Non-linear Dynamics in Multiagent Reinforcement Learning Algorithms

(Short Paper)

Sherief Abdallah
British University
Dubai, United Arab Emirates
sherief.abdallah@buid.ac.ae

Victor Lesser
University of Massachusetts
Amherst, MA
lesser@cs.umass.edu

ABSTRACT

Several multiagent reinforcement learning (MARL) algorithms have been proposed to optimize agents' decisions. Only a subset of these MARL algorithms both do not require agents to know the underlying environment and can learn a stochastic policy (a policy that chooses actions according to a probability distribution). Weighted Policy Learner (WPL) is a MARL algorithm that belongs to this subset and was shown, experimentally in previous work, to converge and outperform previous MARL algorithms belonging to the same subset.

The main contribution of this paper is analyzing the dynamics of WPL and showing the effect of its non-linear nature, as opposed to previous MARL algorithms that had linear dynamics. First, we represent the WPL algorithm as a set of differential equations. We then solve the equations and show that it is consistent with experimental results reported in previous work. We finally compare the dynamics of WPL with earlier MARL algorithms and discuss the interesting differences and similarities we have discovered.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence

Keywords

Reinforcement Learning, Multiagent Systems, Dynamics, Convergence Analysis

1. INTRODUCTION

Our focus in this paper is on a class of MARL algorithms that use a gradient-ascent approach to guide policy search. We will refer to that class as GA-MARL throughout the paper. The general idea of GA-MARL algorithms (more details later in Section 2) is to approximate the policy-gradient using a payoff-gradient and follow the gradient until reaching a local maxima.

A GA-MARL algorithm learns a stochastic policy (a policy that chooses actions according to a probability distribution) without knowing the underlying model of the environment. This ability is particularly important when the world is not fully observable. Another advantage of GA-MARL algorithms is their (relative) simplicity, which makes analyzing their dynamics possible.

The first GA-MARL algorithm whose dynamics were analyzed is the Infinitesimal Gradient Ascent (IGA) algorithm. The dynamics of IGA were linear and IGA's convergence was fairly limited [1]. The IGA-WoLF algorithm had later been developed to address IGA's limitations. The dynamics of IGA-WoLF were piece-wise-linear[1] and IGA-WoLF made strong assumptions in order converge.

We previously developed the Weighted Policy Learner (WPL) [2], which we showed experimentally to converge without knowing the equilibrium strategy, a major improvement over IGA-WoLF. The main contribution of this paper is providing an analysis of WPL's dynamics, showing that it is non-linear, and comparing it to other gradient-based MARL algorithms.

The document is organized as follows. Section 2 introduces previous gradient ascent multiagent learning algorithms. Section 3 breifly described the WPL algorithm. Section 4 formulates the WPL algorithm as a set of differential equations. Section 5 discusses the symbolic solution of WPL's differential equations and how it differs from previous gradient ascent MARL algorithms. In Section 6 we present the results of solving WPL's differential equations numerically, and compare our results to the experimental results reported in previous work. Section 7 discusses WPL's dynamics in comparison to previous gradient-based MARL algorithms. Finally in Section 8 we conclude and discuss future work.

2. GRADIENT-BASED MARL ALGORITHMS

The first gradient-based MARL algorithm whose dynamics were analyzed is the Infinitesimal Gradient Ascent (IGA) [1]. IGA is a simple gradient ascent algorithm where each agent i updates its policy π_i to follow the gradient of expected payoffs (or the value function) V_i. The following equations describe how an agent using IGA updates its policy.

$$\Delta\pi_i \leftarrow \eta \frac{\partial V_i(\pi)}{\partial \pi_i}$$
$$\pi_i \leftarrow limit(\pi_i + \Delta\pi_i)$$

[1]We review the analysis of both IGA and IGA-WoLF dynamics later in the paper.

Cite as: Non-linear Dynamics in Multiagent Reinforcement Learning Algorithms (Short Paper), S. Abdallah and V. Lesser, Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008), Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp. 1321-1324.

Variable η is called the learning rate and approaches zero in the limit ($\eta \to 0$) (hence the word Infinitesimal in IGA). Function *limit* projects the updated policy to the space of valid policies, i.e. where $limit(x) = argmin_{x':valid(x')}|x - x'|$.[2] A policy is valid if it sums to 1 and every action is played with non-negative probability.

IGA does not converge in all two-player-two-action games. Algorithm IGA-WoLF (WoLF stands for Win or Learn Fast) was proposed [4] in order to improve convergence properties of IGA by using two different learning rates. More formally,

$$\Delta\pi_i(a) \leftarrow \frac{\partial V_i(\pi)}{\partial \pi_i}(a) * \begin{cases} \eta_{lose} & \text{if } V_i(\pi_i, \pi_{-i}) < V_i(\pi_i^*, \pi_{-i}) \\ \eta_{win} & \text{otherwise} \end{cases}$$
$$\pi_i \leftarrow limit(\pi_i + \Delta\pi_i)$$

Notice that if an agent moves away from its equilibrium policy, this means the value (expected reward) of the current policy is higher than the value of the equilibrium policy and vice versa (which explains the conditions in the above equation). The dynamics of IGA-WoLF have been analyzed and proven to converge in all 2-player-2-action games [4], as we briefly review in the following section. IGA-WoLF has limited practical use, however, because it requires each agent to know its equilibrium policy.

3. WEIGHTED POLICY LEARNER (WPL)

The WPL algorithm is shown in Algorithm 1 for agent i. Variable Δ is the policy gradient that is used to update policy π_i. The idea of the algorithm is to start learning fastest when Δ changes its direction and then to gradually slow down learning if the policy gradient does not change its direction.

Algorithm 1: WPL: Weighted Policy Learner

begin
  $\hat{V} \leftarrow$ total average reward $= \frac{\sum_{a \in A_i} V_i(a)}{|A|}$.
  foreach *action* $a \in A_i$ **do**
    $\Delta(a) \leftarrow V_i(a) - \hat{V}$
    if $\Delta(a) > 0$ **then** $\Delta(a) \leftarrow \Delta(a)(1 - \pi_i(a))$
    else $\Delta(a) \leftarrow \Delta(a)(\pi_i(a))$
  end
  $\pi_i \leftarrow limit(\pi_i + \eta\Delta)$
end

WPL detects changes in the gradient direction using the difference between action rewards. If the reward of action a is decreasing, then the change in $\pi_i(a)$, $\Delta(a)$, is weighted by $\pi_i(a)$, otherwise it is weighted by $(1 - \pi_i(a))$. Therefore, the largest positive change in $\pi_i(a)$, $\Delta(a)$, is when $\pi_i(a)$ is low and $V_i(a)$ is higher than the average reward $\hat{V}$, and the largest negative change is when $\pi_i(a)$ is near 1 and $V_i(a)$ is lower than $\hat{V}$.

Notice that there are few differences and similarities between IGA-WoLF and WPL's update rules. Both algorithms have two modes of learning rates. IGA-WoLF needs to know the equilibrium strategy in order to distinguish between the two modes, unlike WPL. Also while IGA-WoLF has fixed learning rates for the two modes, WPL uses a continuous spectrum of learning rates, depending on the current policy. It is this particular feature that causes WPL's dynamics to be non-linear, as we discuss in the following section.

[2]This general definition of the *limit* function was later developed [3].

4. FORMULATING WPL AS DIFFERENTIAL EQUATIONS

The policies of two agents, p and q, following WPL can be expressed as follows

$$q(t) \leftarrow limit(q(t-1) + \Delta q(t-1))$$
$$p(t) \leftarrow limit(p(t-1) + \Delta p(t-1))$$

where

$$\Delta q(t) = \begin{cases} \eta(1 - q(t))(u_3 p(t) + u_4) & \text{if } u_3 p(t) + u_4 > 0 \\ \eta q(t)(u_3 p(t) + u_4) & \text{if } u_3 p(t) + u_4 < 0 \end{cases}$$
$$\Delta p(t) = \begin{cases} \eta(1 - p(t))(u_1 q(t) + u_2) & \text{if } u_1 q(t) + u_2 > 0 \\ \eta p(t)(u_1 q(t) + u_2) & \text{if } u_1 q(t) + u_2 < 0 \end{cases}$$

We continue derviation of $q(t)$, and similar analysis holds for $p(t)$.

$$\frac{q(t) - q(t-1)}{\eta} =$$
$$\begin{cases} (1 - q(t))(u_3 p(t) + u_4) & \text{if } p(t) > p^* = -u_4/u_3 \\ q(t)(u_3 p(t) + u_4) & \text{if } p(t) < p^* = -u_4/u_3 \end{cases}$$

As $\eta \leftarrow 0$, the equations above become differential:

$$q'(t) = \begin{cases} (1 - q(t))(u_3 p(t) + u_4) & \text{if } p(t) > p^* = -u_4/u_3 \\ q(t)(u_3 p(t) + u_4) & \text{if } p(t) < p^* = -u_4/u_3 \end{cases}$$

Since WPL is a gradient ascent approach, WPL will converge to a deterministic NE if one exists, similar to IGA and IGA-WoLF. This is clear from the gradient definition: a deterministic NE means one action is *always* better than the other, and therefore the gradient direction always points to it leading to eventual convergence [1]. The challenging case is when there is no deterministic NE (the NE is inside the joint policy space). We will therefore focus on this case.

It should be noted that while IGA and IGA-WoLF needed to take the limit function into account, we can safely ignore the limit function while analyzing the dynamics of WPL for 2-player-2-action games. This is due to the way WPL scales the learning rate using the current policy. By the definition of $\Delta p(t)$, a positive $\Delta p(t)$ approaches zero as $p(t)$ approaches one and a negative $\Delta p(t)$ approaches zero as $p(t)$ approaches zero. In other words, as p (or q) approaches 0 or 1, the learning rate approaches zero, and therefore p (or q) will never go beyond the valid period $[0, 1]$. The following section discusses the symbolic solution of these equations.

5. SYMBOLIC SOLUTION

Our goal is to prove that $p(t)$ and $q(t)$ will eventually converge (i.e. in the limit, when $t \to \infty$) to p^* and q^* respectively. To do so, it is enough to show that if one player starts at a policy q^*, then the next time the player returns to q^* the other player will be closer to its NE, and consequently the joint policy will also be a bit closer.

Figure 1 illustrates this point and depicts $p(t)$ and $q(t)$ over a period of time $0 \to T4$, (the figure to the left shows

policies evolution over time, while the figure to the right shows the joint policy space). If we can prove that over the period $0 \rightarrow T4$ an agent's policy $p(t)$ gets closer to the NE p^*, i.e. $p_{min2} - p_{min1} > 0$ in Figure 1, then by induction the next time period p will get closer to the equilibrium and so on.

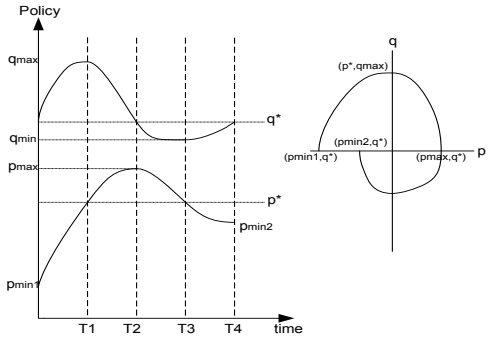


Figure 1: An illustration of WPL convergence.

For readability, p and q will be used instead of $p(t)$ and $q(t)$ for the remainder of this section. The overall period $0 \rightarrow T4$ is divided into four intervals defined by times $0, T1, T2, T3$, and $T4$. Each period corresponds to one combination of $p(t)$ and $q(t)$ as follows. For the period $0 \rightarrow T1$, where $p(t) < p^*$, $q(t) > q^*$: by dividing p' and q'

$$\frac{dp}{dq} = \frac{(1-p)(u_1 q + u_2)}{(1-q)(u_3 p + u_4)}$$

Then by separation we have

$$\int_{p_{min1}}^{p^*} \frac{u_3 p + u_4}{1-p} dp = \int_{q^*}^{q_{max}} \frac{u_1 q + u_2}{1-q} dq$$

$$-u_3(p^* - p_{min1}) + (u_3 + u_4) ln \frac{1-p_{min1}}{1-p^*} =$$

$$-u_1(q_{max} - q^*) + (u_1 + u_2) ln \frac{1-q^*}{1-q_{max}}$$

Unlike IGA and IGA-WoLF, however, the equations are non-linear and do not have a closed-form solution (note the existence of both x and $ln(x)$). This is the case for the remaining three time periods as well. We solve the equations numerically as described in the following section.

6. NUMERICAL SOLUTION

We used Mathematica and Matlab to solve the equations numerically. Figure 2 shows the theoretical behavior predicted by our model for the matching-pennies game. There is a clear resemblance to the actual (experimental) behavior that was reported in the original WPL paper [2] for the same game (Figure 3). Note that the time-scale on the horizontal axes of both figures are effectively the same, because what is displayed on the horizontal axis in Figure 3 is decision steps. When multiplied by the actual learning rate η used in the experiments, 0.001, both axes become identical.

Figure 4 plots $p(t)$ versus $q(t)$, for a game with NE= $(0.9, 0.9)$ $(u1 = 0.5, u2 = -0.45, u3 = -0.5, , u4 = 0.45)$ and starting from 160 initial joint policies. Figure 6 plots $p(t)$ and $q(t)$ against time, verifying convergence from each of the 160 initial joint policies.

We repeated the above numerical solution for 100 different NE(s) that make a 10x10 grid in the p-q space (starting

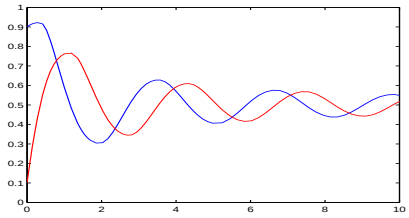


Figure 2: Convergence of WPL as predicted by the theoretical model for the matching pennies game.

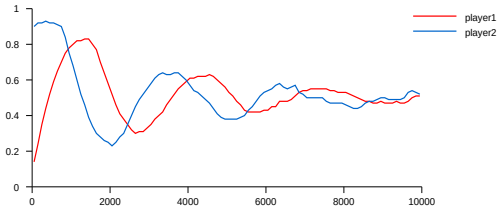


Figure 3: Convergence of WPL through experiments [2].

from the 160 boundary joint policies). The WPL algorithm converges to the NE in a spiral fashion similar to Figure 4 in all the 100 cases. Instead of drawing 100 figures (one for each NE), Figure 5 plots the merge of the 100 figures in a compact way: plotting the joint policy from time 700 to 800 (which is enough for convergence as Figure 6 shows). The two agents converge in all the 100 NE cases, as indicated by the centric points (a diverging algorithm would not have a clean grid with concentrated centric plots).

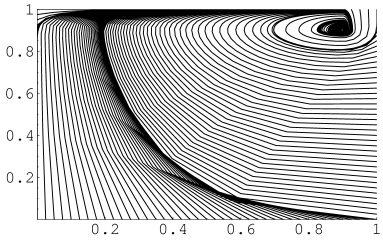


Figure 4: An illustration of WPL convergence to the (0.9,0.9) NE in the p-q space: p on the horizontal axis and q on the vertical axis.

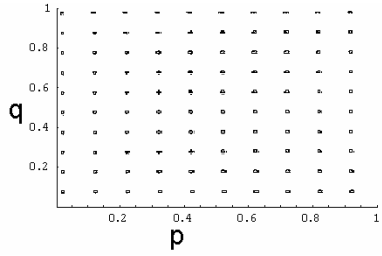


Figure 5: An illustration of WPL convergence for 10x10 NE(s).

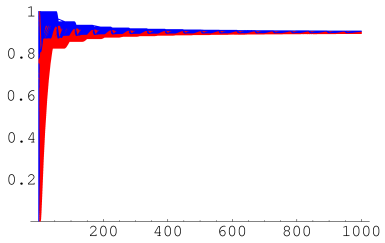


Figure 6: An illustration of WPL convergence to the (0.9,0.9) NE: p(t) (gray) and q(t) (black) are plotted on the vertical axis against time (horizontal axis).

7. COMPARING DYNAMICS OF IGA, IGA-WOLF, AND WPL

With differential equations modeling each of the three algorithms, we now compare their dynamics and point out the main distinguishing characteristics of WPL. Matlab was again used to solve the differential equations (of the three algorithms) numerically. Figure 7 shows the dynamics of the three algorithms in a game with $u1u3 < 0$ and the NE=(0.5,0.5). The joint strategy moves in clockwise direction. The dynamics of WPL are very close to IGA-WoLF, with slight advantage in favor of IGA-WoLF (after one complete round around the NE, IGA-WoLF is closer to the NE than WPL). It is still impressive that WPL has comparable performance to IGA-WoLF, while WPL does not require agents to know their NE strategy a priori.

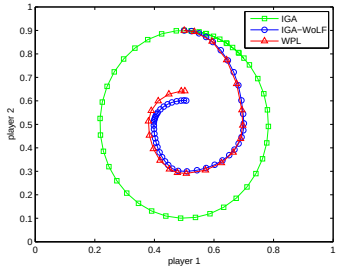


Figure 7: Dynamics of IGA, IGA-WoLF, and WPL in a game with NE=(0.5,0.5).

Figure 8 shows the dynamics in a game with again $u1u3 < 0$ but the NE=(0.5,0.1). Three interesting regions in the figure are designated with A,B, and C. Region A shows that both IGA and IGA-WoLF dynamics are *discontinuous* due to the hard constraints on the policy. Because WPL uses a smooth policy weighting scheme, the dynamics remain continuous. This is also true in region B. In region C, WPL initially deviates from the NE more than IGA, but eventually converges as well. The reason is that because the NE, in this case, is closer to the boundary, policy weighting makes the vertical player move at a much slower pace when moving downward (the right half) than the horizontal player.

Figure 9 shows the dynamics for the coordination game, starting from initial joint policy (0.1,0.6). The coordination game has two NEs: (0,0) and (1,1). All algorithms converge to the closer NE, (0,0), but again we see that both IGA and IGA-WoLF have discontinuity in their dynamics, unlike WPL which smoothly converge to the NE.

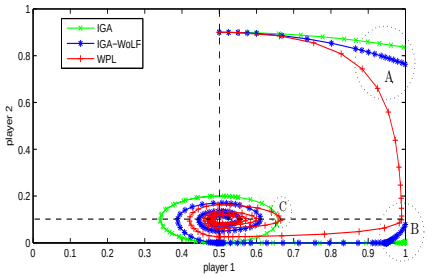


Figure 8: Dynamics of IGA, IGA-WoLF, and WPL in a game with NE=(0.5,0.1).

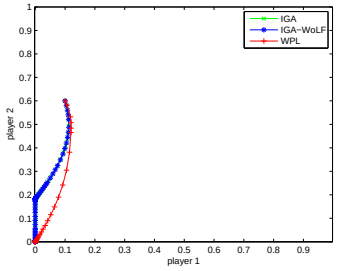


Figure 9: Dynamics of IGA, IGA-WoLF, and WPL in the coordination game with two NEs=(0,0) and (1,1).

8. CONCLUSION AND FUTURE WORK

The main contribution of this paper is formally analyzing the Weighted Policy Learner algorithm and showing that it is the first gradient-ascent (GA) MARL algorithm with nonlinear dynamics. The paper models the WPL algorithm for two-player-two-action games as a set of differential equations and then discusses both symbolic and numerical solutions to the equations. The predicted theoretical behavior closely resembles and confirms previously obtained experimental results. Furthermore, the paper solves the equations for 100 games, each starting from 160 initial joint policies and verified WPL's convergence in all of them. Finally, a comparison of WPL's dynamics with previous GA-MARL algorithms' dynamics is given, along with a discussion of similarities and differences.

9. REFERENCES

[1] Singh, S., Kearns, M., Mansour, Y.: Nash convergence of gradient dynamics in general-sum games. In: the 16th Conference on Uncertainty in Artificial Intelligence. (2000) 541–548

[2] Abdallah, S., Lesser, V.: Learning the task allocation game. In: Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS). (2006)

[3] Zinkevich, M.: Online convex programming and generalized infinitesimal gradient ascent. In: Proceedings of the International Conference on Machine Learning. (2003) 928–936

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

Expediting RL by Using Graphical Structures (Short Paper)

Peng Dai
Dept of Computer Science and Engineering
University of Washington
Seattle, WA 98195
daipeng@cs.washington.edu

Alexander L. Strehl
Yahoo! Research
New York, 10018
strehl@yahoo-inc.com

Judy Goldsmith
Dept of Computer Science
University of Kentucky
Lexington, KY 40506-0046
goldsmit@cs.uky.edu

ABSTRACT

The goal of Reinforcement learning (RL) is to maximize reward (minimize cost) in a Markov decision process (MDP) without knowing the underlying model *a priori*. RL algorithms tend to be much slower than planning algorithms, which require the model as input. Recent results demonstrate that MDP planning can be expedited, by exploiting the graphical structure of the MDP. We present extensions to two popular RL algorithms, Q-learning and RMax, that learn and exploit the graphical structure of problems to improve overall learning speed. Use of the graphical structure of the underlying MDP can greatly improve the speed of planning algorithms, if the underlying MDP has a nontrivial topological structure. Our experiments show that use of the *apparent* topological structure of an MDP speeds up reinforcement learning, even if the MDP is simply connected.

1. INTRODUCTION

Given a set of states, a set of actions, an initial state and a set of goal states, classical planning finds a sequence of actions that proceeds from the initial state to a goal state while minimizing cost. Decision theoretic planning [2] is a powerful extension that introduces outcome uncertainty.

Markov decision processes are a widely used model for AI researchers to represent decision theoretic planning problems. Given an MDP model, a planner finds a solution that has the optimal or at least acceptable cost. Classical MDPs solvers such as value iteration [1] use dynamic programming. This assumes the model is known. In reinforcement-learning (RL), the MDP environment is initially unknown, so dynamic programming is not immediately applicable.

There are two main approaches to RL, *model-free learning* and *model-based learning* (or simply *model-learning*) [12]. Model-free algorithms learn a value function or policy directly from the data, while model-based algorithms first construct an MDP model that they then use to reason about future actions and costs.

We show two basic RL algorithms can be made faster and more practical by learning and exploiting knowledge of the underlying graphical structure of environments. By examining the topological structure of the MDP's *reachability graph* rooted at the initial state, algorithms that use dynamic programming techniques can be modified to find near-optimal policies more quickly in many MDPs.

The contribution of our paper is three-fold. We show, in detail, how two representative reinforcement learning algorithms, Q-learning (model-free) and RMax (model-based), can be modified to use the underlying graphical MDP structure. We discuss how this method can be extended to current and future RL algorithms. Finally, we provide extensive empirical evaluation of our algorithms in comparison with the old algorithms. The experiments show that graphical-structure analysis significantly benefits RL algorithms.

Cite as: Expediting RL by Using Graphical Structures (Short Paper), Peng Dai, Alexander L. Strehl and Judy Goldsmith, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal,pp. 1325-1328.

2. BACKGROUND

A *scenario based MDP* is a six-tuple $\langle S, A, T, C, s_0, G\rangle$. S is a finite set of system states. A is a finite set of actions. $T_a(s'|s)$ is the probability of the system changing from state s to s' by performing action a. $C(s,a)$ is the instantaneous cost of performing action a at state s. $s_0 \in S$ is the initial state and $G \subseteq S$ is a set of goal states. We will use "MDP" for "scenario based MDP" for the rest of the paper.

