA B is equivalent to choosing automata LA A_i and LA B_j to take actions at the next level. The response of environment E_1, $r_t \in \{0, 1\}$, is a success or failure, where the probability of success is given by c^1_{ik} . At the second level the learning automata LA A_i and LA B_k choose their actions a_{ij} and b_{kl} respectively and these will elicit a response from environment of which the probability of getting a positive reward is given by $c^2_{ij,kl}$. At the end of the episode all the automata which were involved in the games, update their action selection probabilities based on the actions performed and the responses of the environments.

IV. Bootstrapping with Learning Automata

A. Monte Carlo

In the Monte Carlo method, the updating of the probabilities is based on averaged sample returns. This averaged return is ideally generated at the end of an episode. Monte Carlo methods thus work best in episodic tasks where eventually each strategy leads to a clear end state. Each time such a clear end state is reached, an averaged return is generated by

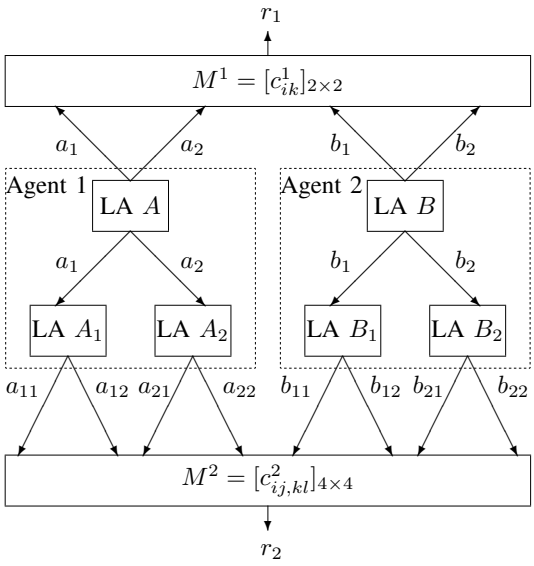

Fig. 5. An interaction of two agents constructed by learning automata hierarchies. The top-level automata play a single stage game and produce a reward r_1. Then one learning automata of each hierarchy at the second level play another single stage game, resulting in reward r_2.

creating a weighted sum of all the returns. This sum is then given to all learning automata that were active during the last episode so they can update their action probabilities. Thus when we reach an end stage at time step t we generate the following sum:

$$R = \theta_1 r_1 + \theta_2 r_2 + \cdots + \theta_t r_t$$

where r_i is the reward generated at time step i. Note that the following restrictions apply on the weights

$$\sum_{i=1}^{t} \theta_i = 1 \quad \text{and} \quad 0 \leq \theta_i \leq 1.$$

These restrictions ensure that the averaged reward is within the [0,1] interval, a condition necessary for the linear reward-inaction update scheme.

In [17] the authors proof that if all the automata of the hierarchical learning automata update their action probabilities at each stage using the L_{R-I} update scheme and if the composite reward is constructed as a Monte Carlo reward (described above) and at each level the step sizes of the automata are chosen sufficiently small then the expected payoff of the overall system is non-decreasing. This result means that the hierarchical learning automata will always converge to an equilibrium path in an identical pay-off multi-stage game. To which path the automata will converge to however is not known. Neither is known how (sub-)optimal this path is. This largely depends on the initial settings of the action probability distribution and on the step size used.

B. Forward Monte Carlo

In [18], [19] we introduced an update mechansim based on Intermediate Rewards (which we call Forward Monte Carlo in this paper). With this technique the learning automata at level l only get informed about the immediate reward and the rewards on the remainder of the path. It doesn't get informed about the rewards that are given to automata on the levels above him because he has no direct influence over them and they would clutter up his combined reward. While the rewards are still given at an explicit end-stages and while the automata receive less rewards compared to the traditional Monte Carlo technique, we still were able to construct a theoretical proof that hierarchical learning automata using the Forward Monte Carlo technique will converge to an equilibrium path in an identical pay-off multi-stage game (under the same conditions we described above for the traditional Monte Carlo technique).

The complete algorithm can be found in Algorithm 1. When rewards are used unweighted (only normalized to the interval $[0, 1]$), this technique based on intermediate rewards coincides with the Monte Carlo version of the forward $TD(\lambda)$ view, [6]. Because the learning automata get updated at the end of an episode, the Forward Monte Carlo technique is still an off-line algorithm.

Algorithm 1 Forward Monte Carlo

1: All the learning automata: initialise action probabilities: $\forall a \in A : p_0(a) = \frac{1}{|A|}$
2: **for** each trial **do**
3: Activate the top LA of the hierarchies
4: **for** each level l in hierarchy h **do**
5: The active learning automata take action $a^l_t(h)$
6: $\Rightarrow$ joint-action $\mathbf{a} = [a^l_t(1), \ldots, a^l_t(h), \ldots]$
7: Store immediate reward r_t (team reward based on **a**)
8: **end for**
9: **for** each level l in hierarchy h **do**
10: Compute combined reward
11: $R^l(h) = \theta_t r_t + \theta_{t+1} r_{t+1} + \cdots + \theta_T r_T$
12: Update the automaton that was active at level l in hierarchy h with reward $R^l(h)$
13: **end for**
14: **end for**

C. One-Step Estimates

In the One-Step Estimates technique, the updating of the learning automata will no longer take place at an explicit end-stage. The automata get informed immediately about the local reward they receive for their actions. Each automaton has estimates about the long term reward for each of its actions. These estimates are updated by combing the local rewards with an estimate of possible rewards that this action might give on the remainder of the path (see Line 10 and 11 in Algorithm 2). The behavior of the algorithm is controlled by four parameters: α, ρ, σ and τ. α is the step size parameter from Eq. (1)-(2). σ is used to normalize the combined reward

Algorithm 2 One-Step Estimates

1: All the learning automata: Initialise action probabilities: $\forall a \in A : p_0(a) = \frac{1}{|A|}$
2: Initialise my-estimates: $\forall a \in A : myest(a) = 0$ (estimates for all my actions, meaning: what is the long term reward associated with this action)
3: Initialise children-estimates: $\forall a \in A : est_0(a) = 0$ (estimates for all my children, meaning: what is the average of the long term rewards for the learning automaton associated with this action)
4: **for** each trial **do**
5: Activate the top LA of the hierachies
6: **for** each level l in hierarchy h **do**
7: Take action $a_t^l(h)$
8: $\Rightarrow$ joint-action $\mathbf{a} = [a_t^l(1), \ldots, a_t^l(h), \ldots]$
9: Observe immediate reward r_t (reward based on **a**)
10: Compute $R_t = \sigma r_t + (1-\sigma) est(a_t^l(h))$
11: Update my-estimates: $myest(a_t^l(h)) = myest(a_t^l(h)) + \rho[R_t - myest(a_t^l(h))]$
12: Update action probability $\mathbf{p}$ using $myest(a_t^l(h))$ as the reward for the L_{R-I} scheme
13: Propagate $myest(a_t^l(h))$ up to parent $\Rightarrow$ *see Algorithm 3: Updating estimates*
14: **end for**
15: **end for**

to the interval $[0, 1]$, it can also determine whether the weight is on the local rewards or on the long term estimates (see Line 10 in Algorithm 2). ρ controls the influence of the difference between the combined reward and the old-estimate on the new-estimate (see Line 11 in Algorithm 2). And finally τ is used for normalized the estimates received from the children (see Line 2 in Algorithm 3).

Algorithm 3 Updating estimates

1: κ is the estimate received from the child (the last action this automaton took was $a_t^l(h)$)
2: $est(a_t^l(h)) \leftarrow \tau est(a_t^l(h)) + (1-\tau)\kappa$

Currently, we only considered one-step returns. In a similar way we could extend the algorithm to a 2-, 3- or n-step return, and finally create a combined reward as in the backup diagram shown in Figure 1.