Given an MDP, we define a *policy* $\pi : S \rightarrow A$ to be a mapping from the state space to the action space. A *value function* V^π for policy π, $V^\pi(s) : S \rightarrow \mathbf{R}$ denotes the value of the total expected cost starting from state s and following the policy π: $V^\pi(s) = C(s,\pi(s)) + \sum_{s' \in S} T_{\pi(s)}(s'|s)V^\pi(s')$. A policy π_1 dominates another policy π_2 if $V_{\pi_1}(s) \leq V_{\pi_2}(s)$ for all $s \in S$. An *optimal policy* π^* is a policy that is not dominated by any other policy, and is the optimal solution of the MDP. The value function of the optimal policy is called the *optimal value function* $V^*(\cdot)$. Bellman [1] showed that $V^*(\cdot)$ can be calculated by solving a system of linear equations in the form

$$V^*(s) = min_{a \in A(s)}[C(s,a) + \sum_{s' \in S} T_a(s'|s)V^*(s')]. \quad (1)$$

Equation 1 is also know as the *Bellman equation*. Using the Bellman equation as an assignment operator over a particular state is denoted as a *Bellman backup*. Bellman backups are the basic operations of dynamic programming, a technique of solving MDPs by calculating their optimal value functions.

Dynamic programming works directly in value function space. It backs up the value of states according to some order, until a time when further backups would result in only a very small change to the value function. We use the term *converge* loosely and informally to mean that the learned value function is sufficiently close to the optimal value functions. The simplest variant of value iteration [1], for example, initializes the value functions arbitrarily, and updates its value function by applying Bellman backups on every state in a fixed order. The algorithm halts when the largest change

in value during the most recent iteration is smaller than a threshold. Once the optimal value function is sufficiently approximated, a near-optimal policy, π, is easily extracted by choosing an action for each state:

$$\pi(s) = argmin_{a \in A(s)}[C(s,a) + \sum_{s' \in S} T_a(s'|s)V(s')].$$

Topological value iteration (TVI) [4] is a recent dynamic programming MDP algorithm . It makes use of the graphical structure of MDPs to perform Bellman backups in a better order. TVI first constructs a directed graph G from an MDP: the vertices of G are the states of the MDP, and the directed edges are state transitions. If the probability $T_a(s'|s) > 0$, then the edge $s \rightarrow s'$ is in G. TVI then computes the strongly connected components (SCCs) of G and their topological order. It solves every connected component sequentially by value iteration, according to this order. TVI outperforms VI significantly in MDP domains that have a reasonable number of SCCs.

3. MODEL-FREE LEARNING

Previously, we discussed how to obtain a near-optimal value function and policy for an MDP assuming we already have a *model*. The model consists of the cost function C and transition function T. In the reinforcement learning setting, we want to find a near-optimal value function and policy when the model is not initially provided. RL algorithms interact with the environment to get approximations of the model, and therefore solve the MDP.

Q-learning [13, 14] is a standard RL algorithm for MDPs. The algorithm maintains *Q-values* for each state action pair. $Q^*(s,a) = C(s,a) + \sum_{s' \in S} T_a(s'|s) min_{a'} Q^*(s',a')$. $Q^*(s,a)$ stands for the minimum expected cost of being in state s, applying action a, and then following the optimal policy. Thus, the optimal value function of s is the minimum Q-value with respect to s, $V^*(s) = min_a Q^*(s,a)$.

In MDPs, Q-learning initiates exploitation trials from the initial state. In each step of the trial, an action a is chosen for the current state s, which transitions the learning agent stochastically to s' according to the (unknown) transition function. A cost $c(s,a)$ is sensed, and $Q(a,s)$ is updated by $Q(s,a) = Q(s,a) + \alpha(c(s,a) + min_{a'} Q(s',a') - Q(s,a))$, where α is the *learning factor*, which it is often decreased as the time passes. Updating a Q-value is called a *Q-backup*.

Q-learning is very powerful, and is guaranteed to converge to an optimal policy, albeit sometimes slowly. One weakness is that it uses the same learning strategies for every MDP. The intuition behind our **topological Q-learning (TQL)** algorithm comes from TVI. TQL has two phases. The first phase is the initial learning phase. Here, we learn graphical information as well as Q-values. We initiate trials from the initial state the same way as Q-learning. As well as updating the value function for state-action pairs encountered along the trials, we record all predecessor-successor pairs visited during those trials. In other words, we mark all the visited edges of G. After a certain number (x) of trials, we use the recorded edge information to construct a directed graph, the *reachability graph,* G_R. Notice that the reachability graph is by no means guaranteed to be identical to the real G, since the trials might not visit all edges or even all states. However, if the learning process is sufficiently long, the information of learned state-action pairs is sufficient to solve the original MDP.

Given the reachability graph, we apply Kosaraju's algorithm to find the SCCs of G_R and their topological order. In the second phase, we choose one component at a time according to this order, pick one state from this component, and initiate trials from that state until the current component is converged. These trials are slightly different from trials of Q-learning. In Q-learning, a trial terminates only when a goal state is encountered. But trials of the second phase TQL finish when they run into a goal state or get into a state belonging to a component whose topological order is larger than the current one. This is because when a component is converged, all its states are converged, and we do not back up converged states. So if a later trial reaches converged states, we stop it. In each component, we initiate trials from the same state, since every other state in the component is reachable from this state. If we do enough trials, every state in that component gets backed up sufficiently.

Suppose we are asked to provide an online RL agent that takes advantage of the topological structure. We outline a simple extension to TQL to achieve this goal. When TQL learns that a transition from state s to s' is possible, it stores this fact in its reachability graph. With little additional overhead, we could store each experience-tuple (s,a,c,s') that is observed by agent, and link each of these tuples to the state s in our reachability graph.[1] Then, in the second phase, instead of initiating more trials from various states according to the topological ordering, we could simply run Q-learning over our saved experience-tuples from those states (and their outgoing neighbors in the reachability graph). This method can be viewed as a version of the "experience replay" algorithm [12] that takes advantage of learned topology.

One problem with the experience-replay approach described above is that storing every experience-tuple is memory intensive. An alternative approach is to maintain and update an approximate model of the underlying MDP.[2] After this, we can initiate Q-learning trials from any state by simulating them in our model. Alternatively, we could solve the model directly. This approach is developed in full detail in the next section.

4. MODEL-BASED LEARNING

Model-based RL algorithms use the agent's experience to estimate the system dynamics (transitions and costs) of the underlying MDP. It is straightforward to compute the maximum-likelihood model of the cost and transition distributions for each state-action pair. For instance, if we've seen n_1 transitions from state s to state s' after action a, out of n_2 total transitions from state s after action a, we would estimate the unknown transition probability $T_a(s'|s)$ by $\hat{T}_a(s'|s) = n_1/n_2$. As the agent gains experience over the state-action space, its model converges to the true MDP. Once the agent estimates the model, it can then solve the model using any MDP planner, and act according to an optimal policy. Unfortunately, when little experience has been gathered, the empirical model may be inaccurate, and resulting policies are suboptimal.

Several effective model-based algorithms have been developed, such as E^3 [7], RMax [3], and MBIE [11]. These algorithms estimate a model and its uncertainty. They use their models to obtain either the best known cost (exploitation) or knowledge that will reduce model uncertainty (exploration). The RMax algorithm is a model-based algorithm that has formal guarantees on its learning time [3, 6]. Therefore, we use it as a representative model-based RL algorithm. We describe RMax and discuss how to augment it to take advantage of the MDP's graphical structure. This very simple modification brings vast improvement.

The MDP model used by **RMax** contains the empirical transi-

[1]Here s' is the state reached and c is the immediate cost of taking action a at s.

[2]The Q-values computed by the experience replay algorithm converge to the optimal Q-values of the approximate model.

tion and reward distributions *only for those state-action pairs that have been experienced by the agent at least m times*, for some exploration parameter m. The transition distribution for other state-action pairs is a self loop, and the cost for those state-action pairs is 0, the minimum possible. The intuition is that the transition and cost estimates for those state-action pairs that have not been tried m times are likely to be inaccurate. Instead of using past experience to compute a model for these state-action pairs, we make them minimally costly in RMax's model. By choosing m carefully, RMax learns a near-optimal policy in polynomial time [3, 6].

Here we present an extension of RMax, **Topological RMax (TRMax)**. In RMax, whenever a new state-action pair (s, a) has been visited at least m times, we gather all the other *relevant* state-action pairs, the state-action pairs (s', a') that have the same property, and perform value iteration over them. Like TQL, TRMax has two phases. The first is the same as RMax, except we also remember the visited successor-predecessor pairs. After x trials, we compute the SCCs of the current reachability graph G_R as well as their topological order, then enter the second phase. From then on, when a state-action pair (s, a) has been visited m times, for a state-action pair (s', a') to be relevant, we require (s', a') to have been visited at least m times, and s' must belong to a component that has a higher topological order than the component of s in G_R. We use topological value iteration in solving the new model.

We originally extended RMax by recomputing the SCCs of the reachability graph and the topological ordering each time a new state-action pair was visited m times. We discarded that approach since constructing the SCCs of a directed graph is costly in practice. One possible improvement is to update the SCCs and topological ordering *incrementally* [10]. The overhead required may limit its practicality, but we plan to test this.

5. EXPERIMENTS

RL algorithms often do not have a well defined stopping criterion. During our experiments we kept a running average of the (estimated) value of the initial state. When the most recent value was sufficiently close to the long-term average, we terminated the experiment.

Any implementation of RMax must choose a technique for solving its model and this choice will affect the computational complexity of the algorithm. For our experiments, we used value iteration.

Our topological RL algorithms are based on the reachability graph that is known when we call Kosaraju's algorithm after x trials. What is a reasonable choice for x?

The *influence* of a state s with respect to a policy π, $I^\pi(s)$, is the expected number of times that state is visited in a trial following policy π [9]. Since any trial originates from the initial state s_0, the influences of s_0 is 1. Similarly, $\sum_{g \in G} I^\pi(g) = 1$. When $I^\pi(s) < 1$, it is the probability of s being visited in the exploitation trial. The influence of a state s with respect to the optimal policy is called the *optimal influence* $I^*(s)$.

$$I^\pi(s) = \sum_{s' \in S, a = \pi(s')} T_a(s|s') I^\pi(s'),$$

$$I^*(s) = \sum_{s' \in S, a = \pi^*(s')} T_a(s|s') I^*(s').$$

The influence measures the effect that changing the value of s will have on the value of s_0.

THEOREM 1. *If a state has an optimal influence of at least ϵ, then with probability $p = 1 - (1 - \epsilon)^t$, the optimal policy will visit it at least once in t trials. (In particular, when $\epsilon = 10^{-6}$, $t = 10,000$, $p = 0.99$.)*

PROOF. From the definition of $I^*(s)$, the probability that state s is *not* visited by a trial is $1 - I^*(s)$. Given t independent trials, the probability that s is not visited in any of them is $(1 - I^*(s))^t$, so the probability of s being visited at least once is $1 - (1 - I^*(s))^t$. By hypothesis, $I^*(s) \geq \epsilon$, so with probability $p = 1 - (1 - \epsilon)^t$, s should be visited at least once in t trials. □

We used $x = 10,000$ in our experiments. When we called Kosaraju's algorithm, states that were not visited in those x trials were ignored. Theoretically, we know from the above theorems that they have very small probabilities of making any real difference to the ultimate $V^*(s)$.

We tested Q-learning (QL), Topological Q-learning (TQL), RMax, and Topological RMax (TRMax). Each algorithm was implemented in C, and executed on the same Intel Pentium 4 1.50GHz processor with 512M main memory and a cache of 256kB.

$\|S\|$	5000		10000		1000		2000	
n_l	QL	TQL	QL	TQL	RMax	TRMax	RMax	TRMax
10	21.72	15.25	50.31	27.93	28.14	9.47	117.87	35.78
20	17.68	11.41	38.55	19.94	30.34	10.33	122.72	36.76
30	13.80	9.03	36.32	18.32	28.27	9.40	81.96	22.41
40	16.66	10.32	32.16	17.83	26.45	8.80	95.60	26.73
50	11.68	7.96	38.99	20.70	21.41	6.80	100.82	28.31
60	11.52	7.19	36.30	18.20	23.23	7.46	34.67	15.96
70	11.21	6.77	34.67	15.96	21.77	7.11	87.97	23.64
80	11.91	7.46	36.26	17.50	22.32	7.08	83.98	22.46
90	14.72	9.13	31.76	15.29	24.63	7.01	79.13	21.48
100	12.51	7.78	34.42	19.06	21.45	6.76	82.41	22.19

Table 1: Convergence time (seconds) of learning algorithms on MDPs m_a=5 and m_s = 10 with various layer numbers

domain	$\|S\|$	QL	TQL	RMax	TRMax
RMDP	1000	3.67	4.09	50.05	50.07
racetrack	1849	3.68	3.14	162.39	109.72
RMDP	2000	17.60	17.07	236.89	140.61
RMDP	4000	13.01	12.85	1043.83	576.53
racetrack	5566	24.90	23.82	976.84	279.65
RMDP	10000	43.55	37.83	-	3566.59
racetrack	21371	139.42	160.84	-	-
racetrack	50077	1443.17	1311.60	-	-

Table 2: Convergence time (seconds) of four algorithms on single connected component domains

We first used "layered" MDP domains,[3] [4], and larger problems for QL and TQL, which are usually faster than RMax and TRMax. Each layered MDP configuration is a four-tuple $\langle |S|, m_a, m_s, n_l \rangle$, where $|S|$ is the size of the MDP, m_a the maximum number of actions of each state, m_s the maximum number of successors of a state-action pair, n_l the number of layers. We fixed $|S|$, m_a=10, m_s=5, and varied n_l. For each configuration, we ran 20 MDPs, and averaged their statistics. For each problem, we measured the convergence time, the time taken to get an optimal policy, and the *deviation* of the calculated policy, the difference between the values $V^*(s_0)$ computed by RL algorithms and by value iteration. Convergence times are listed in Table 1. All the deviations in our experiments were $O(10^{-2})$, so were not listed. Looking at the table, we first notice that our topological learning algorithms converged faster than their basic algorithms. Comparing the left and right table, we also find that TRMax achieved a bigger speedup ratio over RMax compared to TQL over QL. This shows that model-based

[3]The "layered" MDPs are nonrepresentational MDPs with multiple SCCs.

learning benefited more from the graphical structure learning. Another interesting phenomenon is that as the number of layers increased, the running time of all the learning algorithms decreased. This is the opposite to the performance curve of dynamic programming approaches reported in [4].

Using TQL, we solved an MDP with 20,000 states and 100 layers within 1 minute, instead of more than 3 minutes by QL. We also solved an MDP with 4,000 states and 50 layers by TRMax within 2 minutes rather than over 6 minutes using RMax. The constant factor speedup shows that topological RL indeed widens the applicability of RL.

Topological value iteration reduces to value iteration when an MDP is strongly connected. We want to investigate if this is also the case for our topological learning algorithms. In this set of experiments, we used strongly-connected MDP problems. In Table 2, we listed the convergence times of algorithms on eight such problems. Random MDP was abbreviated as RMDP. The cut-off time was set at 90 minutes.

The convergence times of TQL were sometimes slower than QL. In those few cases, however, the termination time increased by at most 15%. Interestingly, TQL ran slightly faster than QL on three random MDP problems and two racetrack problems. This phenomenon is more distinct in the comparison between TRMax and RMax.

For the biggest racetrack problem we tested that RMax and TRMax can solve, TRMax was more than twice as fast as RMax, and consistently faster than RMax except for the smallest problem. This is counter-intuitive, since TRMax behaves like RMax except that it uses additional computation by calling Kosaraju's algorithm. The reason for these results follow. First, the *solution graph* of an MDP, containing the set of states and transitions that can be reached from s_0 using the optimal policy, has many fewer edges than G, so may contain multiple connected components. For our problems, the number of SCCs in the solution graph of two smaller racetrack problems are 546 and 1751 respectively. Similar observations were reported in the evaluation of policies for *partially observable MDPs* [5] (on page 117). In problems where a few actions are obviously better than others, the learning algorithm verifies their optimality quickly. The following trials continue to take these actions. Thus, some suboptimal action transitions might never be traversed. Our reachability graph, G_R, is built on the edges visited in the trials, so it skips unvisited suboptimal action transitions. Basically, backing up a state s is meaningful only when the backup is driven by a value change of the *descendants* of s (the set of states reachable from s) in the solution graph, because such a change might potentially change the value of $V^*(s_0)$. Since G_R skips a lot of edges that are not in the solution graph, most of the backups skipped by TRMax and not by RMax are necessary. So TRMax runs faster than RMax on strongly-connected MDPs.

6. RELATED WORK

The idea of performing value iteration on connected components in their topological order is not new. Our main contribution is to extend its applicability to the learning setting. The procedure described above is roughly outlined (on page 75) in the paper by [2]. It is streamlined and fully developed into the TVI algorithm and analyzed in full by [4].

In Prioritized Sweeping [8], states are prioritized according to their absolute Bellman error and backed up in priority order. Consider an MDP with connected components C_1, C_2, and C_3, connected in a chain, and states s_1, s_2, and s_3 in those components, respectively. Suppose that for actions a, a', and a'', $T_a(s_3|s_1) > 0$, $T_{a'}(s_3|s_2) > 0$, and $T_{a''}(s_2|s_1) > 0$. Suppose that the priority of backing up s_1 is always higher than the priority of s_2. When s_3 is backed up, s_1's value will be recomputed, and then s_2's value, which change s_1's. The situation is more complex when more components proceeds C_1. In TVI, the value for state s_3 is computed exactly (or closely approximated) before it is used to compute the values of s_2 and s_1, so it saves a lot of premature backups on s_2 and s_1. Wingate and Seppi [15] extended the notion of Prioritized Sweeping to General Prioritized Solvers. They consider a variety of prioritization schemes, and introduce the notion of partition. They do not, however, mention partitions via SCCs. They discuss topological order on vertices in a cyclic graph, and focus on approximating a topological order. Within a connected component, it might be possible to use one of their priority metrics to improve TVI.

7. CONCLUSION

We propose a practical method to speed up RL approaches for MDPs. By learning successor-predecessor information of MDP models during learning trials, we are able to construct a reachability graph that restores the dominating graphical structures of the original MDP. Using the topological order of SCCs in this reachability graph can help us either initiate useful future trials (model-free learning), or perform backups wisely (model-based learning). On all the problems tested, TQL and TRMax consistently outperformed their nontopological counterparts by a constant factor. We proved that it is safe to only consider the reachability graph instead of the original MDP as long as our initial learning is sufficient. Therefore, the scope of the problems solvable by these algorithms has been enlarged.

8. REFERENCES

[1] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[2] C. Boutilier, T. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *J. of Artificial Intelligence Research*, 11:1–94, 1999.

[3] R. I. Brafman and M. Tennenholtz. R-max - a general polynomial time algorithm for near-optimal reinforcement learning. *J. of Machine Learning Research*, 3:213–231, 2002.

[4] P. Dai and J. Goldsmith. Topological value iteration algorithm for Markov decision processes. In *Proc. IJCAI-07*, pages 1860–1865, 2007.

[5] E. Hansen. *Finite Memory Control of Partially Observable Systems*. PhD thesis, University of Massachusetts, Amherst, 1998.

[6] S. M. Kakade. *On the sample complexity of reinforcement learning*. PhD thesis, Gatsby Computational Neuroscience Unit, University College London, 2003.

[7] M. Kearns and S. Singh. Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49(2-3):209–232, 2002.

[8] A. Moore and C. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130, 1993.

[9] R. Munos and A. Moore. Influence and variance of a Markov chain : Application to adaptive discretization in optimal control. In *Proc. of IEEE Conference on Decision and Control*, 1999.

[10] D. J. Pearce and P. H. Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *ACM J. of Experimental Algorithmics*, 11:1.7, 2007.

[11] A. L. Strehl and M. L. Littman. A theoretical analysis of model-based interval estimation. In *Proc. of ICML-05*, pages 856–863, 2005.

[12] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

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

[14] C. J. Watkins and P. Dayan. Q-Learning. *Machine Learning*, 8(3-):279–292, 1992.

[15] D. Wingate and K. D. Seppi. Prioritization methods for accelerating MDP solvers. *J. of Machine Learning Research*, 6:851–881, 2005.

Transfer of Task Representation in Reinforcement Learning using Policy-based Proto-value Functions *

(Short Paper)

Eliseo Ferrante
Dept. of Electronics and Information,
Politecnico di Milano
piazza Leonardo Da Vinci, 32,
20133 Milan, Italy
eliseo.ferrante@mail.polimi.it

Alessandro Lazaric
Dept. of Electronics and Information,
Politecnico di Milano
piazza Leonardo Da Vinci, 32,
20133 Milan, Italy
lazaric@elet.polimi.it

Marcello Restelli
Dept. of Electronics and Information,
Politecnico di Milano
piazza Leonardo Da Vinci, 32,
20133 Milan, Italy
restelli@elet.polimi.it

ABSTRACT

Reinforcement Learning research is traditionally devoted to solve single-task problems. Therefore, anytime a new task is faced, learning must be restarted from scratch. Recently, several studies have addressed the issue of reusing the knowledge acquired in solving previous related tasks by transferring information about policies and value functions. In this paper, we analyze the use of proto-value functions under the transfer learning perspective. Proto-value functions are effective basis functions for the approximation of value functions defined over the graph obtained by a random walk on the environment. The definition of this graph is a key aspect in transfer transfer problems in which both the reward function and the dynamics change. Therefore, we introduce *policy-based* proto-value functions, which can be obtained by considering the graph generated by a random walk guided by the optimal policy of one of the tasks at hand. We compare the effectiveness of *policy*-based and standard proto-value functions, on different transfer problems defined on a simple grid-world environment.

Categories and Subject Descriptors

I.2 [**Artificial Intelligence**]: Learning

General Terms

Spectral graph theory

Keywords

Reinforcement Learning, Transfer Learning, Proto-value functions

1. INTRODUCTION

Reinforcement Learning (RL) [9] is a very general learning paradigm. Nonetheless, for each new task, the learning is restarted from scratch and this may lead to prohibitive complexity (*curse of dimensionality*). The goal of transfer learning is to design algorithms able to extract and reuse the knowledge learned in one or more tasks to efficiently develop an effective solution for a new task. Many works of transfer in RL relied on the option framework [8, 4], by learning options that can be profitably reused in a wide range of tasks. Another category of transfer approaches involves transfer of value functions and policies across tasks defined over different domains (i.e., with different state and action spaces). The approach proposed in [10] uses a transfer functional to map the value function of the source task to a corresponding value function for the target task. Finally, in [11], the transfer of policies via inter-task mappings across tasks defined on arbitrary domains is considered.

In this paper, we focus on the problem of learning and transferring the common representation underlying the optimal value functions of a set of related tasks. In particular, we build on the proto-value functions (PVF) framework [6], that provides a technique for the automatic extraction of a set of basis functions based on spectral graph theory. Unlike other works on transfer with PVFs [3], in which matrix perturbation theory and Nyström methods are adopted for domain transfer problems, we focus on the problem in which both state and action spaces are shared across all the tasks but both the dynamics and the reward function may vary. The PVF method is defined under the assumption that the function to be approximated can be effectively represented on a graph. Therefore, in the context of transfer, it is important to build a graph that captures both the dynamics and the reward function. For this reason, we introduce policy-based PVFs obtained from a graph built by considering the optimal policy of one of the *source* tasks at hand.