V. Empirical Results

In this section we report on the experiments we performed with the three different algorithms (pure Monte Carlo, Forward Monte Carlo and One-Step Estimates). For the pure Monte Carlo and the Forward Monte Carlo we can construct theoretical proofs, guaranteeing convergence to an equilibrium path in a multi-stage game. This can be proved under the assumptions that the learning automata use the L_{R-I} update scheme and the step sizes are chosen small enough. We have however no guarantee that the automata will converge to the optimal equilibrium path.

It is often the case that when you want to apply some techniques to an actual setting, you are more interested in how the techniques perform under certain constraints. Here, we want to know how fast and accurate the techniques perform.

A. Narendra 3 stages

In [17] the authors reported on a 2 stage game in which the learning automata at the first stage have to play a local suboptimal to reach the global optimal in the end. We have constructed a game with similar properties meaning that the automata in the first and second stage have to play a local suboptimal to find the global optimal path in the end. The reward matrices are shown in Figure 6. The Nash equilibria of the local reward matrices are underlined and the joint-actions of the optimal path in the game is set in bold font.

$$M^1 = \begin{pmatrix} \underline{0.7} & \mathbf{0.6} \\ 0.1 & 0.1 \end{pmatrix} \quad M^2 = \left(\begin{array}{cc|cc} \underline{0.6} & 0.2 & 0.3 & \underline{0.85} \\ 0.3 & 0.1 & 0.2 & \mathbf{0.7} \\ \hline \underline{0.4} & 0.1 & \underline{0.3} & 0.2 \\ 0.2 & 0.1 & 0.2 & 0.2 \end{array}\right)$$

$$M^3 = \left(\begin{array}{cccc||cccc} \underline{0.7} & 0.6 & 0.6 & \underline{0.7} & \underline{0.7} & 0.3 & \underline{0.3} & 0.1 \\ 0.1 & 0.1 & 0.4 & 0.2 & 0.3 & 0.4 & 0.2 & 0.1 \\ \hline 0.1 & 0.3 & \underline{0.5} & 0.4 & 0.4 & \underline{0.6} & \mathbf{1.0} & 0.9 \\ 0.2 & \underline{0.4} & 0.2 & 0.35 & 0.2 & 0.1 & 0.75 & 0.65 \\ \hline\hline 0.1 & \underline{0.2} & 0.4 & 0.2 & \underline{0.6} & 0.2 & \underline{0.75} & 0.6 \\ 0.2 & 0.2 & 0.45 & \underline{0.65} & 0.2 & 0.1 & 0.55 & 0.3 \\ \hline 0.7 & 0.7 & 0.6 & \underline{0.7} & \underline{0.4} & 0.1 & \underline{0.8} & 0.2 \\ 0.5 & \underline{0.8} & 0.3 & 0.1 & 0.2 & 0.1 & 0.1 & 0.4 \end{array}\right)$$

Fig. 6. Reward matrices of our constructed 3 stage game. The local Nash equilibria are underlined. The joint-actions that should be played to reach the optimal solution are displayed in bold font.

In Figure 7 we see the convergence speed and convergence accuracy for the Monte Carlo technique, the Forward Monte Carlo and the One-Step Estimate technique applied on this game. Note that the full lines correspond to the left Y-axis and depict the number of time steps that have passed before both the agents are converged to one single path in their hierarchy. The dashed line corresponds to the right Y-axis and depicts the number of times the optimal path in the multi-stage game was found. On the X-axis we see value for the step size (i.e. α in Eq. (1)-(2)) we used in the experiments. The results are averaged over 100 runs.

From these results we can see that the best results for convergence are reached when using the Forward Monte Carlo technique. The traditional Monte Carlo does worse, but still outperforms our One-Step Estimates. For the 3 techniques we see that the number of steps needed to converge decreases as the learning rate increases (note that their is no real difference in the speed of convergence between the 3 techniques). However when we look at the convergence accuracies, only those of Monte Carlo and the Forward Monte Carlo decrease while that of the One-Step Estimates remain almost constant.