The rest of the paper is organized as follows. In Section 2, we review the proto-value functions framework. In Section 3, we give the definition of the transfer problem and we introduce the policy-based proto-value functions. In Section 4, we compare original PVFs to policy-based PVFs in a grid world transfer problem. Finally, in Section 5 we conclude and we discuss some possible future directions.

2. PROTO-VALUE FUNCTIONS

RL problems are formally defined as a Markov Decision Process (MDP), described as a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$, where $\mathcal{S}$ is

*

Cite as: Transfer of Task Representation in Reinforcement Learning using Policy-based Proto-value Functions (Short Paper), Eliseo Ferrante, Alessandro Lazaric, Marcello Restelli, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp. 1329-1332.

the set of states, $\mathcal{A}$ is the set of actions, $\mathcal{T}^a_{ss'}$ is the transition model that specifies the transition probability from state s to state s' when action a is taken, and $\mathcal{R}^a_s$ is the reward function. The policy of the agent is defined as a function $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0; 1]$ that prescribes the probability to take an action in each state. The goal of the agent is to learn the optimal policy π^* that maximizes the reward received in the long run. Furthermore, it is possible to compute the optimal action-value function $Q^*(s, a)$, that is, the expected sum of discounted rewards in each state obtained by following π^*. In many practical applications it is unfeasible to store the value function with a distinct value for each state-action (*curse of dimensionality*). The most common approach to face this problem is to use linear function approximators for the action value function:

$$\widehat{Q}(s, a) = \sum_{i=1}^{k} \phi_i(s, a)\theta_i,$$

where $[\theta_1, \ldots, \theta_k]$ is the weights vector to be learned, $[\phi_1 \ldots \phi_k]$ are the basis functions, where $\phi_i : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}, i \in 1, \ldots, k$ is a basis function defined on the state-action space. Most of the RL algorithms consider a set of hand-coded basis functions (e.g., RBFs), while the learning process learns the weights vector that minimizes the approximation error. On the other hand, the PVF framework [6] provides an algorithm for the automatic extraction of a set of basis functions based on spectral graph theory [2]. The basic intuition is that the value functions can be approximated by a set of orthonormal basis computed from the Laplacian of the graph obtained by a random walk on the environment at hand. The Representation Policy Iteration (RPI) algorithm consists in two phases: the representation learning phase and the control learning phase. In the representation learning phase, PVFs are extracted. In particular, an undirected or directed weighted graph $G = \langle N, E, W \rangle$ that reflects the topology of the task is built, where N is the set of nodes (i.e. either states or state-action pairs), E the set of edges and W the matrix containing the weights w_{uv} between each pair of nodes $u, v \in N$. Subsequently, spectral analysis of the graph is performed, extracting the eigenvectors of some graph operator, that is the PVFs. The most used graph operator is the graph Laplacian, that in turns can be defined in many ways, with the most common being the normalized Laplacian definition:

$$\mathcal{L} = I - D^{-1/2} W D^{-1/2},$$

where I is the identity matrix and D is a diagonal matrix called the *valency matrix*, whose entries d_{uu} contain the degree of a node $d_{uu} = \sum_{v \in N} w_{uv}$. Finally, in the control learning phase, LSPI [5] is used to learn the weights vector.

One of the most critical part in the previous algorithm is the construction of the graph used to extract the PVFs. The agent explores the environment using a sampling policy π^σ and a set of sample transitions $\langle s, a, s', r \rangle$ is collected. In general, the sampling policy is the random policy $\pi^\sigma = \pi^R$. Subsequently, a graph can be constructed alternatively by taking into account the estimated transition model of the problem, i.e. by considering the random walk P (containing the probability of going from state s to s') computed as:

$$W \equiv P_s^{s'} = \sum_{a \in \mathcal{A}} \pi^\sigma(s, a)\mathcal{T}^a_{ss'},$$

or by defining a suitable distance function $d(s_i, s_j), s_i, s_j \in \mathcal{S}$ and using an exponential weighting $e^{-d(s_u, s_v)^2}$ to assign weights to each edge on the graph $(u, v) \in E$. In this paper, we focus on graphs defined over the state-action space [7].

3. POLICY-BASED PVFS FOR TRANSFER

We consider the following transfer learning problem. Let $\langle \mathcal{S}, \mathcal{A}, \mathbb{T}, \mathbb{R} \rangle$ with $\mathbb{T} = \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ and $\mathbb{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$ be a family of MDPs sharing the same state space $\mathcal{S}$ and action space $\mathcal{A}$ but with different transition models and reward functions. We define two probability distributions: $\mathcal{Q}_\mathcal{T} : \mathbb{T} \rightarrow [0, 1]$ and $\mathcal{Q}_\mathcal{R} : \mathbb{R} \rightarrow [0, 1]$, used to select the transition model and reward function respectively. In particular, we consider the scenario with one source task, from which the representation knowledge is extracted, and one or more target tasks, where learning exploiting transfer occurs. Both source and target tasks are drawn according to $\mathcal{Q}_\mathcal{T}$ and $Q_\mathcal{R}$. In the following, we distinguish between *goal transfer* and *dynamics transfer*. In goal transfer we assume that the goal changes between the source and the target task, whereas the transition model remains unchanged. In dynamics transfer the two tasks share the same goal, whereas the dynamics is different.

3.1 Policy-based Proto-value Functions

The basic assumption underlying the original PVF framework (whose PVFs will be denoted as *dynamics-based* PVFs) is that the optimal value function V^* can be represented on the graph G obtained through a random walk following a fully random policy π^R on the task. In goal transfer, we assume that all the optimal value functions can be well approximated by the basis of the dynamics-based graph, i.e., the one that captures the dynamics of the environment but completely ignores the reward functions. However, in the more general case, both the transition model and the reward function may vary. Hence, PVFs should be extracted taking into account both of them. Unfortunately, it is not possible to use the reward function in the construction of graphs directly. Nonetheless, it is possible to bias the exploration of the environment towards the optimal policy of the source task, thus indirectly taking its reward function into account. This yields to a new sampling policy which is different from the random policy $\pi^\sigma \neq \pi^R$. As a result, we can compute a new kind of graph based on the new sampling policy. Such graph will be denoted as *policy-based graph*.

We consider a task with a completely connected but stochastic dynamics consisting of 5 states and with the goal in the center, denoted with `X`. A dynamics-based graph would be the one in Figure 1-*(top)*. On the other hand, a policy-based graph should take the goal into account. To strengthen the policy contribution, we compute its t-th power to get the distribution after t steps and we obtain the graph in Figure 1-*(center)*. This graph captures information about the goal, since only the weights on the edges directed towards the goal are very high. However, dynamics information is completely lost. Hence, we introduce averaged graphs (Figure 1-*(bottom)*). This new graph keeps information about both the goal and the dynamics.

Following these observations, we propose the state-action graph construction method. The method takes *policy bias factor* δ as input parameter used to adjust the bias towards the optimal policy π^* of the source task in the construction of the graph. In particular, we set the sampling policy π^σ

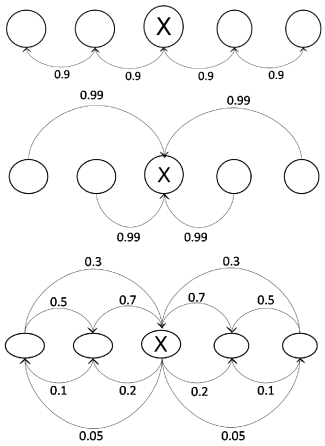


Figure 1: An example of dynamics-based graph (top), policy-based graph (center) and averaged-graph (bottom)

to π^R with probability $1-\delta$ and to π^* with probability δ. When $\delta = 0$, the optimal policy is not used and dynamics-based PVFs are extracted. On the other hand, when δ is close to 1, the policy contribution is very strong and the walk (and hence the graph) is strongly leaned towards the goal. The sampling policy π^σ is used to compute the initial weight matrix for the state-action graph [7] by setting each entry to

$$W((s_i, a_k), (s_j, a_l)) = \mathcal{T}_{s_i s_j}^{a_k} \pi^\sigma(a_l|s_j),$$

$\forall s_i, s_j \in \mathcal{S}, a_k, a_l \in \mathcal{A}$. Subsequently, the graph averaging is done. Here, we use the *time-averaged transition probability matrix* [1] with discounting, thus leading to the graph

$$W_{\tilde{t}} = \sum_{i=1}^{t} \frac{W^i(1-\gamma)\gamma^{i-1}}{1-\gamma^t}.$$

Finally, the graph needs to be symmetrized, especially when using the Laplacian as defined on undirected graphs. With our method, the amount of biasing towards the policy is controlled by the δ parameter.

4. EXPERIMENTAL RESULTS

In order to compare the learning performance of dynamics- and policy-based PVFs in transfer problems, we consider the three-rooms grid-world domain used in [6]. This consists in a stochastic environment, where each action is successful with probability 0.9, whereas with probability 0.1 the agent stands still. In all experiments we use state-action graphs and 15,000 samples during both the representation and learning phase. LSPI parameters are: discount factor $\gamma = 0.9$, the maximum number of iterations is 16 and $\epsilon = 0,001$.

4.1 Goal Transfer Experiment

We first perform goal transfer experiments, in which all the tasks share the same dynamics but have different reward functions. We extract a total of 24 state-action dynamics-based PVFs. We would expect dynamics-based PVFs to perform well, since they effectively capture the dynamics of the task. In the first experiment we consider one source task

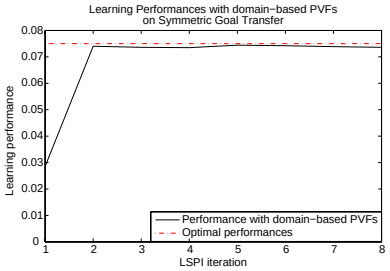


Figure 2: Learning performance in goal transfer obtained by moving the goal in a symmetric position

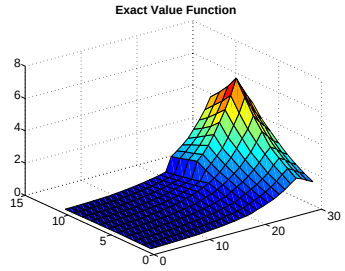


Figure 3: A value function with nonlinearities due to the reward function and not the transition model.

with the goal in the upper right angle of the grid-world, and one target task with the goal in the upper left corner. The exact value function of the source task presents some nonlinearities in the regions that are close to the walls. In [6] it is shown that dynamics-based PVFs can effectively capture those nonlinearities. Learning performance are reported in Figure 2. In this case, the two tasks are completely unrelated in terms of their goal and their optimal policy, whereas their dynamics is the same. Results show that dynamics-based PVFs can effectively achieve goal transfer in this case.

We now consider a goal transfer problem in which the reward functions are obtained by perturbation of the reward function of a source task. In the target task the goal is placed at $(8, 27)$, close to the upper-right corner and the corresponding optimal value function is reported in Figure 3. It is interesting to notice that, in this case, the value function has many nonlinearities that are not related to the dynamics of the environment but that are generated by the reward function. Thus, the dynamics-based PVFs are likely to fail to approximate the optimal value functions of target tasks that share these characteristics with the source task. On the other hand, the policy-based PVFs generated from the graph obtained by biasing the exploration towards the optimal policy of the source task, better capture the particular shape of the value functions to approximate.

We consider 9 target tasks where the goal is moved around the goal of the source task (including itself). We plot the average learning performances of dynamics-based PVFs and policy-based PVFs obtained with policy bias factor $\delta = 0.75$, and we compare them with the average optimal performance. As it can be noticed in Figure 4-*(left)*, policy-based PVFs performs better than dynamics-based PVFs. This is because policy-based PVFs help to capture and transfer the information about the nonlinearities close to the goal.

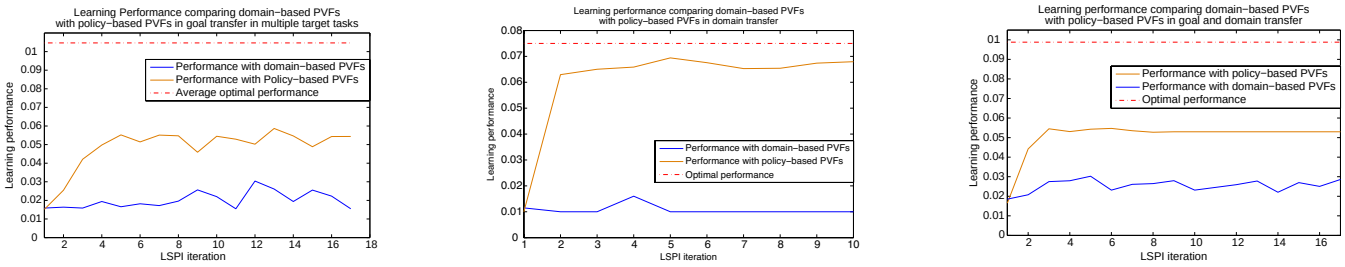


Figure 4: Learning performance in goal (*left*), dynamics (*center*), and goal-dynamics (*right*) transfer

4.2 Dynamics Transfer Experiment

We consider a dynamics transfer experiment in which tasks are strictly related in terms of their optimal policy but with different dynamics. Furthermore, we assume that the shared representation can be compactly extracted by using a low number of PVFs (6 in the experiment). The goal is placed in the upper-right corner in both tasks. The source task has a dynamics which is "tilted" in the opposite direction of the goal. This means that the probability of success of actions that aims in the opposite direction of the goal are higher than the one aiming towards the goal. In the target task, the dynamics is unchanged, with actions having the same probability of success. Figure 4-*(center)* compares the learning performance of dynamics-based PVFs with those of policy-based PVFs obtained with $\delta = 0.75$. As it can be noticed, policy-based PVFs outperform dynamics-based PVFs in this transfer experiment. This is because the two tasks share the representation about their optimal policy, and this can be better captured using policy-based PVFs.

4.3 Goal-Dynamics Transfer Experiment

In the final experiment we consider a source task whose dynamics is tilted towards the bottom-left direction and a goal at $(8, 27)$. We consider three target tasks: *(i)* standard untilted dynamics and the goal at $(8, 28)$, *(ii)* dynamics tilted towards north and the goal at $(9, 27)$, and *(iii)* dynamics tilted towards south and the goal at $(9, 26)$. We consider 6 PVFs and $\delta = 0.5$. Figure 4-*(right)* compares the learning performance of dynamics-based PVFs with those of policy-based PVFs. Optimal performances are reported as well. Also in this case, policy-based PVFs outperform dynamics-based PVFs. By transferring information about the optimal policy, we are able both to counter-balance the effect of the nonlinearity close to the goal and the change in the dynamics.

5. CONCLUSIONS

In this paper, we focused on the problem of transfer in terms of the extraction of a set of basis functions that can be profitably used for the approximation of the optimal value functions of a set of related tasks. In particular, building on the PVF framework, we showed that, in the transfer problem, the graph construction method should take into consideration both the transition model and the reward function. Hence, we proposed policy-based PVFs, extracted using an averaged discounted graph obtained through an exploration biased towards the optimal policy of the source task. In case of goal transfer, dynamics-based PVFs achieve effective transfer whenever the optimal value functions has nonlinearities directly related with the intrinsic structure of the environment. On the other hand, when optimal value functions present some nonlinearities *caused* by the reward function, the use of the optimal policy of the source target leads to policy-based PVFs that can better approximate the target functions. Furthermore, the dynamics transfer experiments showed that the policy-based PVFs can improve transfer capabilities in cases where the source and the target task share a similar optimal policy, but the dynamics are different.

A direction for future work is to define a method to incrementally adapt the initial set of PVFs according to the target tasks at hand, thus improving their approximation capabilities on the tasks that must be actually solved.

6. REFERENCES

[1] A. T. Bharucha-Reid. *Elements of the Theory of Markov Processes and Their Applications*. Dover Publications, 1997.

[2] F. R. Chung. *Spectral Graph Theory*. Amer Mathematical Society, 1997.

[3] K. Ferguson and S. Mahadevan. Proto-transfer learning in markov decision processes using spectral methods. In *ICML Workshop on Transfer Learning*, 2006.

[4] G. Konidaris and A. G. Barto. Building portable options: Skill transfer in reinforcement learning. In *IJCAI*, pages 895–900, 2007.

[5] M. G. Lagoudakis and R. Parr. Least-squares policy iteration. *JMLR*, 4:1107–1149, 2003.

[6] S. Mahadevan and M. Maggioni. Proto-value functions: A laplacian framework for learning representation and control in markov decision processes. *JMLR*, 8:2169–2231, 2007.

[7] S. Osentoski and S. Mahadevan. Learning state-action basis functions for hierarchical mdps. In *ICML '07*, pages 705–712, 2007.

[8] T. J. Perkins and D. Precup. Using options for knowledge transfer in reinforcement learning. Technical report, University of Massachusetts, Amherst, MA, USA, 1999.

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

[10] M. E. Taylor, P. Stone, and Y. Liu. Value functions for RL-based behavior transfer: A comparative study. In *AAAI*, pages 880–885, July 2005.

[11] M. E. Taylor, P. Stone, and Y. Liu. Transfer learning via inter-task mappings for temporal difference learning. *JMLR*, 8:2125–2167, 2007.

Reinforcement Learning for DEC-MDPs with Changing Action Sets and Partially Ordered Dependencies

(Short Paper)

Thomas Gabel and Martin Riedmiller
Neuroinformatics Group, Department of Computer Science, Institute of Cognitive Science
University of Osnabrück, 49069 Osnabrück, Germany
{thomas.gabel|martin.riedmiller@uos.de}

ABSTRACT

Decentralized Markov decision processes are frequently used to model cooperative multi-agent systems. In this paper, we identify a subclass of general DEC-MDPs that features regularities in the way agents interact with one another. This class is of high relevance for many real-world applications and features provably reduced complexity (NP-complete) compared to the general problem (NEXP-complete). Since optimally solving larger-sized NP-hard problems is intractable, we keep the learning as much decentralized as possible and use multi-agent reinforcement learning to improve the agents' behavior online. Further, we suggest a restricted message passing scheme that notifies other agents about forthcoming effects on their state transitions and that allows the agents to acquire approximate joint policies of high quality.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed AI

General Terms

Algorithms, Design, Theory

Keywords

Decentralized MDPs, Interaction, Communication

1. INTRODUCTION

Research on distributed control of cooperative multi-agent systems has received a lot of attention during the past years. Among the models discussed in the literature, the DEC-MDP framework [4], that is characterized by each agent having only a partial view of the global system state, has been frequently investigated. In this regard, it has been shown that the complexity of general DEC-MDPs is NEXP-complete, even for the benign case of two cooperative agents.

Decentralized decision-making is required in many real-life applications. Examples include distributed sensor networks, teams of autonomous robots, or production planning and optimization scenarios. Being important for practice, the enormous computational complexity of solving DEC-MDPs conflicts with the fact that real-world tasks do typically have a considerable problem size. Therefore, in this paper we will identify a subclass of general DEC-MDPs that features regularities in the way the agents interact with one another. For this class, we can show that the complexity of optimally solving an instance of such a DEC-MDP is provably lower (NP-complete) than the general problem (Section 2). Moreover, we analyze methods for the agents to benefit from partially knowing about the state transition dependencies. To this end, we propose the use of a restricted message passing scheme that notifies other agents about forthcoming effects on their state transitions and we investigate its usefulness (Section 3). For adapting the agents' policies, we propose the usage of a multi-agent reinforcement learning (RL) approach, where the agents are independent learners and do their learning online which we evaluate in the context of larger-sized scheduling problems (Section 4).

Cite as: Reinforcement Learning for DEC-MDPs with (...) (Short Paper), Thomas Gabel, Martin Riedmiller, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp.1333-1336.

2. PROBLEM DESCRIPTION

2.1 DEC-MDP Framework

Basically, the subclass of problems we are focusing on in this paper may feature an arbitrary number of agents whose actions influence, besides their own, the state transitions of maximally one other agent in a specific manner. We embed the problem settings of our interest into the framework of decentralized Markov decision processes (DEC-MDP) [4].

Definition 1. A factored m-agent DEC-MDP M is defined by a tuple

$$\langle Ag, S, A, P, R, \Omega, O \rangle$$

where $Ag = \{1, \ldots, m\}$ is a set of agents, S is the set of world states that can be factored into m components $S = S_1 \times \cdots \times S_m$ (S_i belong to one of the agents each), $A = A_1 \times ... \times A_m$ is the set of joint actions to be performed by the agents ($a = (a_1, \ldots, a_m) \in A$ denotes a joint action that is made up of elementary actions a_i taken by agent i), P is the transition function with $P(s'|s, a)$ denoting the probability that the system arrives at state s' upon executing a in s, R is a reward function with $R(s, a, s')$ denoting the reward for executing a in s and transitioning to s'.

$\Omega = \Omega_1 \times \cdots \times \Omega_m$ is the set of all observations of all agents ($o = (o_1, \ldots, o_m) \in \Omega$ denotes a joint observation with o_i as the observation for agent i) and O denotes the observation function that determines the probability $O(o_1, \ldots, o_m|s, a, s')$ that agent 1 through m perceive observations o_1 through o_m.

Moreover, M is jointly fully observable, i.e. the current state is entirely determined by the amalgamation of all agents' observations: if $O(o|s,a,s') > 0$, then $Pr(s'|o) = 1$.

We refer to the agent-specific components $s_i \in S_i$, $a_i \in A_i$, and $o_i \in \Omega_i$ as the local state, action, and observation of agent i. A joint policy π is a set of local policies $\langle \pi_1, \ldots, \pi_m \rangle$ each of which is a mapping from agent i's sequence of local observations to local actions, i.e. $\pi_i : \overline{\Omega}_i \rightarrow A_i$. In the following, we allow each agent to fully observe its local state, i.e. we consider factored m-agent DEC-MDPs with local full observability which implies that for all agents i and for all local observations o_i there is a local state s_i such that $Pr(s_i|o_i) = 1$. Note that joint full observability and local full observability of a DEC-MDP do generally not imply full observability, which would allow us to consider the system as a single large MDP and to solve it with a centralized approach.

A factored m-agent DEC-MDP is called *reward independent*, if there exist local functions R_1 through R_m, each depending on local states and actions of the agents only, as well as a function r that amalgamates the global reward value from the local ones, such that maximizing each R_i individually also yields a maximization of r. Throughout this paper, we will consider the global reward to be the sum of the local ones.

If, in a factored m-agent DEC-MDP, each agent's observation depends only on its current and next local state and on its action, then that DEC-MDP is called *observation independent*. Then, in combination with local full observability, the observation-related components Ω and O are redundant and can be removed from Definition 1.

While the DEC-MDPs of our interest are observation independent and reward independent, they are *not* transition independent. That is, the state transition probabilities of one agent may very well be influenced by another agent.

2.2 Variable Action Sets

We assume that there are some regularities that determine the way local actions exert influence on other agents' states. First, we assume that the sets of local actions A_i change over time.

Definition 2. A factored m-agent DEC-MDP is said to feature *changing action sets*, if the local state of agent i is fully described by the set of actions currently selectable by i ($s_i = A_i \setminus \{\alpha_0\}$) and A_i is a subset of the set of all available local actions $\mathcal{A}_i = \{\alpha_0, \alpha_{i1} \ldots \alpha_{ik}\}$, thus $S_i = \mathcal{P}(\mathcal{A}_i \setminus \{\alpha_0\})$. Here, α_0 represents a null action that does not change the state and is always in A_i. We abbreviate $\mathcal{A}_i^r = \mathcal{A}_i \setminus \{\alpha_0\}$.