We can conclude from this figure that bootstrapping in a game with a low depth does not perform well. This was of course a bit expected because only the top level can really benefit from the bootstrapping.

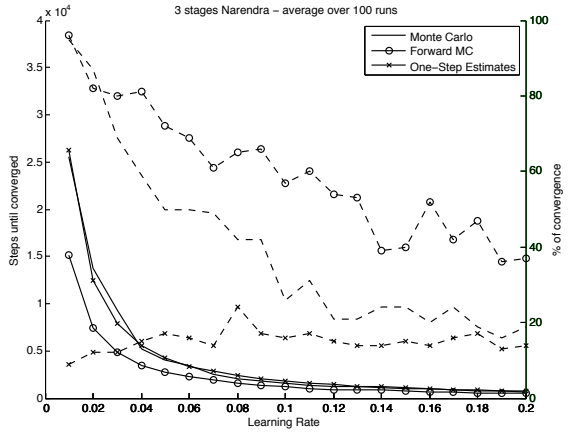

Fig. 7. 3 Stages version of the Game of Narendra. The full line depicts the number of steps needed to converge to a path in both the hierarchies. The dotted line represents the percentage of convergence.

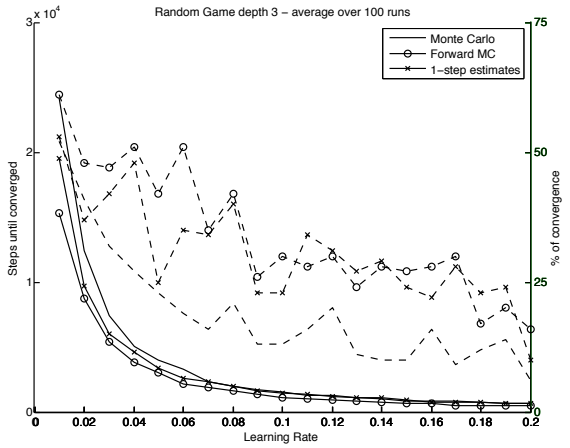

Fig. 8. Random Game of 3 stages. The full line depicts the number of steps needed to converge to a path in both the hierarchies. The dotted line represents the percentage of convergence.

B. Random Game of 3 stages

In our next experiment we applied the 3 techniques to a Random Game of depth 3. The total number of joint-paths in the game is: $(2^3)^2 = 64$. In a Random Game, the matrices are filled with random values. Again these results are averaged over 100 runs (and every run generates a new Random Game). The convergence speed of the 3 techniques remains about the same. While the performance for the traditional Monte Carlo and the Forward Monte Carlo drops significantly (optimal percentage from the previous experiment goes down from 95%/96% to 51%/61%) the performance of the One-Step Estimates rises from 24% to 53%. In the previous experiment the learning automata are misled in the first and second stage of the game. Having information about the full reward of the path improved the performance of (Forward) Monte Carlo. Because in this game, the matrices are generated randomly, the chance that such a situation occurs is smaller and hence the performance of the One-Step Estimates rises.

C. Random Game of 6 stages

For our next experiment, we create Random Games of 6 stages. This gives a total of $(2^6)^2 = 4096$ joint-paths in the tree. Again we compare the 3 techniques. The results can be seen in Figure 9. While the performance drops for all the techniques (which is to be expected because we went from 64 possible solutions to 4096) we begin to see that the One-Step Estimates are starting to perform better. We contribute the bad results of the traditional Monte Carlo to the fact that their learning automata get updated with rewards that are averaged over 6 stages. The more the rewards are averaged, the closer the averages become and the more difficult it becomes for the learning automata to differentiate between optimal and suboptimal paths. Also the Monte Carlo technique needs more time to distinguish between the different paths .