Concerning state transition dependencies, one can distinguish between dependent and independent local actions. While the former influence an agent's local state only, the latter may additionally influence the state transitions of other agents. As pointed out, our interest is in non-transition independent scenarios. In particular, we assume that an agent's local state can be affected by an arbitrary number of other agents, but that an agent's local action affects the local state of maximally one other agent.

Definition 3. A factored m-agent DEC-MDP is said to have *partially ordered transition dependencies*, if there exist functions σ_i for each agent i with

1. $\sigma_i : \mathcal{A}_i^r \rightarrow Ag \cup \{\emptyset\}$ and
2. $\forall \alpha \in \mathcal{A}_i^r$ the directed graph $G_\alpha = (Ag \cup \{\emptyset\}, E)$ with $E = \{(j, \sigma_j(\alpha)) | j \in Ag\}$ is acyclic and contains a path of length m

and it holds $P(s_i'|s, (a_1 \ldots a_m), (s_1' \ldots s_{i-1}', s_{i+1}' \ldots s_m')) = P(s_i'|s_i, a_i, \{a_j \in \mathcal{A}_j | i = \sigma_j(a_j), j \neq i\})$.

The influence exerted on another agent always yields an extension of that agent's action set: If $\sigma_i(\alpha) = j$, i takes local action α, and the execution of α has been finished, then α is added to $A_j(s_j)$, while it is removed from $A_i(s_i)$.

So, the σ_i functions indicate whose other agents' state is affected when agent i takes a local action. Also, condition 2 in Definition 3 implies that for each local action α there is a total ordering of its execution by the agents. While these orders are total, the global order in which actions are executed is only partially defined by that definition and subject to the agents' policies. Lemma 1 states that for the problems considered any local action may appear only once in an agent's action set and, thus, may be executed only once.

Lemma 1. In a factored m-agent DEC-MDP with changing action sets and partially ordered transition dependencies it holds: $\forall i \in Ag$, $\forall \alpha \in \mathcal{A}_i^r$, $\forall t \in \{1 \ldots T\}$ and $\forall \overline{s}_i = (s_i^1 \ldots s_i^t)$: If there is a t_a ($1 \leq t_a < T$) with $\alpha \in s_i^{t_a}$ and a t_b ($t_a < t_b \leq T$) with $\alpha \notin s_i^{t_b}$, then $\forall \tau \in \{t_b \ldots T\} : \alpha \notin s_i^\tau$.

Proof. The proofs of this and of the following lemmas are omitted due to space constraints. □

2.3 Implications on Complexity

While the complexity of solving general DEC-MDPs is known to be NEXP-complete [4], several authors have identified subclasses of the general problem that provably yield lower (NP-complete) complexity (e.g. [3, 6, 2]). As shown in [9], a key factor that determines whether the problem complexity is reduced to NP-completeness is whether the agents' histories can be compactly represented. In particular, there must exist an encoding function $Enc_i : \overline{\Omega}_i \rightarrow E_i$ such that

1. a joint policy $\pi = \langle \pi_1 \ldots \pi_m \rangle$ with $\pi_i : E_i \rightarrow \mathcal{A}_i$ is capable of maximizing the global value and
2. the encoding is polynomial, i.e. that $|E_i| = O(|S|^{c_i})$.

For our class of factored m-agent DEC-MDPs with changing action sets and partially ordered transition dependencies we can define an encoding that adheres to both of these conditions, thus showing that those problems are NP-complete.

The interaction history of a DEC-MDP is the sequence of local observations $\overline{o}_i \in \overline{\Omega}_i$ which in our case corresponds to the history of local states $\overline{s}_i \in \overline{S}_i = \times_{t=1}^{T} S_i$, since we assume local full observability (recall that $S_i = \mathcal{P}(\mathcal{A}_i^r)$).

Definition 4. Given a local action set $\mathcal{A}_i = \{\alpha_0 \ldots \alpha_k\}$ and a history $\overline{s}_i = (s_i^1 \ldots s_i^t) \in \overline{S}_i$ of local states of agent i, the encoding function is defined as $Enc_i : \overline{S}_i \rightarrow E_i$ with $E_i = C_{\alpha_1} \times \cdots \times C_{\alpha_k}$ and $C_{\alpha_j} = \{0, 1, 2\}$. And it holds $Enc_i(\overline{s}_i) = (c_{i,\alpha_1} \ldots c_{i,\alpha_k}) \in E_i$ with

$$c_{i,\alpha_j} = \begin{cases} 0 & \text{if } \nexists \tau \text{ with } \alpha_j \in s_i^\tau \\ 1 & \text{if } \alpha_j \in s_i^t \\ 2 & \text{else} \end{cases}$$

Basically, the encoding guarantees that each agent knows whether some local action has not yet been, is currently, or had been in its action set. Proving that this encoding is capable of representing the optimal policy and showing that it is a polynomial encoding, we can conclude that the subclass of DEC-MDPs we identified is NP-complete.

Lemma 2. Enc_i provides a polynomial encoding of agent i's observation history.

Lemma 3. Enc_i provides an encoding of agent i's observation history such that a joint policy $\pi = \langle \pi_1 \dots \pi_m \rangle$ with $\pi_i : E_i \rightarrow \mathcal{A}_i$ is sufficient to maximize the global value.

As deciding a polynomially encodable DEC-MDP is NP-hard [9], solving a factored m-agent DEC-MDP with changing action sets and partially ordered dependencies is so, too.

3. RESOLVING DEPENDENCIES

Besides using an encoding of an agent's interaction history (Section 2), there are other options for exploiting the regularities in the transition dependencies of the class of DEC-MDPs we identified that.

3.1 Reactive Policies and Their Limitations

An agent taking its action based solely on its most recent local observation $s_i \subseteq \mathcal{A}_i$ is in general not able to contribute to optimal joint behavior: It faces difficulties in assessing the value of its idle action α_0. Because a purely reactive agent has no information related to other agents and dependencies at all, it is incapable of properly distinguishing when it is favorable to remain idle and when not. For these reasons, we exclude α_0 from all $\mathcal{A}_i$ for purely reactive agents.

Definition 5. For an m-agent DEC-MDP with changing action sets and partially ordered transition dependencies, a *reactive policy* $\pi^r = \langle \pi_1^r \dots \pi_m^r \rangle$ consists of m reactive local policies with $\pi_i^r : S_i \rightarrow \mathcal{A}_i^r$ where $S_i = \mathcal{P}(\mathcal{A}_i^r)$.

That is, purely reactive policies always take an action $\alpha \in \mathcal{A}_i(s_i) = s_i$ (except for $s_i = \emptyset$), even if it was better to stay idle and wait for a transition from s_i to some $s_i' = s_i \cup \{\alpha'\}$ induced by another agent, and then execute α' in s_i'.

3.2 Awareness of Dependencies

In Definition 4, we stated that the probability that agent i's local state moves to s_i' depends on that agent's current local state s_i, its action a_i, as well as on the set $\{a_j \in \mathcal{A}_j | i = \sigma_j(a_j), i \neq j\} =: \Delta_i$, i.e. on the local actions of all agents that may influence agent i's transition. Although knowing Δ_i is in general not feasible for each agent, we may enhance the capabilities of a reactive agent i by allowing it to get at least some partial information about this set. For this, we extend a reactive agent's local state space from $S_i = \mathcal{P}(\mathcal{A}_i^r)$ to $\hat{S}_i$ such that for all $\hat{s}_i \in \hat{S}_i$ it holds $\hat{s}_i = (s_i, z_i)$ with $z_i \in \mathcal{P}(\mathcal{A}_i^r \setminus s_i)$. So, z_i is a subset of the set of actions currently *not* in the action set of agent i.

Definition 6. Let $1 \dots m$ be reactive agents acting in a DEC-MDP, as specified in Definition 3, whose local state spaces are extended to $\hat{S}_i$. Assume that current local actions $a_1 \dots a_m$ are taken *consecutively*. Given that agent j decides for $a_j \in A_j(s_j)$ and $\sigma_j(a_j) = i$, let also s_i be the local state of i and $\hat{s}_i$ its current extended local state with $\hat{s}_i = (s_i, z_i)$. Then, the transition dependency between j and i is said to be *resolved*, if $z_i := z_i \cup \{a_j\}$.

Resolving transition dependencies according to Definition 6 means letting agent i know some of those current local actions of other agents by which i's local state will soon be influenced. Since, for the class of problems we are dealing with, inter-agent interferences are always exerted by changing (extending) another agent's action set, in this way agent i gets to know which further action(s) will soon be available in its action set. Integrating this piece of information into i's extended local state description $\hat{S}_i$, i gets the opportunity to willingly stay idle (execute α_0) until the announced action $a_j \in z_i$ enters its action set and can finally be executed.

The notification of agent i, which instructs him to extend its local state component z_i by a_j, may easily be realized by a simple message passing scheme (assuming cost-free communication between agents) that allows agent i to send a single directed message to agent $\sigma_i(\alpha)$ upon the local execution of α. Obviously, this kind of partial resolving of transition dependencies is particularly useful in applications where the execution of atomic actions takes more than a single time step and where, hence, decision-making proceeds asynchronously across agents. Under those conditions, up to half of the dependencies in Δ_i (over all i) may be resolved.

4. DISCUSSION AND EVALUATION

Distributed problem solving in practice is often characterized by a factored system state description where the agents base their decisions on local observations. Also, our assumptions that local actions may influence the state transitions of maximally one other agent and that any action has to be performed only once are frequently fulfilled. Sample real-world applications include scenarios from manufacturing, production planning, or assembly line optimization, where typically the production of a good involves a number of processing steps that have to be performed in a specific order. In a factory, however, usually a variety of products is assembled concurrently, which is why an appropriate sequencing of single operations is of crucial importance for overall performance. Thus, the class of factored m-agent DEC-MDPs with changing action sets and partially ordered transition dependencies covers a variety of such scheduling problems, for example flow-shop and job-shop scheduling scenarios [7]. Beyond that, a big portion of supply chain problems where complex items are assembled through a series of steps are covered. Other practical application domains to which our model is of relevance include network routing (e.g. sub-task of determining the order of forwarding packets), railway traffic control (e.g. task of allowing trains to pull into the station via agent-based track switches), or workflow management.

Joint Policy Acquisition with RL

Solving a DEC-MDP optimally is NEXP-hard and intractable for all except the smallest problem sizes. Unfortunately, the fact that the subclass of DEC-MDPs we identified is in NP and hence simpler to solve, does not rid us from the computational burden implied. So, our goal is not to develop yet another optimal solution algorithm applicable to small problems only, but to use a technique capable of quickly finding approximate solutions in the vicinity of the optimum.

We let the agents acquire their local policies independently of the other agents by repeated interaction with the DEC-MDP and concurrent evolvement of their policies. Our learning approach is made up of alternating data collection and learning stages that are being run concurrently within

all agents. At its core, a specialized variant of a neural fitted Q iteration (NFQ) algorithm [8], enhanced for usage in multi-agent domains, is used that allows the agents to determine a value function over their local state-action spaces. A detailed description of that approach can be found in [5].

Experiments

For the purpose of evaluation, we focus on various job-shop scheduling (JSS) benchmark problems (taken from [1]) that are known to be NP-hard. The goal of scheduling is to allocate a given number of jobs to a limited number of resources such that some objective is optimized. In job-shop scheduling, n jobs must be processed on m machines in a pre-determined order, while minimizing maximum makespan C_{max}, which corresponds to finishing processing as quickly as possible. Each job consists of a specific number of operations that each have to be handled on a certain resource for a certain duration, where the whole job is finished after its last operation has been entirely processed.

JSS problems are suited to be modelled as factored m-agent DEC-MDPs with changing action sets and partially ordered transition dependencies: The world state can be factored, if we assume that to each of the resources one agent i is associated whose local action is to decide which waiting job to process next. Further, the local state of i can be fully described by the changing set of jobs currently waiting for further processing, and after having finished an operation of a job, this job is transferred to another resource, which corresponds to influencing another agent's local state by extending that agent's action set. Examining the formal definition of JSS problems [7], it is obvious that we can also easily define $\sigma_i : \mathcal{A}_i^r \rightarrow Ag \cup \{\emptyset\}$ (see Definition 3) for all agents/resources i and that the corresponding directed graph G_α is indeed acyclic with a path of length m.

The primary concern of the experiments conducted was on an analysis of the three approaches discussed in this paper. We compared agents that independently learn purely reactive policies π_i^r (see Section 3.1) defined over $S_i = \mathcal{P}(\mathcal{A}_i^r)$ that never remain idle when their action set is not empty (RCT), reactive policies $\hat{\pi}_i$ that are partially aware of their dependencies on other agents (being notified about forthcoming influences exerted by other agents, COM), and policies $\pi_i : E_i \rightarrow \mathcal{A}_i$ where E_i is an encoding of that agent's observation history $\overline{S}_i$ according to Definition 4 (ENC).

Findings

Using RCT-agents, only schedules from the class of non-delay schedules $\mathbb{S}_{nd}$ can be created by applying reactive policies. Since $\mathbb{S}_{nd} \subseteq \mathbb{S}_a$ and it is known that the optimal schedule is always in $\mathbb{S}_a$ [7], but not necessarily in $\mathbb{S}_{nd}$, RCT-agents can at best learn the optimal solution from $\mathbb{S}_{nd}$. By contrast, learning with ENC-agents, the optimal solution can in principle be attained, but we found that the time required by our learning approach for this to happen increases significantly due to larger-sized local state spaces.

We also found that the awareness of inter-agent dependencies achieved by partial dependency resolvements via communication in fact realizes a good trade-off between the former two approaches. On the one hand, resolving a transition dependency according to Definition 6, an agent i can become aware of an incoming job. Thus, i may decide to wait for that arrival, instead of starting to execute another job. Hence, also schedules can be created that are not non-delay. On the other hand, very poor policies with unnecessary idle times can be avoided, since a decision to stay idle may be taken only when a future job arrival has been announced.

Averaged over 24 different benchmark instances [1] of varying sizes (up to 15 agents) for which it is known that the optimal solution is not in $\mathbb{S}_{nd}$, the learned policies nearly reach the theoretical optimum (schedule with minimal C_{max}) missing it by 6.18% for RCT-agents, by 9.55% for ENC-agents, and by 4.78% for COM-agents. Dispatching rule based scheduling approaches are clearly surpassed (best conventional scheduling rule reaches a remaining error of 8.59%).

5. CONCLUSION

We have identified a class of cooperative decentralized MDPs that features a number of regularities in the way agents influence the state transitions of other agents. Exploiting the knowledge about these correlations, we have proven that this class of problems is easier to solve (NP-hard) than general DEC-MDPs (NEXP-hard). Subsequently, we have looked at possibilities for modeling memoryless agents and enhancing them by restricted allowance of communication. For solving instances of the DEC-MDP class identified we relied on a coordinated batch-mode reinforcement learning algorithm that facilitates the agents to concurrently and independently learn their local policies of action online.

6. ACKNOWLEDGEMENTS

This research has been supported by the German Research Foundation (DFG) under grant number Ri-923/2-3.

7. REFERENCES

[1] J. Beasley. OR-Library, 2005, `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`.

[2] R. Becker, S. Zilberstein, and V. Lesser. Decentralized Markov Decision Processes with Event-Driven Interactions. In *Proceedings of AAMAS 2004*, pages 302–309, New York, USA, 2004. ACM Press.

[3] R. Becker, S. Zilberstein, V. Lesser, and C. Goldman. Solving Transition Independent Decentralized MDPs. *Journal of AI Research*, 22:423–455, 2004.

[4] D. Bernstein, D. Givan, N. Immerman, and S. Zilberstein. The Complexity of Decentralized Control of Markov Decision Processes. *Mathematics of Operations Research*, 27(4):819–840, 2002.

[5] T. Gabel and M. Riedmiller. Adaptive Reactive Job-Shop Scheduling with Learning Agents. *International Journal of Information Technology and Intelligent Computing*, 2(4), 2008.

[6] C. Goldman and S. Zilberstein. Optimizing Information Exchange in Cooperative Multi-Agent Systems. In *Proceedings of AAMAS 2003*, pages 137–144, Melbourne, Australia, 2003. ACM Press.

[7] M. Pinedo. *Scheduling. Theory, Algorithms, and Systems*. Prentice Hall, 2002.

[8] M. Riedmiller. Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method. In *Proceedings of ECML 2005*, Porto, Portugal, 2005. Springer.

[9] J. Shen, R. Becker, and V. Lesser. Agent Interaction in Distributed POMDPs and Implications on Complexity. In *Proceedings of AAMAS 2006*, pages 529–536, Hakodate, Japan, 2006. ACM Press.

Using Adaptive Consultation of Experts to Improve Convergence Rates in Multiagent Learning

(Short Paper)

Greg Hines
Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
ggdhines@cs.uwaterloo.ca

Kate Larson
Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
klarson@cs.uwaterloo.ca

ABSTRACT

We present a regret-based multiagent learning algorithm which is provably guaranteed to converge (during self-play) to the set of Nash equilibrium in a wide class of games. Our algorithm, FRAME, consults *experts* in order to obtain strategy suggestions for agents. If the experts provide effective advice for the agent, then the learning process will quickly reach a desired outcome. If, however, the experts do not provide good advice, then the agents using our algorithm are still protected. We further expand our algorithm so that agents learn, not only how to play against the other agents in the environment, but also which experts are providing the most effective advice for the situation at hand.

Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]: Multiagent Systems

General Terms

Algorithms, Theory

Keywords

Multiagent Learning, Game Theory

1. INTRODUCTION

How and what agents should learn in the presence of others is one of the important questions in multiagent systems. The problem has been studied from several different perspectives, and in particular has garnished a lot of interest from both the game-theory community (see, for example, [4]) and the AI community (see, for example, [2]).

In this paper we investigate the problem of whether identical agents, who repeatedly play against each other, can learn to play strategies which form a Nash equilibrium (see, for example [2]). In particular, we are interested in settings where there are potentially more than two agents, and where agents have potentially more than just two actions to choose from. We are also interested in ensuring that agents learn to play a best response against stationary opponents.

Our learning procedure, a *Framework for Regret Annealing Methods using Experts* or *FRAME*, is a regret-based learning algorithm for repeated games which combines a greedy random sampling method with consultation of *experts*, that return strategy profiles. More importantly, by consulting carefully chosen experts we can greatly improve the convergence rate to Nash equilibria in self-play, but in the case where the experts do not return useful advice, then we still have guarantees that our algorithm will lead agents to a Nash equilibrium.

Cite as: Using Adaptive Consultation of Experts to Improve Convergence Rates in Multiagent Learning (Short Paper), Greg Hines and Kate Larson, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp.1337-1340.

2. BACKGROUND

An n-player *stage game* is defined as $G = \langle N, A_1, \ldots, A_n, u_1, \ldots, u_n \rangle$ where $N = \{1, 2, \ldots, n\}$ is the set of agents participating in the game, and A_i is the set of possible actions that agent i can take. During the stage game, agents simultaneously choose to play actions and each agent receives a reward based on the joint action $a = (a_1, \ldots, a_n)$. In particular $u_i : A_1 \times \ldots \times A_n \rightarrow \mathbb{R}$ is the utility function for agent i, and so $u_i(a)$ is the reward that agent i receives if the joint action is a. With out loss of generality, we assume $u_i \in [0, 1]$. Agents play *strategies*, where a strategy, σ_i, of agent i is a probability distribution over action space A_i and $\sigma_i(a_j)$ denotes the probability with which agent i chooses to play action $a_j \in A_i$. We let Σ_i denote the strategy space of agent i, and let $\sigma = (\sigma_1, \ldots, \sigma_n) \in \Sigma = \Sigma_1 \times \ldots \times \Sigma_n$ denote a joint strategy. If there exists an action a_j such that $\sigma_i(a_j) = 1$, then σ_i is called a pure strategy. We use the notation $\sigma = (\sigma_i, \sigma_{-i})$ to represent a joint strategy, where σ_{-i} is defined to be equal to $(\sigma_1, \ldots, \sigma_{i-1}, \sigma_{i+1}, \ldots, \sigma_n)$. By abuse of notation, we can define the utility of an agent in terms of a joint strategy $\sigma = (\sigma_i, \sigma_{-i})$ as

$$u_i(\sigma_i, \sigma_{-i}) = \sum_{a \in A} u_i(a) \Pi_{j=1}^{n} \sigma_j(a_j).$$

We assume that agents are self-interested and that they wish to play strategies that maximize their own utility. That is, if all agents but i are playing σ_{-i}, then agent i should play a strategy σ_i that maximizes its utility, i.e. σ_i should be a best response to σ_{-i}. We say that agents' strategies are in (Nash) equilibrium if no agent is willing to change their strategy, given that no other agents change.

DEFINITION 1. *A strategy profile $\sigma^* = (\sigma_1^*, \ldots, \sigma_n^*)$ is a*

Nash equilibrium if for every agent i

$$u_i(\sigma_i^*, \sigma_{-i}^*) \geq u_i(\sigma_i', \sigma_{-i}^*) \ \forall \sigma_i' \neq \sigma_i^*.$$

A strategy profile σ^* is an ϵ-Nash equilibrium if for every agent i, $u_i(\sigma_i^*, \sigma_{-i}^*) \geq u_i(\sigma_i', \sigma_{-i}^*) - \epsilon \ \forall \sigma_i' \neq \sigma_i^*$.

Agents are also able to evaluate their strategy choice by measuring the *regret* they experience from playing a particular strategy.

DEFINITION 2. *Given a joint strategy σ, agent i's regret is $r_i(\sigma) = \max_{\sigma_i' \in \Sigma_i} [u_i(\sigma_i', \sigma_{-i}) - u_i(\sigma_i, \sigma_{-i})]$.*

Given σ, we define the *regret of a game* to be the maximum regret among all agents, i.e. $r(\sigma) = \max_{i \in N}(r_i(\sigma))$.

A repeated game, $\mathcal{G}$, is a game where agents play a specific stage game over and over. At stage t we denote the strategy profile that the agents played by σ^t and the actual action profile that the agents played by a^t. Given σ^t, each agent i is able to compute its immediate regret, $r_i(\sigma^t)$.

As the stage game is repeated, agents gain experience and are able to adjust their strategies so that they fair better against their opponents. In this paper we are interested in learning approaches which use *regret*, and in particular regret-minimization, to guide the agents' strategy adaptations. Our goal is to develop a learning procedure which will converge to an interesting set of strategies for the agents. In particular, we would like to develop an approach such that $r(\sigma^t) \to 0$ as $t \to \infty$ (*i.e.* the process converges to the set of Nash equilibria for the stage game).

Regret-based learning is a broad type of learning that can achieve various degrees of convergence. However, the results for achieving convergence to the set of Nash equilibria are mostly negative. Some positive results have been achieved using *randomized* learning algorithms. One example of this approach is *Experimental Regret Technique (ERT)* [5]. The basic idea of ERT is to have all agents with high regret randomly choose a new strategy, to have all agents with medium regret to slightly modify their current strategy in some systematic way, and to have agents with low regret to keep playing their strategy. Germano and Lugosi further improved upon this technique with their algorithm Annealed Localized Experimental Regret Technique (ALERT) which provably converges to the set of Nash equilibria for almost all games and the set of ϵ-Nash equilibria for all games.