While the techniques are still able to find the optimal solution in a Random Game of depth 6, we have no information

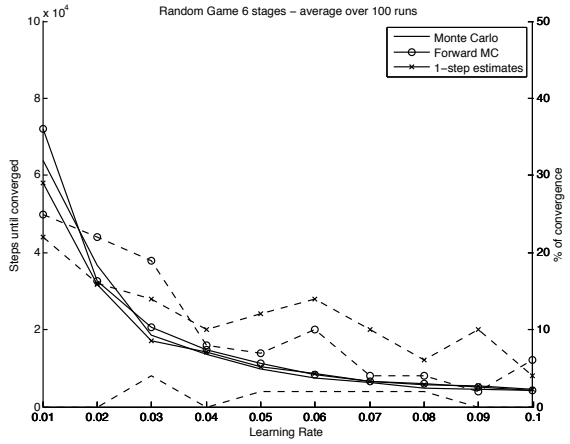

Fig. 9. Random Game of 6 stages. The full line depicts the number of steps needed to converge to a path in both the hierarchies. The dotted line represents the percentage of convergence.

about the quality of the paths reached when they fail to reach the optimal path. Therefore we repeated the experiment, this time however we monitored how close the techniques came to finding *the optimal solution* $-\epsilon$. In this experiment we made $\epsilon = 0.05$. Thus if the optimal path gives a reward of 0.8 in total, the agents behave optimal when they converged to a path with a pay-off in the interval $[0.75, 0.8]$. The results for this experiment can be seen in Figure 10. While the speed of convergence remains about the same, the percentage of converges almost triples. This indicates that when the automata converge to a suboptimal path, this path still gives a high reward.

D. Random Game of 8 stages

Since we expected the One-Step Technique to perform better on games with more stage where bootstrapping could really have an added benefit, we created Random Games of 8 stages.

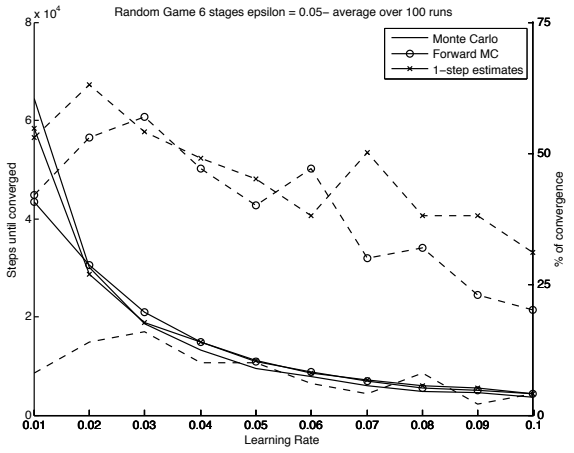

Fig. 10. Random Game of 6 stages and $\epsilon = 0.05$. The full line depicts the number of steps needed to converge to a path in both the hierarchies. The dotted line represents the percentage of convergence.

This gives a total of $(2^8)^2 = 65536$ possible joint-actions in the multi-stage game. The results are plotted in Figure 11. It is clear that these results confirm our suspicion. While the performance of the Monte Carlo technique has dropped to almost 0, the One-Step Estimates still find the optimal result (with a small convergence margin of $\epsilon = 0.01$). The parameter setting used for the One-Step Estimates are: $\rho = 0.5$, $\sigma = 0.5$ and $\tau = 0.5$.

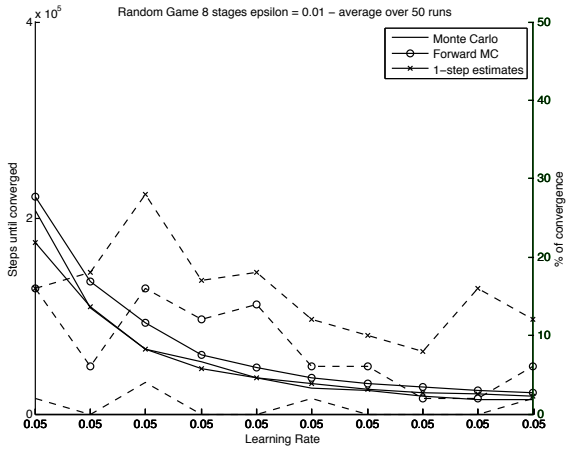

Fig. 11. Random Game of 8 stages and $\epsilon = 0.01$. The full line depicts the number of steps needed to converge to a path in both the hierarchies. The dotted line represents the percentage of convergence.