3. FRAME

Although ALERT is theoretically important, there are two main issues which limit its applicability in actual multiagent systems. First, since ALERT is an uncoupled algorithm, agents have almost no information from which they can determine whether they are playing an ϵ-equilibrium. Instead, ALERT's guarantees are in the form of bounds on the probability of *not* being in an ϵ-equilibrium. Second, ALERT uses a naive method for having agents find new strategies. In particular, ALERT has the agents choose new strategies uniformally at random and then checks whether these strategies meet a set of conditions. Our algorithm, a *Framework for Regret Annealing Methods using Experts*, or *FRAME*, is inspired by ALERT but explicitly addresses these two issues, while still maintaining the theoretical guarantees of ALERT.

To address the first issue, FRAME is not a fully uncoupled algorithm. Instead, we assume that the agents' strategies are publicly available to all agents, as is done by several other researchers [2]. We also assume that the maximum regret of all agents is publicly available. Our algorithm will still work without these two assumptions, as it is possible to experimentally determine regret (both for individual agents and overall), but this comes with a substantial increase in the number of iterations required by our algorithm.

Algorithm 1 $FRAME_i$

- σ_i^0 is a strategy picked uniformly at random
for $t = 0, 1, \ldots$ **do**
  - with probability p, β_i^{t+1} is the strategy returned by consulting the expert
  if β_i^{t+1} is not in the bounded region $B(\sigma_i^t, d(r(\sigma^t)))$ or the expert was not consulted **then**
    - β_i^{t+1} is the strategy picked uniformly from $B(\sigma_i^t, d(r(\sigma^t)))$
  end if
  if the regret of β is less than the regret of σ^t **then**
    - $\sigma^{t+1} = \beta^{t+1}$
  else
    - $\sigma^{t+1} = \sigma^t$
  end if
  - τ_i is strategy picked uniformly at random from Σ_i
  if the regret of τ is less than half the regret of σ^{t+1} **then**
    - with probability η, set $\sigma^{t+1} = \tau$.
  end if
end for

To deal with the second issue, FRAME allows an agent, with some probability, to consult an expert, which returns a possible new strategy. Any expert will work, even a malicious one that actively provides bad strategies. If the expert provides good strategies, then FRAME will be able to reduce an agent's regret quickly. If all agents are using FRAME and are consulting good experts, then the convergence rate to a Nash equilibrium greatly improves.

The FRAME algorithm for agent i is shown in Algorithm 1. The algorithm, with respect to agent i, works as follows. Agent i first chooses an initial strategy σ_i^0 uniformly at random from Σ_i. To obtain a new strategy for time $t+1$, FRAME then uses the provided expert, which agent i consults with a provided probability of p, independent of all other agents. If consulted, the expert returns a possible strategy β_i^{t+1}. To provide protection against poor experts, FRAME checks to see if β_i^{t+1} is inside the bounded region $B(\sigma_i^t, d(r^t))$, which is centered on σ_i^t and has a minimum width of $d()$.[1] If β_i^{t+1} is not, or the expert was not consulted, β_i^{t+1} is chosen uniformly at random from the bounded search region. Agent i then calculates $r_i(\beta^{t+1})$. If $r(\beta^{t+1}) < r(\sigma^t)$, then $\sigma^{t+1} = \beta^{t+1}$, otherwise, $\sigma^{t+1} = \sigma^t$. To avoid the off-chance of getting stuck at a locally optimal joint strategy, each agent chooses an alternative strategy τ_i uniformly at random from Σ_i. If the regret at τ is less than half the current regret, then with a given probability η, the game *resets* to τ. This process repeats until the regret is zero.

FRAME's correctness is provided by Proposition 1.

PROPOSITION 1. *If $\eta > 0$, then as t approaches infinity, σ^t approaches the set of Nash equilibria.*

The proof is omitted due to space limitations.

[1] $d()$ may be any function so long as $d(x) > 0$, for $x > 0$.

0, 0	1, 0	0, 1
0, 1	0, 0	1, 0
1, 0	0, 1	0, 0

Figure 1: Shapley's Game.

It should be noted that FRAME also works when some subset of the agents are playing stationary strategies. Specifically, agents using FRAME are able to achieve a best response against those agents playing stationary strategies.

3.1 Experimental Results

In this section we discuss our findings from a series of experiments.

3.1.1 Experimental Setup

While in theory any expert will work in FRAME, methods that make gradual adjustments to the strategies of agents are preferred. In our experiments we chose two such experts; Win or Learn Fast (WoLF) [2] and Logistic Fictitious Play (LFP) [4]. As a basis for comparison, we also used the *Naive Expert*, which always picks a strategy at random.

WoLF is a variable learning rate applied to a gradient-ascent learning approach. Each turn the strategy is moved towards a best response, however the strategy is moved more aggresively when the agent is doing worse than expected.

LFP is a form of learning where, at each iteration, the agent chooses a particular action with a probability that is in proportion to an exponential function of the utility that this action has yielded in the past.

We ran experiments on a wide range of games, including repeated Prisoner's Dilemma, Battle of the Sexes, 2-Player Matching Pennies and 3-Player Chicken. Due to space limitations we are unable to report our findings in these games in any detail, except to say that in self-play, agents using FRAME were able to quickly converge to Nash equilibria. We report, in detail, our findings from Shapley's game (Figure 1). Shapley's game is a classic but challenging one. In particular, WoLF does not converge in Shapley's game whereas LFP does.

For our experiments, LFP was run with $\lambda = 0.5$ and WoLF with $\delta_w = \frac{1}{100+t}$ and $\delta_l = 3\delta_w$. For FRAME, we let $p = 0.75$.

3.1.2 Results

A trial was said to have converged when the joint strategy was within three decimal places of any Nash equilibrium. Each of our experiments consisted of 1000 trials. We present our findings in a histogram format, which show the percentage of each experiment (grouped into 25 bins) that took a certain number of iterations to converge.

As shown in Figure 2, convergence in Shapley's Game is achieved using just a Naive Expert. However, by picking a better expert, we can do much better. Figure 2, shows the convergence when LFP is used as the expert and consulted 75% of the time. The convergence rate improves by three orders of magnitudes. We also conducted other experiments which showed that as LFP was consulted more and more often, the convergence continued to improve. On the other hand, Figure 2, shows the convergence rate when an expert poorly suited for Shapley's game, such as WoLF, is used as the expert, the convergence rate suffers but convergence is still achieved.

4. ADAPTIVE-FRAME

Despite the success of FRAME, it has one fundamental limitation. As our experiments showed, any specific expert is only useful for a limited set of games. Hence, once an agent picks its expert, it has limited the set of games for which it can achieve good convergence rates. Furthermore, even if an agent was allowed to pick a new expert for each game, it would not always be possible to know, before the game started, which expert was best to use.

To address this problem, we created a generalization of FRAME called adaptive-FRAME. Adaptive-FRAME allows an agent, at any point in a game, to choose from many possible experts. To help agents make the decision of which expert to actually consult, agents make use of an *experts algorithm*. An experts algorithm is any algorithm that, given a set of experts and their past performances, suggests which expert to consult. This allows adaptive-FRAME the flexibility to deal with new and unknown games.

Formally, the set of possible experts for agent i to consult is denoted by $E_i = \{e_{i,0}, \ldots, e_{i,|E_i|-1}\}$. The Naive Expert is always $e_{i,0}$. With slight abuse of notation, we define e_i to be some specific but undefined expert for agent i. At time t expert e_i is consulted with probability $p_i^t(e_i)$ and returns a suggested strategy β_{e_i}. Agent i's experts algorithm is denoted by æ$_i$ and p_i is called æ$_i$'s policy. We only require that for all t, $p^t_{e_{i,0}} > 0$ and $\sum_{t=0}^{\infty} p^t_{e_{i,0}} = \infty$; as long as this holds, the correctness for adaptive-FRAME follows directly from the proof of correctness for FRAME.

4.1 LERRM

To create a MAL experts algorithm, we first need a useful way of measuring performance of the experts. Since the goal of experts is to try and reduce an agent's regret, we created a metric, *Expected Regret Reduction* (ERR), defined as

$$ERR(e_i)_i^T = \frac{\sum_{t=0}^{T-1}(r(\beta^t)_i^t - r(\beta_{e_i}^{t+1}, \beta_{-i}^{t+1})_i^{t+1})}{T}.$$

ERR estimates expert e_i's ability to reduce an agent's regret over some time period $\{0, \ldots, T\}$ by assuming that all other agents' strategies are fixed but that e_i's suggested strategies were always followed. ERR then calculates the average reduction in regret e_i's strategies would have achieved.

Our experts algorithm, *Logistic Expected Regret Reduction Maximization* (LERRM), is based on the idea of LFP;

$$LERRM(e_i)_i^t = \frac{e^{\frac{1}{\lambda}ERR(e_i)_i^t}}{\sum_{e_i' \in E_i} e^{\frac{1}{\lambda}ERR(e_i')_i^t}}.$$

LERRM is designed as a general approach that can be used in other MAL settings.

4.2 Experimental Results

We tested adaptive-FRAME using Shapley's game. We tested three different experts algorithms. The Naive Experts Algorithm, which chooses each expert with equal probability, served as a benchmark by which to compare the others. Besides our experts algorithm, LERRM, we also used Hedge, a standard experts algorithm [3]. Hedge assigns "weights" to each expert and then consults an expert with a probability equal to that expert's weight proportional to all of the weights.

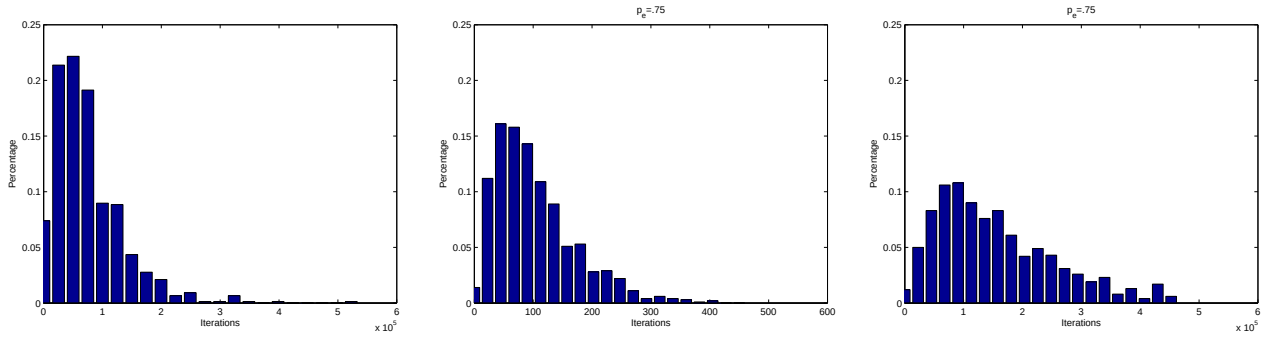


Figure 2: Convergence Rates for Shapley's Game using FRAME with the Naive Expert, LFP and WoLF, respectively. Note the difference in order of magnitude for the results for LFP.

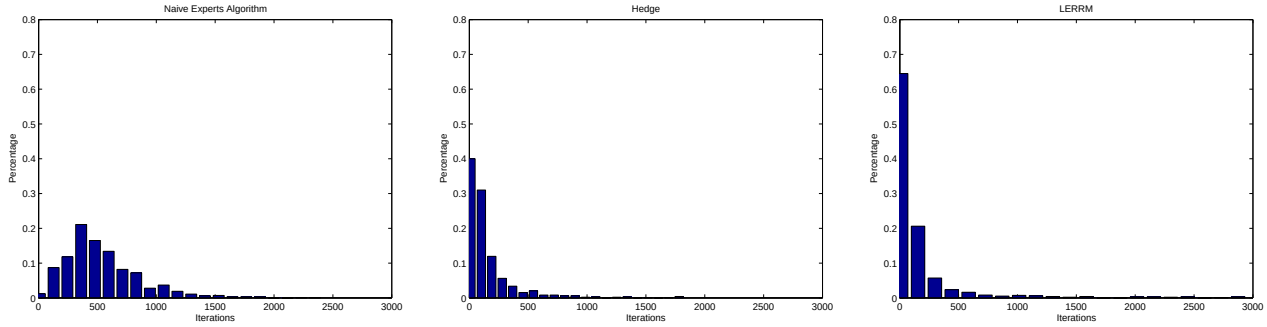


Figure 3: Convergence Rates for Shapley's Game using adaptive-FRAME with various experts algorithms.

4.2.1 Results

As seen by comparing Figure 3 to the results in Figure 2, all three of the experts algorithms do much better than the worst expert. Hedge and LERRM give, on average, much faster convergence rates compared to the Naive Experts Algorithm. In particular LERRM performs very well.

How are Hedge and LERRM able to achieve this performance? Since LFP is the best-suited expert for this game, Hedge and LERRM should consult LFP with high probability and WoLF with low probability. Our experimental results confirm this. At the point of convergence, Hedge was consulting LFP almost exclusively 20% of the time and LERRM consulted LFP almost exclusively 90% of the time. This difference helps explain why LERRM out performed Hedge.

5. CONCLUSION

In this paper we introduced two new multiagent learning algorithms, FRAME and adaptive-FRAME, and showed that, under certain assumptions, agents using either of these algorithms in self-play will converge to the set of Nash equilibria. The key idea of FRAME is that it will sometimes consult experts. If the expert is an effective learning procedure itself, then FRAME will also be effective. However, if the expert performs poorly, then FRAME's theoretical properties still hold, and in particular FRAME is still guaranteed to converge to a Nash equilibrium. The key idea of adaptive-FRAME is to allow agents the possibility of consulting different experts. Furthermore, agents can use experts algorithms to help them decide which expert to consult.

There are several research directions which we intend to pursue. First, there are several other experts, each specializing in their own class of games, that could be used [1]. By combining experts we might be able to create a powerful and highly effective general learning procedure.

6. REFERENCES

[1] B. Banerjee and J. Peng. $RV_{\sigma(t)}$: A unifying approach to performance and convergence in online multiagent learning. In *Proceedings of AAMAS-2006*, pages 2–7, Hakodate, Japan, 2006.

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

[3] Y. Freund and R. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

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

[5] F. Germano and G. Lugosi. Global Nash convergence of Foster and Young's regret testing. *Games and Economic Behavior*, 60(1):135–154, 2007.

A New Perspective to the Keepaway Soccer: The Takers (Short Paper)

Atil Iscen
Middle East Technical University
Ankara, Turkey
atil@ceng.metu.edu.tr

Umut Erogul
Middle East Technical University
Ankara, Turkey
umuero@ceng.metu.edu.tr

ABSTRACT

Keepaway is a sub-problem of RoboCup Soccer Simulator in which 'the keepers' try to maintain the possession of the ball, while 'the takers' try to steal the ball or force it out of bounds. By using Reinforcement Learning as a learning method, a lot of research has been done in this domain. In these works, there has been a remarkable success for the intelligent keepers part, however most of these keepers are trained and tested against simple hand-coded takers. We tried to address this part of the problem by using Sarsa(λ) as a Reinforcement Learning method with linear tile-coding as function approximation and used two different state spaces that we specially designed for the takers. As the results of the experiments confirm, we outperformed the hand-coded taker which results in creating a better trainer and tester for the keepers. Also when designing the new state space, we noticed that smaller state spaces can also be successful for this part of the problem.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent systems*

General Terms

Experimentation, Performance

Keywords

RoboCup, Keepaway Soccer, Takers

1. INTRODUCTION

Keepaway is a subproblem of the RoboCup Soccer Simulator (RCSS) in which one team, 'the keepers' tries to maintain possession of the ball within a limited region, while the opposing team, 'the takers' tries to gain possession of the ball[2]. This game is commonly preferred in Machine Learning researches [4][5][7], because it can be a good testbed with less agents resulting in a less complex problem rather than two teams with 11 agents playing full team soccer each having different roles. Because of this, there are many researches focusing on the keepaway game, but most of these researches apply machine learning methods to maximize the possession of keeper agents.

In these researches the experiments are made with basic takers, which run towards the ball without considering to cooperate, which does not create a big challenge for the keepers. When developing learning keepers, this type of takers does not really test the potential of the adversary agents. In our project we plan to address this part of the problem by developing learning takers. Another interesting point of developing takers is that in learning keepers problem only the keeper with the ball decides an action whereas in learning takers all the agents have to decide an action in each step. This makes the agents more dependent on each others decision, making the game more suitable for cooperation.

Among learning methods we used Reinforcement Learing[3] which is one of the most preferred learning method in RCSS, because it is well suited to meeting its challenges, like sequential decision making, achieving delayed goals and handling noise. As a Reinforcement Learning algorithm we have chosen Sarsa(λ) learning with tile coding because of its previous success in application of keepers in one of the best known paper in this area.[2]

Cite as: A New Perspective to the Keepaway Soccer: The Takers (Short Paper), A. Iscen and U. Erogul, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal,pp. 1341-1344.

2. ALGORITHM

2.1 Problem Definition

In keepaway a team of m players faces the task of keeping possession of the ball within a rectangle region of play, resisting attempts of the opposing team of n takers to wrest possession. For research purposes, this problem is embedded into RCSS with the keepaway framework developed by Stone et Al [1]. This framework contains many classes and methods and some high level functions like passing and marking. In addition, although this framework has the main functions for the learning process, we had to modify some of them to make the framework suitable for the takers.

We accepted the task as episodic, each starting with one of the keepers having possession, and finishing when any of the takers gets the ball or the ball goes out of bounds. Apart from the keepers, which decide to an action only when the agent has the ball until its decision to pass, the takers need to decide to an action in each cycle, which can prevent them from reaching any of the keepers if their decided action changes repeatedly. As this makes it impossible to see the effects of their immediate decisions, we decided to make the takers do the selected action n consequent cycles. With

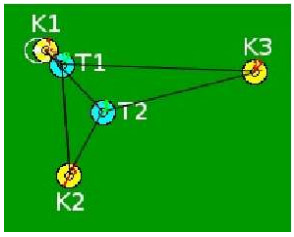


Figure 1: Keepaway scene and labels

using this 'n step same action' trick, we made the learning process easier at start, but this causes a disadvantage when further agility or sensitivity to the states of disregarded cycles is required. We decided the value of n as 15 cycles, which is the duration of a successful pass execution.

2.2 States and Actions

Since we are dealing with the takers, the only possible actions are GotoBall and Mark(n) which means marking the nth keeper, where keepers are ordered by their distance to the ball. When deciding on states, we wanted to minimize the number of states by trying to use the information that would be sufficient. First, to have a dynamic labeling, we sort the keepers according to their distance to the ball. K1 means the keeper with the ball. Then the takers are sorted such that the first taker will be itself. The others are sorted according to their distance to this taker(Fig. 1). For our first taker model (atum) the state variables are constructed as the distances of each taker to each keeper (T1-K1, T1-K2, T1-K3, ..., T2-K1, T2-K2, ...). For increasing number of players, the size of the state space becomes a problem. To overcome this, we minimized the state space by constructing a second taker model. For this model, only the distances for the current taker are taken into consideration, for the other takers only the label of the nearest keeper is considered. This gives for m keepers and n takers $m + (n-1)$ variables, whereas first model has $m * n$ state variables.

2.3 Learning Algorithm

For the learning algorithm, because of its success in learning keepers [2], we have chosen sarsa(λ) which is a commonly used algorithm in RL [3]. For feedback, the rewards are zero until the end of the episode and it becomes 1 when the takers force keepers to end episode. Eventhough we have less state variables, the state space is still too large. To decrease the size, and to generalize the states we used function approximation. For function approximation we used tile coding, which is a linear function approximation scheme that partitions the input space into axis aligned regions called tiles.

3. EXPERIMENTAL RESULTS

3.1 Methodology

For the keepaway problem the common evaluation method is the average episode length of the game. Our aim is to decrease these durations, especially the ones presented in P.Stone's research[2], since we also use the keepers they developed.

After several tries, we chose the learning parameters giving best learning curves. Although we have infinitely many

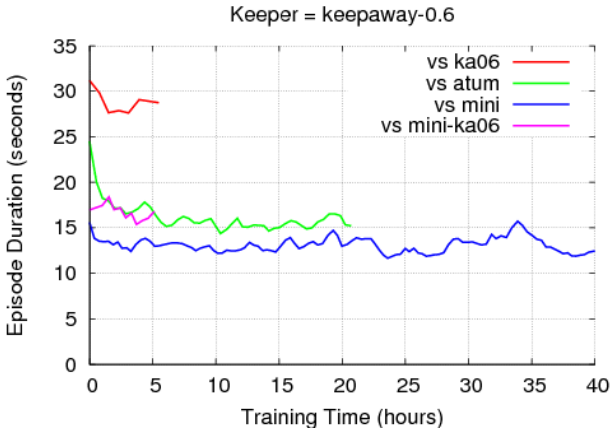


Figure 2: Keeper = ka06

choices, we decided on $\alpha = 0.125, \lambda = 0.5, \epsilon = 0.05, \gamma = 0.8$($\alpha beingLearningRate, \gamma beingDiscountFactor$).

All of the testing and implementation has been done on 32 bit 2.6.22-14 linux kernel, with rcssbase-10.0.11, rcssserver 10.0.7 and keepaway-0.6 on a 2.20Ghz AMD Athlon PC. The results are converted to graphics using the points constructed by a sliding window containing 300 episodes. All of the experiments are conducted in sync mode(server advances cycle immediately when all clients have responded which allows games to be much faster) with unrestricted vision settings(360 degree view of field).

The first keeper that we used to test is the original hand-coded version that we got from the keepaway framework and is denoted as ka06. For further testing we used the learning keepers provided by M.Taylor et Al[1] which will be denoted as mt07. For mt07 we used weights learned previously, which were saved after a learning process having one of the highest possession durations among learning keepers. For the takers part, we have only one previous taker to compare our work with, which is ka06. Our first taker model is denoted as atum, and the second is denoted as mini in the graphics and the results. The extended versions of the takers like mini-ka06 express takers (in this example mini) which use the previously saved weights against the keepers written after the '-' symbol (ka06 in this example). For atum-l and mini-l, the extension 'l' signifies that they load weights saved during a learning session against learning agents. For durations of experiments we used long sessions for learning agents, and shorter ones for agents loading previously learned weights.

3.2 Results

At first, we compared various takers performances against hand-coded keepers of the keepaway framework[1]. As seen in the Table 1, the hand-coded keepers versus hand-coded takers get an average result of 29.2 seconds, whereas our first taker atum developed to decrease this duration became successfull by getting 16.3 as average. For the third model (mini), although the number of state variables is reduced, it shortens the durations further to 12.9. These statistics clearly show that our expectations in the success of learning takers come true.

For the second test, when we compare atum and mini, there is two important points. First, during the first 5 hours

Table 1: Episode durations against ka06

Takers	Average	Min	Max
ka06	29.2	27.6	31.1
atum	16.3	14.3	24.4
mini	12.9	10.8	15.5

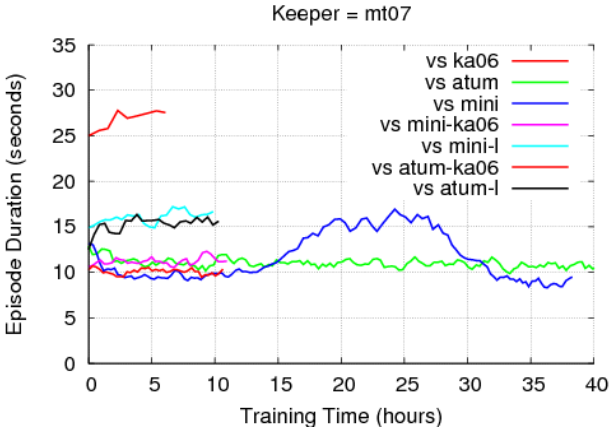


Figure 3: Keeper = mt07

Table 2: Episode durations against mt07

Takers	Average	Min	Max
ka06	26.7	24.9	27.7
atum	10.9	9.9	12.8
mini	11.1	8.3	16.9
mini-ka06	11.1	10.5	12.2
mini-l	15.9	14.8	17.2
atum-ka06	10.1	9.4	10.6
atum-l	15.1	12.4	16.0

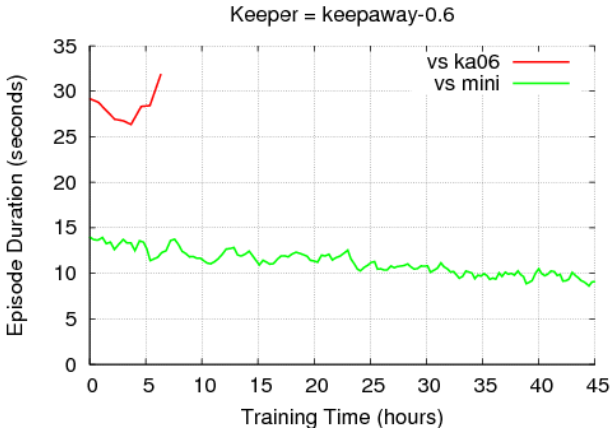


Figure 4: Keeper = ka06 in 4 vs 3

of training(Fig. 3), mini converges more quickly. Secondly, mini has a lower minimum (Table. 2) but its seems more unstable than atum. In our opinion, this is caused by having less state variables not being able to represent the state clearly.

Interestingly there is a big difference between atum,mini and atum-l,mini-l respectively. We believe that the reason of this is the atum-l and mini-l are trained against the keepers which are at the start of the learning process.

Another interesting point for this test is, the takers trained against the hand coded takers are as successful as the takers trained specifically against mt07. This means that the opposing keepers ka06 and mt07 behave similarly, as expected from the statistics given by Stone et Al.[2]

For the last test (Fig. 4), we see that the mini especially developed for keepaway with more agents is successfull at his primary mission with a clear improvement over the hand-coded takers.

4. CONCLUSIONS AND FUTURE WORK

Looking at the results, we can clearly say that we achieved our initial goal which is to develop a successful learning taker which performs better than the hand-coded ones. After testing against various keepers, we have shown that our algorithm is robust to different types of keepers. We also concluded that previous studies on learning keepers can be also applied to the takers with the help of an addition like 'n-step same action'. As a further research, n could be decreased during learning when the agent needs more agility.

Also for the takers part of the keepaway learning problem, using a second model of taker, we saw that we can achieve similar (sometimes better) results in a less stable way. With this new model using less state variables, the same learning process can be used with more than 5 agents with a manageable state space. As a new application area, RoboCup-breakaway can be used to test the general success of this algorithm for the defense team.[6]

In addition to these ones, one of our main contributions is providing a better challenge and a benchmark to the researchers of the keepaway framework. We think that using a better taker for the experiments will help the researchers analyze the true potential of the keepers

5. REFERENCES

[1] P. Stone, G. Kuhlmann, M. E. Taylor, and Y. Liu. Keepaway soccer: From machine learning testbed to benchmark. In I. Noda, A. Jacoff, A. Bredenfeld, and Y. Takahashi, editors, *RoboCup-2005: Robot Soccer World Cup IX*, volume 4020, pages 93–105. Springer Verlag, Berlin, 2006.

[2] P. Stone, R. S. Sutton, and G. Kuhlmann. Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

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

[4] M. E. Taylor, S. Whiteson, and P. Stone. Temporal difference and policy search methods for reinforcement learning: An empirical comparison. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence*, pages 1675–1678, July 2007. (Nectar Track).

[5] M. E. Taylor, S. Whiteson, and P. Stone. Transfer via inter-task mappings in policy search reinforcement learning. In *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–8, New York, NY, USA, 2007. ACM.

[6] L. Torrey, T. Walker, J. W. Shavlik, and R. Maclin. Using advice to transfer knowledge acquired in one reinforcement learning task to another. In *ECML*, pages 412–424, 2005.

[7] S. Whiteson, N. Kohl, R. Miikkulainen, and P. Stone. Evolving keepaway soccer players through task decomposition. *Machine Learning*, 59(1):5–30, May 2005.

On the Usefulness of Opponent Modeling: the Kuhn Poker case study

(Short Paper)

Alessandro Lazaric
Politecnico di Milano
Dept. of Elect. and Inf.
Piazza Leonardo da Vinci, 32
Milan, Italy
lazaric@elet.polimi.it

Mario Quaresimale
Politecnico di Milano
Dept. of Elect. and Inf.
Piazza Leonardo da Vinci, 32
Milan, Italy
mario.quaresimale@mail.polimi.it

Marcello Restelli
Politecnico di Milano
Dept. of Elect. and Inf.
Piazza Leonardo da Vinci, 32
Milan, Italy
restelli@elet.polimi.it

ABSTRACT

The application of reinforcement learning algorithms to Partially Observable Stochastic Games (POSG) is challenging since each agent does not have access to the whole state information and, in case of concurrent learners, the environment has non-stationary dynamics. These problems could be partially overcome if the policies followed by the other agents were known, and, for this reason, many approaches try to estimate them through the so-called opponent modeling techniques. Although many researches have been devoted to the study of the accuracy of the estimation of opponents' policies, still little attention has been deserved to understand in which situations these model estimations can be actually useful to improve the agent's performance.

This paper presents a preliminary study about the impact of using opponent modeling techniques to learn the solution of a POSG. Our main purpose is to provide a measure of the gain in performance that can be obtained by exploiting information about the policy of other agents, and how this gain is affected by the accuracy of the estimated models. Our analysis focus on a small two-agent POSG: the Kuhn Poker, a simplified version of classical poker. Three cases will be considered according to the agent knowledge about the opponent's policy: no knowledge, perfect knowledge, and imperfect knowledge. The aim is to identify which is the maximum error that can affect the model estimate without leading to a performance lower than that reachable without using opponent-modeling information. Finally, we will show how the results of this analysis can be used to improve the performance of a reinforcement-learning algorithm coped with a simple opponent modeling technique.

Categories and Subject Descriptors

I.2 [**Artificial Intelligence**]: Learning

General Terms

Algorithms

Cite as: On the Usefulness of Opponent Modeling: the Kuhn Poker case study (Short Paper), Alessandro Lazaric, Mario Quaresimale and Marcello Restelli, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp. 1345-1348.

Keywords

Multi-agent Learning, Opponent Modeling, Reinforcement Learning

1. INTRODUCTION

In this paper, we propose a preliminary study about the effectiveness of using Opponent Modeling (OM) techniques to improve the performances of reinforcement-learning algorithms in Partially Observable Stochastic Games (POSG).

To this day, the main studies in this field concern the development of OM algorithms [6] devoted to improve the accuracy of opponent behavior estimation. OM approaches can be classified according to the amount of prior knowledge required for their application. Statistical classifiers, artificial neural networks, deterministic finite automata, and decision trees are examples of general-purpose methods, while expert systems, feature-based methods, and plan recognition belong to the set of domain-specific techniques. Regardless of which technique is considered, we want to point out that, when the approximation error of the estimated model is too large, using this information could prove detrimental for the learning process.

In order to avoid this eventuality, McCracken and Bowling studied OM from a different point of view, aiming to ensure efficacy from its usage. Their research is founded on admitting success of OM, but also focuses on the fact that such success depends on situations: exploiting a wrong or ineffective opponent model may drastically reduce performances. They introduced the Safe Policy Selection algorithm to profitably exploit OM [5], defining as safe a policy that leads to a total reward not lower than the expected value of the optimal policy; in this way, when OM yields to decrease such safety value, it is not used. McCracken and Bowling apply the cited above algorithm to Rock-Paper-Scissors, a zero sum matrix-game, where information about the state space is complete, while we aim to study OM effectiveness in a POSG context.

Going in the same direction of such evaluation, we provide a preliminary analysis on the performances achievable by exploiting OM techniques, in order to numerically quantify them both in worst and best cases. In particular, we show how the knowledge of the policies followed by other agents can be effectively used by the player to improve her performance. On the other hand, when the opponent's policy is not exactly known, but the player can exploit only an

estimated model based on the previously observed actions, the advantage can be significantly reduced, or it can even turn into a loss of performance. On the basis of this analysis, we experimentally show that it is possible to improve the performance of an RL agent by avoiding to exploit OM information when the accuracy of the estimated model is too low.

Recently, many research works have focused on Texas Hold'em Poker [1] [2], considered as the ideal testbed for studying POSG. Nevertheless, as Texas Hold'em is too complex for a preliminary analysis, we focus our attention on studying OM techniques in a simplified version of Poker Game: the Kuhn Poker [4]. Although this problem is quite trivial, it still has the key features of the primal game, and for this reason it was already studied, with other purposes, in past works [6] [3].

The rest of the paper is structured as follows: next section briefly describes Kuhn Poker's rules and its formalization as a POSG. In Section 3 we expose our OM analysis, which is so structured: at first, we study the case where no information about the opponent's policy is considered, then we analyze the improvement that can be obtained when the policy followed by the opponent is known, and finally we show how the use of an approximate model of the opponent's policy may have negative effects. In Section 4 we experimentally compare the performance of three RL agents: without OM, with OM, and using OM only when the model estimation is accurate enough. In the last section we draw conclusions and describe future directions.

2. KUHN POKER

Kuhn Poker is a simplified two-person poker, its rules are as follows:

- Two player, each of whom has two dollars
- 3 card deck: King (K), Queen(Q), Jack (J)
- At start, both players ante one dollar and receive a private card; the third card remains hidden to each of them.
- After anting, players can choose between two actions: BET and PASS.

After both players anting, the non-button chooses whether to BET or to PASS; then the button replies with her chosen action. A hand terminates when both players choose BET or the second action of betting sequence is PASS. The most long betting sequence is when the non-button chooses PASS and the button replies with BET: only in this case, non-button must act again, then the hand is terminated. A player wins a hand when her opponent folds, or when she has the highest card in the showdown. The game goes to showdown when both players bet or pass. If only one player bets and the other replies with a PASS, showdown does not occur. Given this betting sequence, the highest pot is 4 dollars, so the best gain an agent can obtain is 2 dollars; this occurs when both players bet. If showdown is reached by a two pass sequence, the pot is 2 dollars and the gain for the winning agent is 1 dollar.

Although the Kuhn Poker is a Partially Observable Stochastic Game (POSG), in this paper we limit our analysis to the case of fixed opponents, so that the problem can be modeled as a POMDP, that is described as the tuple: $< \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, \mathcal{O} >$, where $\mathcal{S}$ is the state space describing the environment, $\mathcal{A}$ is the set of actions that can be performed in the environment, $\mathcal{T}$ is the transition function, expressing the probability to go from a starting state to a next state when a given action is executed, and $\mathcal{R}$ is the reward function, measuring the goodness of taking an action in a certain state. Ω is the set of observations that the agent can make; $\mathcal{O} : \mathcal{S} \times \mathcal{A} \times \Omega \longrightarrow$ [0,1] is the observation function, where $\mathcal{O}(s', a, o) = P(\Omega_t = o | S_t = s', A_{t-1} = a)$ is the probability of experiencing observation o, given the performed action a and being s' the ending state. The behavior of each player is specified by her policy π, which is a function that, given a state s and an action a, returns the probability to execute a in s: $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$.

3. OPPONENT MODELING ANALYSIS

In this section, we analyze the effectiveness of exploiting information about the opponent's policy in the Kuhn Poker game. Our analysis is carried out by considering that the opponent is following a stationary policy, so that the problem can be formalized as a POMDP, where the opponent's actions can be used as observations of the hidden part of the state space. In particular, we consider opponent's policies that depend only on her private card, and, fixed one policy, we compute the utility value for the best-response policy.

Without any information about the opponent's policy, each player knows only her own private card. This means that, if a player owns a Queen, the probabilities that her opponent owns a Jack or a King are both equal to 0.5. On the other hand, by knowing the opponent's policy and observing her actions, a player can exploit this information to reduce her uncertainty about the private card of the opponent. To measure the amount of information that can be obtained about the opponent's private card by knowing her policy, we use the *mutual information* quantity between the random variable A, which represents the opponent's action, and the random variable C, which represents the opponent's private card:

$$\mathcal{I}(A; C) = \mathcal{H}(C) - \mathcal{H}(C|A), \tag{1}$$

where $\mathcal{H}$ is the entropy function, that measures the uncertainty about a stochastic variable. Since the private card of the opponent is always randomly extracted, the entropy $\mathcal{H}(C)$ is constant, and attains its maximum value. On the other hand, the conditional entropy of variable C given the value of variable A is strictly dependent from the policy π_{opp} followed by the opponent. Given the assumption that the opponent's policy depends only on the value of her private card, $\mathcal{H}(C|A)$ is formally defined as:

$$\begin{aligned} \mathcal{H}(C|A) &= -\sum_{a \in A} Pr(a) \sum_{c \in C} Pr(c|a) log(Pr(c|a)) \\ &= -\sum_{a \in A} \left(\sum_{c \in C} \pi_{opp}(c, a) \cdot Pr(c) \right) \cdot \\ &\quad \cdot \sum_{c \in C} \frac{\pi_{opp}(c, a)}{Pr(a)} log \left(\frac{\pi_{opp}(c, a)}{Pr(a)} \right). \end{aligned} \tag{2}$$

Low values of the conditional entropy $\mathcal{H}(C|A)$ (and, consequently, high values of the mutual information $\mathcal{I}(A; C)$) mean that, by knowing the opponent model and observing her actions, we can significantly reduce the uncertainty

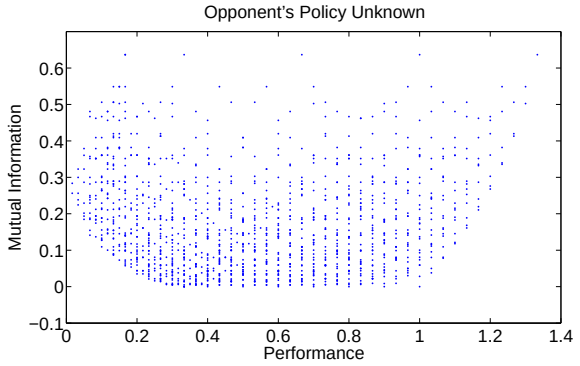


Figure 1: Mutual information and best-response performance for 1,000 fixed opponent's policies. No information about the policies is used.

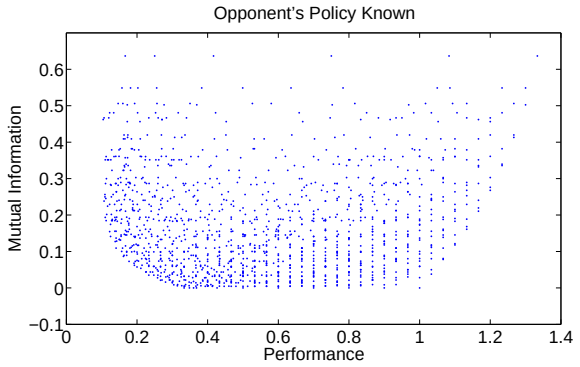


Figure 2: Mutual information and best-response performance for 1,000 fixed opponent's policies. Policies are exactly known.

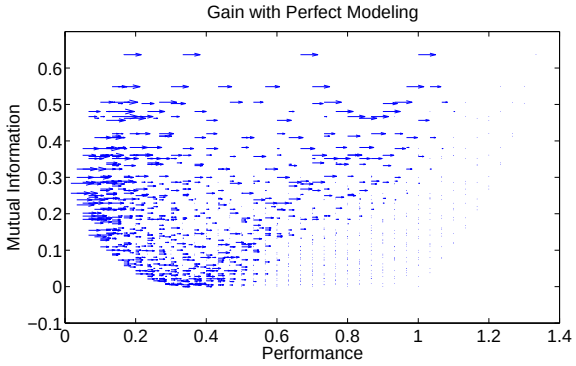


Figure 3: The arrows show the improvement due to the knowledge of the opponent's policy.

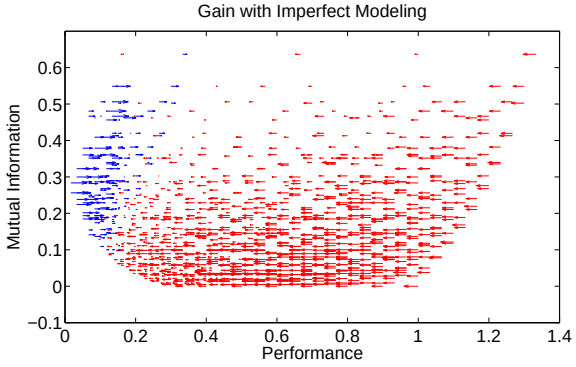


Figure 4: The arrows show the improvement (blue) or the worsening (red) when the player exploits an estimated model whose distance from the actual opponent's policy is up to 0.1.

about the opponent's card. On the other hand, when the opponent follows a policy that does not depend on the value of her own card (e.g., a random policy), the conditional entropy $\mathcal{H}(C|A)$ is equal to the entropy $H(C)$, so that the mutual information is zero; in these cases, the use of OM techniques is useless.

To study the effect of OM techniques in the Kuhn Poker, we consider several possible stationary policies for the opponent. For each one of these policies, we compute the corresponding mutual information $\mathcal{I}(A;C)$ and the performance attained by its best-response policy. In general, given an opponent policy π_{opp}, the expected performance of a policy π is the average of the expected values of the states weighted by the probability of visiting the corresponding state:

$$U(\pi|\pi_{opp}) = \sum_{s \in \mathcal{S}} Pr(s|\pi, \pi_{opp}) V(s|\pi, \pi_{opp}).$$

The best-response policy π^* against a given policy π_{opp} is the one which attains the highest utility:

$$\pi^*_{\pi_{opp}} = arg \max_{\pi \in \Pi} U(\pi|\pi_{opp}).$$

Figure 1 shows relation between the mutual information of the opponent's policy (y-axis) and the performance of its best-response policy (x-axis), in the case where no information about the opponent's policy is exploited [1]. As we can see from the graph, the opponent's policy which is less exploited has a quite high mutual information value. In particular, it is worth noting that there is no policy for the opponent that is placed near the origin of the graph. This means that, if the opponent wants to adopt a policy that can be hardly exploited, it has to follow a policy that reveals information about her private card. On the other hand, when the opponent wants to hide at most the value of her card, she can be easily exploited. This trade-off is what makes the use of OM techniques interesting.

Figure 2 shows how the situation changes when the player knows the policy followed by the opponent, so that the problem can be formalized as a POMDP and solved by using the observable histories as state information. As it can be noticed, several points have been moved to the right, since exploiting the information of the opponent's policy has allowed to identify best-response policy with higher performances. To give a better visualization of the effect of knowing the opponent's policy, in Figure 3 we have used arrows to represent the gain. As expected, when the mutual information is low, knowing the opponent's policy results in small gains.

[1]Each point corresponds to a different opponent's policy. The 1,000 policies have been generated by considering ten evenly-spaced values for the probability of betting given each value of the private card.

On the other hand, it is not always true that the knowledge of policies that convey much information can lead to high gains, especially when the policies are quite weak (look at the right side of the graph).

Unfortunately, in adversarial problems, a player does not know the policy of her opponent. For this reason, the resort to OM techniques is quite common. The problem is that the estimated model is an approximation of the policy actually followed by the opponent. Using a model affected by a large approximation error could lead to a performance that is worse than that achievable using no model at all. Figure 4 shows how much, in the worst case, the performances change when the distance between the actual opponent's policy and the estimated one is not larger than 0.1[2]. The arrows that point toward left (red arrows) corresponds to opponent's policies for which the agent may have a loss of performance when using a model with a low accuracy. As we can notice, the opponent's policies that are associated to larger losses are those that have little or not advantage when knowing the actual policy followed by the opponent.

In the next section, we show how this analysis can be used to improve the performance of an RL player.

4. LEARNING EXPERIMENTS

In this section, we show some preliminary experiments obtained by using Q-learning [7], a popular reinforcement-learning algorithm, against a fixed opponent. In particular, we consider three different versions of Q-learning:

- Q-learning without OM: the state space depends only by the player's private card;
- Q-learning with OM: the state space depends by the player's private card and by the observed opponent's action[3]
- Q-learning with reliable OM: in this version, we keep an estimate of the accuracy of the opponent's model, so that when the accuracy is below a certain threshold we use Q-learning without OM, otherwise we exploit the opponent modeling information. The choice of the threshold is made according to the analysis described in the previous section.

In Figure 5, the performances of the three learning algorithms are represented. As it can be noticed, Q-learning with OM is ineffective in the first learning steps when the information about the opponent's policy is still highly uncertain. On the other hand, Q-learning without OM is able to quickly learn a good solution, but it has not enough information to exploit the opponent at best. As we can notice, the third approach, which uses the opponent-modeling information only when it is accurate enough, is able to attain both a good learning speed and a good performance in the long-run.

5. CONCLUSIONS

[2]The distance between two policies is computed as the L2-norm of the difference vector between the two vectors that specify the policies in the two models.

[3]In this problem, this information is equal to the history of observations, thus allowing Q-learning to solve the POMDP problem.

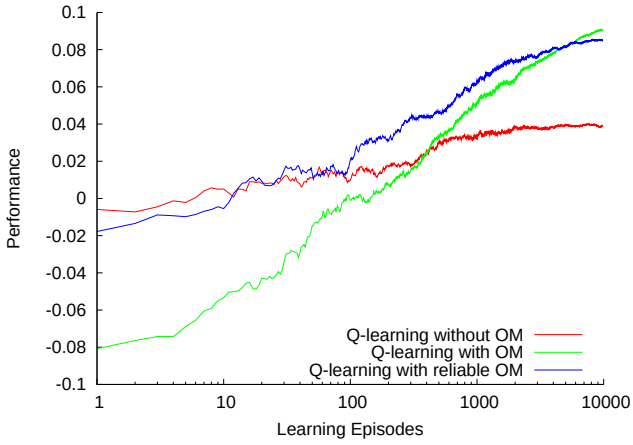


Figure 5: Comparison of three RL algorithms against a fixed opponent's policy. Results are averaged over 1,000 runs

In this paper we have presented a preliminary study on measuring the usefulness of using opponent-modeling techniques in Partially Observable Stochastic Games, by focusing on a simple poker game. The results of our analysis show that, in a context like Kuhn Poker, OM technique can be very useful, but only under the necessary condition that the model describing the opponent's behavior is accurately estimated.

This paper represents just a first step and opens several directions for future research. The following steps will be devoted to extend this analysis to cases where the opponent can adopt more complex policies, such as stationary policies that consider the actions performed by the player, non-stationary policies, and non-stationary policies based on OM information. The final goal of this work is to extend the results of these analyses to more complex problems, such as the Texas Hold'em Poker.

6. REFERENCES

[1] D. Billings, A. Davidson, J. Schaeffer, and D. Szafron. The challenge of poker. *Artificial Intelligence*, 134(1-2):201–240, 2002.

[2] D. Billings, D. Papp, J. Schaeffer, and D. Szafron. Opponent modeling in Poker. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 493–498, Madison, WI, 1998. AAAI Press.

[3] D. Koller and A. Pfeffer. Representations and solutions for game-theoretic problems. *Artificial Intelligence*, 94(1-2):167–215, 1999.