VI. Conclusion

The work in this paper shows that the idea of bootstrapping can be used in hierarchical learning automata. While for the traditional Monte Carlo and Forward Monte Carlo methods we can construct a theoretical proof of convergence to an equilibrium path, these proofs only remain valid for a small step size. In practice such a small step size results in a slow convergence. With empirical results we argue that One-Step Estimates are a more suitable algorithm for finding good (often optimal) solutions in large sequential decision problems. Furthermore One-Step Estimates converge faster and with a higher accuracy in large games.

Acknowledgment

Research of Maarten Peeters is funded by a Ph.D grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT- Vlaanderen).

References

[1] M. L. Tsetlin, "On the behavior of finite automata in random media," *Avtomatika i Telemekhanika*, vol. 22, no. 10, pp. 1345–1354, 1961.

[2] K. S. Narendra and M. A. L. Thathachar, "Learning automata - a survey," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-4, no. 4, pp. 323–334, 1974.

[3] ——, *Learning Automata: An Introduction*. Englewood Cliffs, NJ: Prentice Hall, 1989.

[4] M. A. L. Thathachar and P. S. Sastry, *Networks of Learning Automata: Techniques for Online Stochastic Optimization*. Kluwer Academic Publishers, 2004.

[5] A.Nowé, K.Verbeeck, and M.Peeters, "Learning automata as a basis for multi-agent reinforcement learning," *Lecture Notes in Computer Science*, vol. 3898, pp. 71–85, 2006.

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

[7] K.Verbeeck, A.Nowé, J. Parent, and K. Tuyls, "Exploring selfish reinforcement learning in repeated games with stochastic rewards," *Journal of Autonomous Agents and Multi-agent Systems*, to appear.

[8] M. A. L. Thathachar and K. R. Ramakrishnan, "A hierarchical system of learning automata," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-11, no. 3, pp. 236–241, 1981.

[9] K. R. Ramakrishnan, "Hierarchical systems and cooperative games of learning automata," Ph.D. dissertation, Indian Institute of Science, Bangalore, India, 1982.

[10] C. Boutilier, "Sequential optimality and coordination in multiagent systems," in *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, 1996, pp. 478–485.

[11] K. Verbeeck, A. Nowé, K. Tuyls, and M.Peeters, "Multi-agent reinforcement learning in stochastic single and multi-stage games." in *Kudenko et al (Eds): Adaptive Agents and Multi-Agent Systems II*. Springer LNAI 3394, 2005, pp. 275–294.

[12] C. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8(3), pp. 279 – 292, 1992.

[13] Y. Shoham, R. Powers, and T. Grenager, "Multi-agent reinforcement learning: a critical survey," Stanford University, Tech. Rep., 2003.

[14] L. P. Kaelbling, M. L. Littman, and A. P. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[15] J. Tsitsiklis, "Asynchronous stochastic approximation and q-learning," *Machine Learning*, vol. 16, pp. 185 – 202, 1994.

[16] M. A. L. Thathachar and K. R. Ramakrishnan, "A hierarchical system of learning automata," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-11, pp. 236–241, 1981.

[17] K. S. Narendra and K. Parthasarathy, "Learning automata approach to hierarchical multiobjective analysis," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, no. 2, pp. 263–273, 1991.

[18] M. Peeters, A. Nowé, and K. Verbeeck, "Bootstrapping versus monte carlo in a learning automata hierarchy," in *Adaptive Learning Agents and Multi-Agent Systems*, 2006, pp. 61–71.

[19] ——, "Toward bootstrapping in a hierarchy of learning automata," in *Proceedings of the Seventh European Workshop on Reinforcement Learning*, 2005, pp. 31–32.