[4] H. Kuhn. A simplified two person poker. In W. H. Kuhn and A. W. Tucker, editors, *Contributions to theory of games*, pages 97–103. Princeton University Press, 1950.

[5] P. McCracken and M. Bowling. Safe strategies for agent modelling in games. In *AAAI 2004 Symposium on Artificial Multi-Agent Learning*. AAAI Press, 2004.

[6] T. Schauenberg. Opponent Modeling and Search in poker. Master's thesis, University of Alberta, 2006.

[7] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. Bradford Book, 1998.

Graph Laplacian Based Transfer Learning in Reinforcement Learning (Short Paper)

Yi-Ting Tsao
Department of Computer Science
National Tsing-Hua University
HsinChu, Taiwan
yiting.tsao@gmail.com

Ke-Ting Xiao
Department of Computer Science
National Tsing-Hua University
HsinChu, Taiwan
peter.xiau@gmail.com

Von-Wun Soo
Department of Computer Science
National Tsing-Hua University
HsinChu, Taiwan
soo@cs.nthu.edu.tw

ABSTRACT
The aim of transfer learning is to accelerate learning in related domains. In reinforcement learning, many different features such as a value function and a policy can be transferred from a source domain to a related target domain. Many researches focused on transfer using hand-coded translation functions that are designed by the experts a priori. However, it is not only very costly but also problem dependent. We propose to apply the Graph Laplacian that is based on the spectral graph theory to decompose the value functions of both a source domain and a target domain into a sum of the basis functions respectively. The transfer learning can be carried out by transferring weights on the basis functions of a source domain to a target domain. We investigate two types of domain transfer, scaling and topological. The results demonstrated that the transferred policy is a better prior policy to reduce the learning time.

Categories and Subject Descriptors
I.2.6 [**Artificial Intelligence**]: Learning – *knowledge acquisition, parameter learning.*

General Terms
Experimentation, Theory.

Keywords
reinforcement learning, transfer learning, graph Laplacian

1. INTRODUCTION
One of the disadvantages in reinforcement learning (RL) [1] is that two different domains with different initial states and goal states must be learned separately to acquire an optimal policy for each domain. It would waste time to simply learn twice in two different domains even if they might share some similar subtasks. Transfer learning is an approach to improve the performance of cross domains by avoiding redundant learning.

In a reinforcement learning problem, the value function provides a guideline for action selection in a given state that is known as a policy. Many transfer methods that transfer different features from a source domain to a target domain have been proposed [2, 3, 4]. One work is a rule transfer method that acquires some rules that approximate the policy in a source domain and translates into ones that can be used as a policy for a target domain [2]. Thus an agent may apply the translated policy that is acquired by hand-coded translation functions and revise a partial policy in a target domain. However, designing general translation functions becomes a problem. Another work based on case-based reasoning uses a similar idea but it acquires rules using a decision-tree method [3]. The other work is to transfer the policy from a source domain to a target domain directly [4] but it also requires hand-coded translation functions. Proto-value functions derived from spectral graph theory, harmonic analysis, and Riemannian manifold can be used to represent a set of the basis functions to approximate a value function [5, 6, 7]. A novel transfer method has been proposed to reuse a set of the basis functions from a source domain and just to learn the weights of the set of the basis functions to compose a value function for a target domain. This method can transfer domain features without hand-coded translation functions but it needs some exploring trials for a target domain to acquire the combination weights.

The aim of the transfer learning is to use the knowledge learned from a source domain to accelerate learning in a related target domain. In this paper, we propose a transfer method to obtain a better prior policy from a source domain to reduce the learning time in a similar target domain without hand-coded translation functions by spectral graph theory.

2. BACKGROUND
Most reinforcement learning researches are based on Markov Decision Processes (MDP) and a value function to guide an agent's actions in solving a domain. However, a value function can be too rigid to apply to a domain such that to transfer it directly to another domain is hard. Finding a set of suitable basis functions to express the value function helps the transfer. In this paper, the development is based on a discrete MDP and the spectral graph theory.

2.1 Markov Decision Process
A discrete Markov Decision Process M which is defined by a 4-tuple $(S, A, P^a_{ss'}, R^a_{ss'})$ where S is a finite set of states, A is a finite

Cite as: Graph Laplacian Based Transfer Learning in Reinforcement Learning (Short Paper), Yi-Ting Tsao, Ke-Ting Xiao, Von-Wun Soo, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp. 1349-1352.

set of actions, $P^a_{ss'}$ and $R^a_{ss'}$ represent the probability and reward of transiting to state *s'* when taking action *a* on state *s* respectively [1]. A function which determines the action that an agent should take at any state that the agent could reach is called a policy π. A policy is a mapping from a state to a unique action. The value function V^π represents the value by using policy function π and the optimal policy π^* is defined as a unique optimal value function V^* that can maximize the expected reward starting at a given state *s* with discount factor γ. The optimal value function $V^*(s)$ is defined as follows.

$$V^*(s) = \max_a \sum_{s'} P^a_{ss'}(R^a_{ss'} + \gamma V^*(s'))$$

The value function is represented in tabular form with one output for each input tuple. However, the state space in the real world is often so huge that to memorize the value table is impossible. We can approximate the value function in terms of a linear combination of a set of the basis functions as:

$$V^{\pi} = \alpha_1 V_1^B + \ldots + \alpha_n V_n^B$$

where each V_i^B is a basis function. Approximating by the basis functions saves a lot of memory. However, different sets of the basis functions may affect the function approximation. Therefore, for an agent to have good performance, selecting good basis functions to make good value approximation plays an important role.

2.2 Spectral Graph Theory

A Fourier analysis is to decompose a function in terms of a sum of trigonometric functions with different frequencies that can be combined together to represent the original function. Each frequency of trigonometric functions is inversely proportional to its importance in representing characteristics of the function. Therefore, if two functions are similar, their trigonometric functions tend to be similar at low frequencies and differ at high frequencies.

A graph Laplacian can be defined as the combinatorial Laplacian or the normalized Laplacian [8]. The combinatorial Laplacian *L* of the undirected unweighted graph *G* is defined as $L = D - A$ where *A* is the adjacency matrix and *D* is a diagonal matrix whose entries are the row sums of *A*. In problem solving, the states are represented as the vertices and the edges represent the connection (undirected) or transitions (directed) between the states so that one state can reach another. Let *u* and *v* represent two states in a graph and d_v represents the degree of *v*, a graph Laplacian $L(u, v)$ is defined as follows:

$$L(u,v) = \begin{cases} d_v & \text{if } u = v \\ -1 & \text{if } u \text{ and } v \text{ are adjacent} \\ 0 & \text{otherwise} \end{cases}$$

Let *f* denote a function mapping each vertex *u* of the graph into a real number. The combinatorial Laplacian *L* acts on a function *f* as

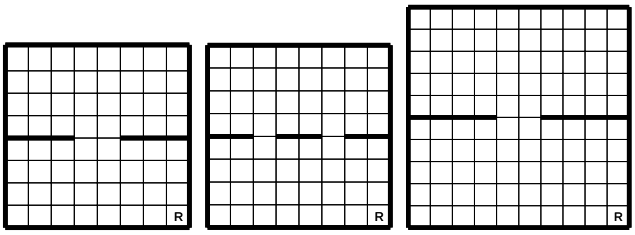
$$Lf(u) = \sum_{u \sim v} (f(u) - f(v))$$

where *u* and *v* are adjacent vertices. Functions that solve $Lf = 0$ are called harmonic functions [9]. It turns out that to find the harmonic functions is equivalent to finding the eigenvectors (or eigenfunctions) of $Lf = \lambda f$, where *f* is the eigenfunction and λ is the associated eigenvalue. A smaller eigenvalue implies a smoother eigenfunction. Furthermore, we can extend the idea in general with normalized graph Laplacian[8]. In our cases, the normalized graph Laplacian has the better consequences than the combinatorial Laplacian.

The spectral analysis of the graph Laplacian operator provides an orthonormal set of the basis functions that can approximate any square-integrable functions on a graph [8]. These basis functions which are called as proto-value functions in [5, 6, 7] construct a global smooth approximation of a function on the graph. In other words, the function can be decomposed into a sum of the basis functions [10]. Besides, the notion of the spectral analysis on graph Laplacian is similar to the Fourier analysis. The basis functions of a graph Laplacian corresponding to the smaller eigenvalues represent more valuable features and are thus more important. It implies that if two graphs are similar, their features tend to be similar at low-order basis functions and different at high-order basis functions.

3. THE TRANSFER METHOD

In [6], the authors distinguished three transfer types: task transfer, topological domain transfer, and scaling domain transfer as shown in Figure 1. The domain transfer problem means only the topology of the state space changes and rewards do not change. In this paper, we focus on both topological and scaling domain transfer.



(a) 8 x 8 grids source (b) 8 x 8 grids topological target (c) 10 x 10 grids scaling target

Figure 1. The example of topological and scaling domain transfer.

The transfer algorithm is described in Figure 2. The first step is to collect the topological knowledge of both domains retrieving basis functions respectively. The second step is to compute the corresponding basis functions of the graph Laplacian. The third step is to compute the coefficients of the basis functions approximating the real value function in the source domain. The fourth step is to approximate the target value function in terms of the target basis functions and the weights that are obtained from the source domain. The last step is to acquire the target policy through the approximated target value function.

The reason why the transfer algorithm works is that the basis functions of both domains with the same order play the same important role for the value functions at each domain respectively. Therefore, we transfer the obtained weights from a source domain to a target domain. If two domains are similar, the basis functions tend to be similar. It does not imply similar numeric value but

similar structure as shown in Figure 3. On the one hand, a small change of the domain cannot affect the global smooth structure so the low-order basis functions for the target domain tend to be the same as the corresponding basis functions for the source domain. On the other hand, the high-order basis functions for the target domain are affected by a small change of the domain so the target policy can be obtained from the target low-order basis functions that are similar to the source low-order basis functions and the high-order basis functions that are modified by a small change.

1. Perform random walk of M trials, each with maximum N steps on source domain and target domain and build the undirected graphs G_S, G_T respectively.
2. Construct the normalized Laplacian on G_S, G_T and solve the Laplacian to obtain the basis functions V_S^B, V_T^B. Sort them by eigenvalue in ascending order.
3. Approximate the source value function V_S^* using V_S^B by the least-square error fit method to obtain the weight w_i corresponding to the source basis function V_{Si}^B.
4. Transfer the weight w_i from the source basis function V_{Si}^B to the corresponding target basis function V_{Ti}^B.

$$V_T = \sum_i w_i * V_{Ti}^B$$

5. Convert the approximation target value function to the target policy.

Figure 2. Pseudo-code of the transfer algorithm.

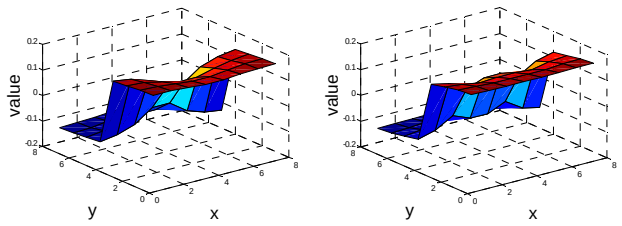


Figure 3. The similar structure of the basis functions of Figure 1(a) and 1(b).

4. EXPERIMENTS

First of all, we illustrate the basis functions of the graph Laplacian with different size of domains with the same topology. The upper two graphs and lower two graphs in Figure 4 and 5 show some low-order and high-order basis functions from graph Laplacian respectively. We note the two upper graphs in Figure 4 and 5 that represent the smoothest k basis functions of different domains respectively tend to be very similar while the lower graphs are not.

We design the experiments on scaling and topological domain transfer and evaluate the performance of an agent in the domains using different policies: random, transferred and optimal respectively. The agent is an active agent with ε-greedy behavior [1]. In other words, the agent has probability ε to act at random. A random policy selects an action at random, a transferred policy is obtained from the transfer method, and an optimal policy selects an action based on the optimal value function obtained by the value iteration method. The results are shown in Figure 6 and 8. The x-axis and the y-axis represent the number of states and the number of steps reaching the reward respectively. The diamond, square, and triangle lines represent the random, transferred and optimal policies respectively. Each point in the line represents the average number of steps reaching the reward state over all possible initial states.

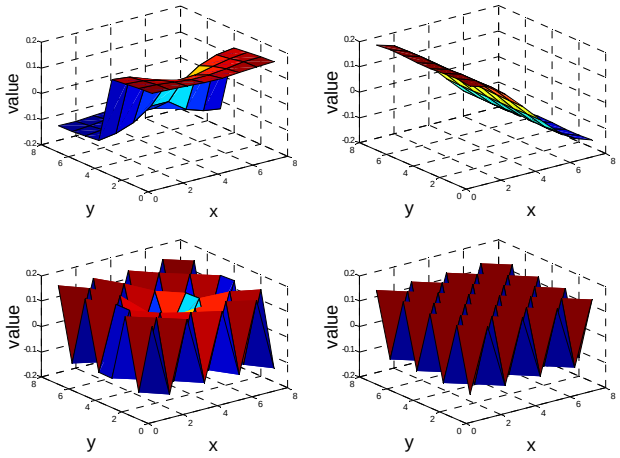


Figure 4. The basis functions of Figure 1(a).

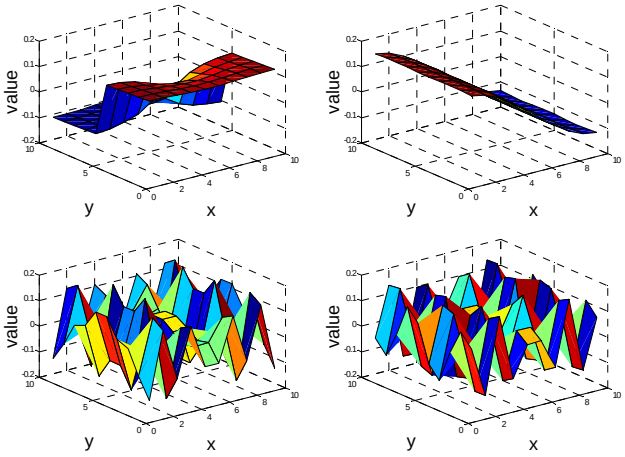
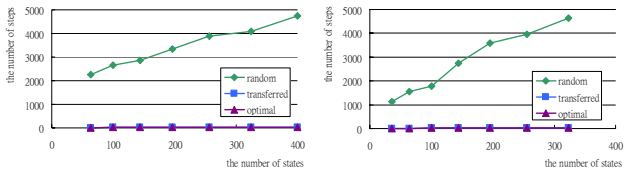


Figure 5. The basis functions of Figure 1(c).

4.1 Scaling Domain Transfer

These experiments investigate the effects of the scaling domain transfer. We separate the scaling domain transfer into two cases: up-scaling and down-scaling. The topology of each case is the same as shown in Figure 1(a). In up-scaling case, we choose the 6x6 grids world as a source domain and 8x8, 10x10, 12x12, 14x14, 16x16, 18x18, and 20x20 grids as target domains. In down-scaling case, we choose the 20x20 grids world as a source domain and 6x6, 8x8, 10x10, 12x12, 14x14, 16x16, and 18x18 grids as target domains. The results show that regardless of the size in a target domain, the transferred policy still performs very close to the optimal one as shown in Figure 6.

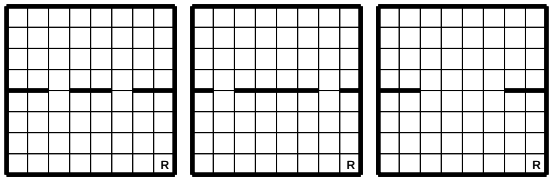


(a) up-scaling case (b) down-scaling case

Figure 6. The results of scaling domain transfer.

4.2 Topological Domain Transfer

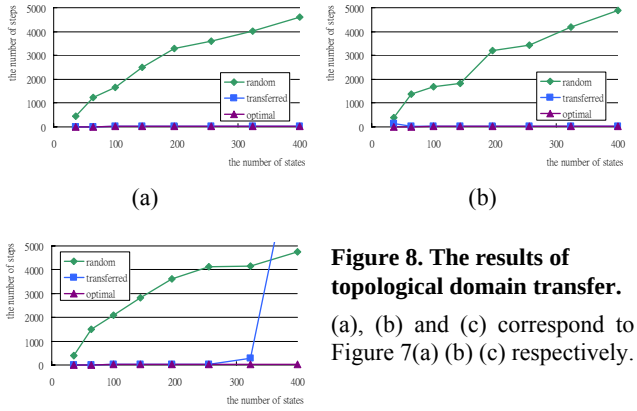
These experiments investigate the effects of the topological domain transfer. The topology in the source domain is shown in Figure 1(a) and we design three different topological cases as target domains. Figure 7(a) represents a case that splits the door into two separating doors, Figure 7(b) represents a case that splits the door into two separating doors farther, and Figure 7(c) represents a case that increases the size of a door.



(a) (b) (c)

Figure 7. The topological transfer targets.

The results demonstrate that if both domains are similar enough, the transferred policy may perform very close to the optimal one as shown in Figure 8(a). However, when the source and target domains are not similar enough as in the case of Figure 8(c) in which the larger size domains are more affected than the smaller ones. Besides, when the number of states is small, the effect of a small change in the domain is large, but when the number of states is large enough, the effect of a small change in the different size domains is similar as shown in Figure 8(b).



(a) (b)

(c)

Figure 8. The results of topological domain transfer.

(a), (b) and (c) correspond to Figure 7(a) (b) (c) respectively.

5. CONCLUSIONS

The theoretical analysis of the transfer method is based on the spectral analysis on graph Laplacian. The low-order basis functions of the graph Laplacian represent major features of a value function while the high-order ones represent miner features. If the low-order basis functions of the source and target domains are similar, the transfer method performs well. In other words, similar domains tend to keep similar distributions in low-order basis functions so we can transfer the weights of the source domain to the target domain and acquire a good approximate policy for the target domain. In this paper, we have proposed a domain transfer method based on the topology of the state space to support the transfer for reinforcement learning. Our experimental results show that if two domains are similar topologically, the policy learned from transfer learning can be very close to the optimal one. However, how to determine if a topological similarity is enough to apply the transfer learning to ensure its error bound be close to the optimality still needs more theoretical analysis. This work only considers the state space topology of the problem but not the rewards. We should revise the domain transfer method by considering how to map a state in a source domain to the corresponding one in a target domain that considers the rewards in future work.

6. ACKNOWLEDGMENTS

This work is supported by the National Science Council of Taiwan under grant number NSC 96-2628-E-007-044-MY3.

7. REFERENCES

[1] Richard S. Sutton and Andrew G. Barto. Reinforcement learning: an introduction. MIT Press, 1998.

[2] Matthew E. Taylor and Peter Stone. Cross-domain transfer for reinforcement learning. In Proceedings of the Twenty-fourth International Conference on Machine Learning, 2007.

[3] Andreas von Hessling and Ashok K. Goel. Abstracting reusable cases from reinforcement learning. In Proceedings of the Sixth International Conference on Case-Based Reasoning Workshop, 2005.

[4] Mattew E. Taylor, Shimon Whiteson, and Peter Stone. Transfer via inter-task mappings in policy search reinforcement learning. In Proceedings of the Sixth International Conference on Autonomous Agents and Multiagent Systems, 2007.

[5] Sridhar Mahadevan. Proto-value functions: developmental reinforcement learning. In Proceedings of the Twenty-second International Conference on Machine Learning, 2005.

[6] Ferguson Kimberly and Sridhar Mahadevan. Proto-transfer learning in Markov decision processes using spectral methods. In Proceedings of the Twenty-third International Conference on Machine Learning Workshop on Structural Knowledge Transfer for Machine Learning, 2006.

[7] Sridhar Mahadevan and Mauro Maggioni. Proto-value functions: a Laplacian Framework for learning representation and control in Markov decision processes. Technical Report, 2006.

[8] Fan R. K. Chung. Spectral graph theory. American Mathematical Society, 1997.

[9] Sheldon Axler, Paul Bourdon, and Ramey Wade. Harmonic function theory. Springer, 2001.

[10] Mikhail Belkin and Partha Niyogi. Semi-supervised learning on Riemannian manifolds. Machine Learning, 2004.

Autonomous Agent Learning using an Actor-Critic Algorithm and Behavior Models

(Short Paper)

Victor Uc Cetina
Department of Computer Science
Humboldt University of Berlin
Unter den Linden 6, 10099 Berlin, Germany
cetina@informatik.hu-berlin.de

ABSTRACT

We introduce a Supervised Reinforcement Learning (SRL) algorithm for autonomous learning problems where an agent is required to deal with high dimensional spaces. In our learning algorithm, behavior models learned from a set of examples, are used to dynamically reduce the set of relevant actions at each state of the environment encountered by the agent. Such subsets of actions are used to guide the agent through promising parts of the action space, avoiding the selection of useless actions. The algorithm handles continuous states and actions. Our experimental work with a difficult robot learning task shows clearly how this approach can significantly speed up the learning process and improve the final performance.

Categories and Subject Descriptors

I.2.6 [**Computing Methodologies**]: Artificial Intelligence—*learning*

General Terms

Algorithms, Experimentation

Keywords

reinforcement learning, behavior model, actor-critic

1. INTRODUCTION

The idea of supervision or advice giving was first proposed in 1958 by McCarthy [9]. More recently, Clouse and Utgoff [5] presented an online method of SRL. With this method, a human teacher monitors the agent's progress. If the teacher determines that the agent is not performing well, the teacher takes control and offers advice in the form of an action that is executed at that time. Then, the agent learns from such advice by reinforcing the tendency to choose the action recommended and performed by the teacher. Another attempt to accelerate the learning process was the one proposed by Lin [6]. He introduced a very effective way to speed up the Reinforcement Learning (RL) process through the replaying of experiences. The advice-giver provides complete sequences of states and actions $s_t, a_{t+1}, s_{t+1}, a_{t+2}, \ldots$ that the agent replays internally many times. By doing so backwards, the learning process is further accelerated. Maclin and Shavlik [7, 8] approached the advice giving problem in a different way, by using connectionist Q-learning. The advice-giver watches the learner and occasionally makes suggestions, expressed as instructions in a simple programming language. Using knowledge-based neural networks, those programs are included into the agent's utility function. Later, Rosenstein and Barto [10] proposed a combination of supervised learning with and actor-critic architecture. They used a supervised learning method to include the knowledge provided by a human supervisor into the actor-critic learning process. A composite actor is formed with the actor, a supervisor and a gain scheduler. The action executed at each moment is the result of a linear combination of the actions provided by the actor and the supervisor. Abbeel and Ng [1, 2] studied the use of an expert in order to learn to perform a task in situations where the reward function is not provided or it is difficult to design. The main idea consists of using inverse reinforcement learning to try to obtain the unknown reward function which is supposed to be implicit in the expert's behavior. Their method manages to get performances close to that of the expert. Moreover, Atkeson and Schaal [3] used human demonstrations to have a robot learn to perform a task. First, they learn from the demonstrations a reward function and a task model. Then, based on the learned reward function and task model, they compute a policy. Finally, another work that is worth to mention is that by Carpenter *et al* [4], who approached the problem of how to handle advice from several sources and also how to solve conflicting advices. Due to space constraints we can only mention those approaches more similar to ours.

All of the works mentioned above have one thing in common, the fact that the supervisor provides one action or sequences of actions which should be directly performed by the agent in order to learn from the expert. One exception is the method proposed by Rosenstein and Barto[10], where the action executed is a combination of the action suggested by the supervisor and the action selected by the agent. We use an approach where the action suggested by the supervisor is not directly executed by the agent, not even a modification of it. Instead, the action is used to dynamically generate a reduced set of current relevant actions. We call relevant actions to those actions in the close neighborhood

Cite as: Autonomous Agent Learning using an Actor-Critic Algorithm and Behavior Models (Short Paper), Victor Uc Cetina, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp. 1353-1356.

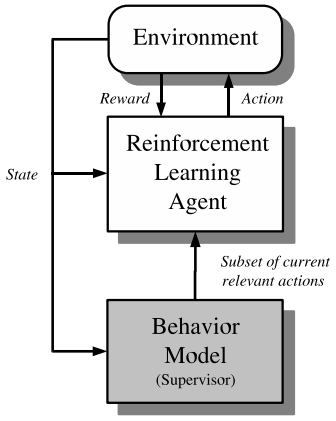


Figure 1: A supervised reinforcement learning architecture that uses a behavior model

of the action suggested by the supervisor. This subset of actions is then passed to the agent which uses it to select the next greedy action. The experiments performed show that our algorithm can significantly reduce the amount of training episodes required to learn a difficult task.

The rest of this paper is organized as follows. In Section 2 we explain the SRL architecture and introduce the algorithm. Section 3 contains the description of our testbed. In Section 4 we present the experimental results. Finally, in Section 5 we conclude the paper and mention our future work.

2. SUPERVISED REINFORCEMENT LEARNING

2.1 Architecture

The SRL architecture, which we have tested previously with Sarsa and Q-learning in [12], focuses on reducing the amount of exploration of the action space, by giving advice to the agent about what actions would be good to try, given the current state of the environment. Figure 1 illustrates the main idea of having the standard RL agent interacting with the environment, meanwhile a behavior model is used to provide the agent with a subset of current relevant actions. Such subset of actions includes the action suggested by the behavior model, and its n closest neighbor actions. Notice that two different actions are considered to be neighbors if they are expected to produce similar results in the environment, when they are applied in similar states.

Two are the key features in this architecture: (1) it allows to considerably reduce the amount of exploration of the action space, and (2) the supervisor does not need to be a perfect teacher with the possession of the optimal policy. Instead, its expertise is used to guide the agent through relevant parts of the action space and not explicitly indicating which action should be performed at each moment. Given that the behavior model can be seen as a Multilayer Perceptron (MLP), or any other Supervised Learning (SL) method, trained with a set of collected examples, it will always provide an action, and therefore the subset of actions can be generated. If the suggested action is wrong, then the reduced action set would probably be also wrong. In those cases, the greedy action selected by the agent could be simply seen as an exploratory action. Of course, our behavior model is expected to be as accurate as possible. Under this condition, the agent will always have much to win and nothing to lose.

2.2 Algorithm

The whole learning process is divided in two phases. In the first phase, an expert is used to generate a set of examples of the form $s_t \rightarrow a_{t+1}$. That is, given the current state of the environment s_t, knowing which is the action a_{t+1} that our expert would perform in the next time step. Using such examples and a SL method we build the behavior model. Once we have built the behavior model, we proceed with the second learning phase.

The second learning phase is shown in Algorithm 1, which is a modified version of the typical actor-critic algorithm described by Sutton and Barto [11]. At each state s the behavior model is used to generate what we call the expert action a_e. Such action a_e is considered to be a near optimal action and we use it to create the set of current relevant actions A_s, where $A_s \subset A$. Such subset A_s is defined by the interval $(a_e - \hat{B}, a_e + \hat{B})$, where $\hat{B}$ specifies how far from the expert action a_e we are willing to explore the action space. By doing so, the agent will always have to select the greedy action from the set of most promising actions, which causes an improvement in the learning rate. The optimal size of $\hat{B}$ grows inversely proportional to the accuracy of our behavior model. In other words, with more accurate behavior models, we need a smaller $\hat{B}$. Notice that the set A_s is used only to choose the greedy action. Random actions are selected from the whole set A. By doing so, we let the agent exploit the knowledge provided by the supervisor as much as possible, at the same time that we allow it to explore the whole action space looking for better actions that are beyond the knowledge of the same supervisor.

The first learning phase can be seen as a straightforward application of SL. Meanwhile, the second learning phase could be implemented using modified versions of any RL algorithm.

3. ROBOT DRIBBLING TASK

In the RoboCup simulation league, one of the most difficult skills that the robots can perform is dribbling. Dribbling can be defined as the skill that allows a player to run on the field while keeping the ball always in its kick range. In order to accomplish this skill, the player must alternate run and kick actions. The run action is performed through the use of the command (**dash** *Power*), while the kick action is performed using the command (**kick** *Power Direction*), where $Power \in [-100, 100]$ and $Direction \in [-180, 180]$. Such commands, belong to the set of basic commands provided by the simulator.

There are three factors that make this skill a difficult one to accomplish. First, the simulator adds noise to the movement of objects, and to the parameters of commands. This is done to simulate a noisy environment and make the competition more challenging. Second, since the ball must remain close to the robot without collisioning with it, and at the same time it must be kept in the kick range, the margin for error is small. And third, the most challenging factor, the use of heterogeneous players during competitions. Using heterogeneous players means that for each game the simulator generates seven different player types at startup, and

Algorithm 1: Supervised Actor-Critic Algorithm

initialize the weights vectors of the *Actor* and *Critic* arbitrarily
foreach *training episode* **do**
  initialize s
  take suggested action a_e from *Behavior Model*
  generate set $(a_e - \hat{B}, a_e + \hat{B})$
  take greedy action $a \in (a_e - \hat{B}, a_e + \hat{B})$
  with probability ϵ choose random action $a \in A$
  repeat for each step of episode
    perform action a, observe r, s'
    $TDError \leftarrow r + \gamma Critic(s') - Critic(s)$
    $TargetValue \leftarrow Critic(s) + \alpha TDError$
    train *Critic* with example $(s, TargetValue)$
    if $TDError > 0$ **then**
      train *Actor* with example (s, a)
    end
    take suggested action a'_e from *Behavior Model*
    generate set $(a'_e - \hat{B}, a'_e + \hat{B})$
    take greedy action $a' \in (a'_e - \hat{B}, a'_e + \hat{B})$
    with probability ϵ choose random action $a' \in A$
    $s \leftarrow s'$, $a \leftarrow a'$
  until s *is terminal*
end

the eleven players of each team are selected from this set of seven types. Given that each player type has different "physical" capacities, an optimal policy learned with one type of player is simply suboptimal when followed by another player of different type. In theory, the number of player types is infinite.

Due to these three reasons, a good performance in the dribbling skill is very difficult to obtain. Up today, even the best teams perform only a reduced number of dribbling sequences during a game. Most of the time the ball is simply passed from one player to another.

4. EXPERIMENTS AND RESULTS

For the first learning phase, we constructed our dribbling behavior model based on the Wright Eagle team, which is a RoboCup team with highly developed skills. We collected the information of 500 games and with the help of some scripts, we extracted the sequences of the games where a player managed to dribble for at least 3 meters. Once that we gathered the examples, we filtered them using an application developed specifically to identify and eliminate incorrect examples. The final set of 18,000 examples were used to train 2 multilayer perceptrons. One MLP learned to predict the dash power and the other the kick power. The input of both MLPs is the current state, seen as a 12-dimensional vector. This vector consists of the following variables: *player decay, dash power rate, kickable margin, kick rand, ball position x - player position x, ball position y - player position y, ball velocity x - player velocity x, ball velocity y - player velocity y, ball velocity x, ball velocity y, player velocity x, player velocity y.* The first 4 variables are some of the parameters that define a type of player, and for this problem, they were the most useful during our experimentation. The other 8 variables are needed to specify the current physical state of the ball and player. The output of the MLPs are

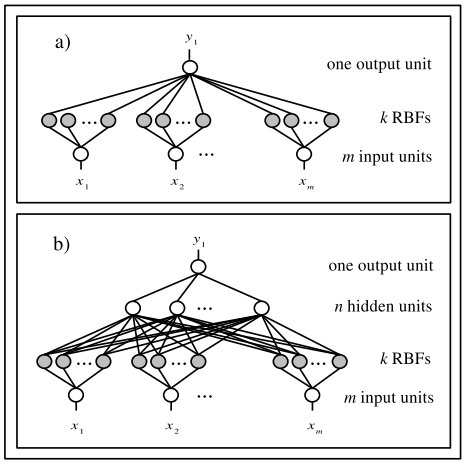


Figure 2: Different structures used to approximate the value function. (a) Radial basis functions. (b) Multilayer perceptron with one layer of radial basis functions

the dash power and kick power respectively, and together formed the behavior model. These MLPs predict the power of dashes and kicks with an error of 15 units. This error is big enough to prevent us from using those MLPs to directly control our agents. However, the knowledge encapsulated in them proved to be very useful when used as a supervisory source of information.

For the second learning phase, we implemented a RL agent that perceives the current state of the environment using the same input vector used by the behavior model. Each training episode was initiated placing the player in the center of the field with the ball besides it, at a distance of 0.5 meters, both with velocity zero. The training episodes were terminated either when the robot kicked the ball away from its kick margin, or when 35 actions were performed. The reward function gives always the scalar value resulting from the calculation of: 0.25(*player position x + ball position x + player velocity x + ball velocity x*). There is also a punishment of -100 everytime the player loses the ball or when there is a collision with it. The learning parameters were: $\epsilon = 0.3$, $\alpha = 0.01$ and $\gamma = 0.5$. A key design point when we work with reinforcement learning in continuous spaces is the structure used to approximate the value function. In our experimental work we employed two different structures: (1) an array of radial basis functions, and (2) a multilayer perceptron enhanced with one layer of radial basis functions. Such structures which are a linear and a non-linear function approximator respectively, are illustrated in Fig. 2.

Figure 3 shows the learning curves of the actor-critic algorithm using radial basis functions to approximate the value function, and 2 different implementations of the SRL algorithm, for different sizes of the relevant actions set A_s. The curves represent moving averages of size 1,000 that were averaged over 10 runs. From these results we can see that the supervised actor-critic method is clearly superior to the pure actor-critic version, when we use $\hat{B} = 15$. However when we use $\hat{B} = 10$, the resulting learning curve is worse than that obtained with the pure actor-critic algorithm. The reason for this is simple. We are reducing the exploration space much more than we should do, given the accuracy

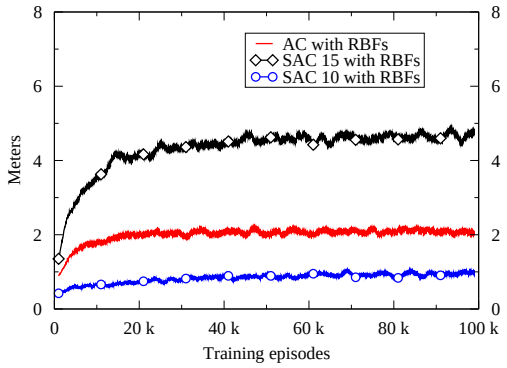


Figure 3: Learning curves of the Actor-Critic and the Supervised Actor-Critic algorithms using radial basis functions

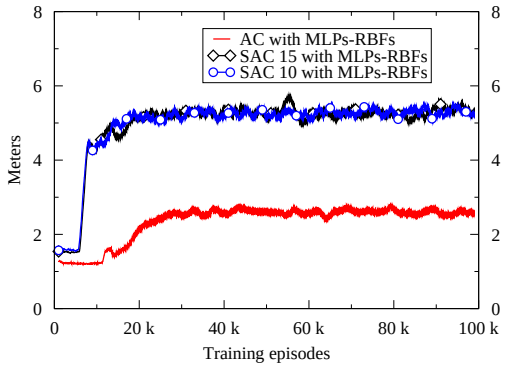


Figure 4: Learning curves of the Actor-Critic and the Supervised Actor-Critic algorithms using multilayer perceptrons with one layer of radial basis functions

of our behavior model. The typical actor-critic algorithm has a performance of 2 meters, after training for 100,000 episodes. Meanwhile, the best SRL algorithm, has a performance slightly under 5 meters.

In Fig. 4 we can see the learning curves of the same three algorithms, but using instead the non-linear value function. It is clear that the supervised actor-critic algorithm has also a much better performance than the typical actor-critic algorithm. Besides, we can see that the result with the simple actor-critic method is slightly better than that obtained with the linear value function in Fig. 3. We can also see that the performance of the supervised algorithm with $\hat{B} = 15$ is better than that obtained using a linear value function. Finally, when we check the performance of the supervised algorithm with $\hat{B} = 10$, something interesting occurs, the learning curve is identical to the curve obtained with the non-linear value function and $\hat{B} = 15$. In this case, the learning rate was not affected by the reduction of $\hat{B}$, as it happened when we used the linear function approximator. This difference is due to a better ability of the non-linear function approximator to generalize, which makes it more robust to changes in $\hat{B}$ than a linear function approximator.

5. CONCLUSION AND FUTURE WORK

We have presented one algorithm to implement supervised actor-critic learning. In our algorithm, behavior models previously learned from examples, are used to dynamically generate subsets of relevant actions at each moment. Using these subsets of actions, the agent can accelerate its learning rate. Performances obtained after 100,000 training episodes are better when we use the supervised version of the actor-critic algorithm, being more robust when the non-linear function approximator is used to represent the value function. We tested our algorithms with the robot dribbling problem, in the framework of the RoboCup simulation league. Such problem involves continuous state and action spaces with high dimensionality. Our future work will consider the use of eligibility traces and options, as a way to improve the final performances.

Acknowledgements This research work was supported by a PROMEP scholarship from the Education Secretariat of Mexico (SEP), and Universidad Autónoma de Yucatán.

6. REFERENCES

[1] P. Abbeel and A. Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the 21st International Conference on Machine Learning*, 2004.

[2] P. Abbeel and A. Y. Ng. Exploration and apprenticeship learning in reinforcement learning. In *Proceedings of the 22nd International Conference on Machine Learning*, 2005.

[3] C. Atkeson and S. Schaal. Robot learning from demonstration. In *Proceedings of the Fourteenth International Conference on Machine Learning*, 1997.

[4] P. V. M. Carpenter P., Riley and G. Kaminka. Integration of advice in an action-selection architecture. *RoboCup 2002: Robot Soccer World Cup VI. Lecture Notes in Computer Science*, 2003.

[5] J. A. Clouse and P. E. Utgoff. A teaching method for reinforcement learning. In *Proceedings of the Ninth International Workshop on Machine Learning*, 1992.

[6] L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, (8):293–321, 1992.

[7] R. Maclin and J. W. Shavlik. Incorporating advice into agents that learn from reinforcements. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1994.

[8] R. Maclin and J. W. Shavlik. Creating advice-taking reinforcement learners. *Machine Learning*, (22):251–282, 1996.

[9] J. McCarthy. Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, 1958.

[10] M. T. Rosenstein and A. G. Barto. Supervised actor-critic reinforcement learning. In *Learning and Approximate Dynamic Programming: Scaling Up to the Real World*. John Wiley & Sons, 2004.

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

[12] V. Uc-Cetina. Supervised reinforcement learning using behavior models. In *Proceedings of the 6th International Conference on Machine Learning and Applications*, 2007.

Teaching Sequential Tasks with Repetition through Demonstration

(Short Paper)

Harini Veeraraghavan
Computer Science Department
Carnegie Mellon University
harini@cs.cmu.edu

Manuela Veloso
Computer Science Department
Carnegie Mellon University
veloso@cs.cmu.edu

ABSTRACT

For robots to become prevalent in human environments, the robots need to be able to perform complex tasks often involving sequential repetition of actions. In this work, we present a demonstration-based approach to teach a robot generalized plans for performing sequential tasks with repetitions. We introduce action definitions through perception. Using the action definitions and the demonstration, the robot learns a task specific plan for tasks containing repetition of sub-sequences.

1. INTRODUCTION

A majority of tasks in human environments involve repetitions, be it assembling furniture using actions such as "HammerNail", "TightenScrew". For a robot to automatically generate a plan using the actions alone for a complex task such as assembling a furniture or performing some elaborate sequence of motions is very challenging. On the other hand, given an example demonstration, the robot can easily learn a task specific plan for performing the same task on different problems.

In this work, we contribute a demonstration-based approach to teach a robot task specific plans. We focus on real world domain and present an approach that learns task specific plans with repeating sub-tasks. Concretely, in our approach, both the human and robot actively participate in the learning task. Through demonstration the human instantiates the task specific actions. The robot learns the appropriate action definitions and then using the sequence of executed actions, learns a task specific plan with repetitions.

This paper is organized as follows. We first present the related work in Section 2 followed by the experimental domain and the basics of the teaching approach in Section 3. We present the learning approach in Section 4, an illustrative result in Section 5 and finally conclude the paper in Section 6.

2. RELATED WORK

Examples of works that actually implement a planning algorithm on a robot for learning to execute a task include the works in [3, 8]. Demonstration based learning approaches such as in [4, 5, 7] learn generalized plans for sequences of actions with little or no repetitions. The work in [1] uses demonstration-based learning for a single action. Another interesting approach to teaching sequential plans is in [9] where a robot learns a specific plan without any generalization or repetition from a user through simple spoken language dialogues. Our work to learning looping plans is most closely related to the work of Winner and Veloso [11] which learns domain specific plans from example demonstrations for completely defined domain specific actions in simulated domains. Another interesting approach to loop learning from demonstration with annotations is in [6]. Our approach differs from the afore-mentioned work in that the action definitions are themselves obtained through perception in a real world problem domain and the robot successfully learns a plan containing repetitions on sub-tasks.

Cite as: Teaching Sequential Tasks with Repetition through Demonstration (Short Paper), H. Veeraraghavan and M. Veloso, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril,Portugal,pp.1357-1360.

3. ELEMENTS OF LEARNING TASK AND EXPERIMENTAL DOMAIN

Figure 1: The robot used for the clear table task.

The reference task domain used in this work for learning plans with loops consists of clearing a table. Fig. 1 depicts the experimental platform and the robot used in this work. The task consists of applying a sequence of actions repeatedly to move all the objects from a table into a destination box. The individual actions such as *pick*, *drop*, *search*, etc are the set of skills that are pre-programmed into the robot. However, the robot does not know in what sequence the various actions need to be carried out to achieve a particular task, such as clearing the table. Similarly, the action definitions are general so that the robot does not know what objects are associated with each task. We call such actions robot behaviors. During demonstration, it learns the association of the robot behaviors to objects relevant to the particular task. We call such actions task-specific actions. The objects are identified by their color using color thresholding. We now define the different terms used in the paper.

- **Robot behavior** is a non-instantaneous sequence of physical actions performed by the robot that changes the state of the world. A behavior is composed of a number of primitive actions that are executed in a predefined sequence. The behaviors are pre-programmed into the robot. Example of a behavior is *PickObject(object)* which can be executed on any object.

- **Task-specific action** is an instantiation of the robot behavior but associated with a specific object type. For example, *PickObjectYellowBall(object - yellowballType)* is an instantiation of behavior *PickObject* but always executed only on objects that are yellow colored balls. The task specific action is also composed of a completely defined precondition and effects corresponding to the specific object types in its argument list. The task specific action with parameterized objects is referred to as task-specific operator or simply operator. The task specific action instantiated with specific objects such as *ball1* is referred to as a grounded action.

- **Predicate or Proposition** is a representation of the sensed measurements as facts or relations between different objects. Each proposition is associated with a visual measurment. For example, in order to verify the truth of a proposition such as *holdingObject(yellowBall)* , the robot first moves its hands to the level of its head and then checks if it can color segment the ball object.

- **State** is a set of observed predicates.

Robot Behavior A robot behavior is activated by the human demonstrator using an appropriate vision-based cue such as a colored card and can be executed on any object. Once associated with an object, a task-specific action is generated which is used for the task specific plan. The task specific actions on the other hand can only be executed on the specific object types associated with the input arguments. The object types in our case correspond to specific object color. The task specific actions are also generated during the demonstration phase.

- *searchObject_objecttype* : This action is performed by executing the following physical actions in order. (a) If object is not detected, follow table for a few time steps, else stop, (b) Move close to table and look for object, (c) If object is detected, stop, else go back to (a). The task specific action is renamed with the appropriate object type such as *searchObject_type1* for an object of type *type1* .

- *pickObject_objecttype* : This action is performed by executing (a) While object is not in center of image, move to align with object, (b) Grasp object with both hands.

- *carryObjectToBasket_objecttypeList* : This action is performed by executing the following physical actions: (a) If basket is not detected, hold object and follow table for a few time steps, else stop, (b) Move close to table and look for basket, (c) If basket is detected, stop, else go back to (a).

- *dropObjectIntoBasket_objecttypeList* : This action is performed as follows: (a) Release hands to drop the object.

Representing World Observations The effect of executing a behavior results in a change in the state of the world. In order to obtain the state of the world, the robot needs to have a knowledge of the relevant observations and know what measurements are required for each observation. These observations are represented in the plan as propositions. Thus, in addition to the behaviors, task specific actions and objects, the robot also contains a list of propositions with appropriate visual measurements. The set of propositions and the associated vision-based measurements used by the robot are as follows:

- *closeToObject_objecttype* Vision-based measurement checks whether the robot is standing sufficiently close to the object by measuring the objects relative distance from the robot. The proposition is set to true when the object is detected within a fixed pre-defined threshold τ from the robot.

- *holdingObject_objectype* To detect whether the robot is holding an object, it moves its arms to the level of its head and checks whether it can segment the appropriate object.

- *in_objectypeList* This is a binary predicate where the vision-based measurement checks which pair of objects in the argument list of action satisfy the condition for *in* . One object *obj1* is said to be inside the other object *obj2* when the bounding rectangle of the *obj1* is enclosed by the bounding rectangle of *obj2* .

4. TEACHING TASKS WITH REPETITIONS

The teaching approach consists of a demonstration phase and a learning phase. In the demonstration phase, the robot executes the sequence of robot behaviors on specific objects as indicated by the human. It then instantiates task specific actions from the behaviors, fills in the appropriate preconditions for those actions, and then learns a task specific plan for the executed action sequence.

4.1 Demonstration Phase

Demonstration is the first step in the teaching approach. To demonstrate the action sequence, the demonstrator indicates the appropriate actions and the objects relevant to the same action. Every action is associated with a specific color that can be identified by color thresholding upon viewing a correspondingly colored *action card*. The human indicates the objects by moving a laser pointer across the object. The robot tracks the laser spots and computes the region of interest as the bounding box enclosing all the tracked spots. It then identifies the appropriate object by matching the target model such as a color histogram or by applying an average (RGB) color threshold on the region of interest. As the goal of problem is not robust sensing, we eliminate perceptual ambiguities such as occlusions.

4.2 Learning Phase: Filling Action Preconditions and Effects

The first step to recognizing the plan for the demonstrated task is to extract the task specific action preconditions and effects. The algorithm for filling the action preconditions and effects is depicted in Algorithm 1. By operator we mean the task specific action associated with an object type. A grounded action on the other hand is a task specific action associated with a particular object corresponding to an object type. For task specific action definition extraction, we make use of the multiple occurrences of the same action instantiated on different objects but of the same type. In our case, the types correspond to the color. Given that the demonstrator is assumed to guide the robot through the actions in the correct order, in general the states preceding an action will contain all the predicates necessary to execute the same action. This in turn simplifies the extraction of the preconditions and effects. However, in the absence of multiple instantiations, a different learning algorithm such as [2]

Algorithm 1 Precondition and Effect Filling

Input: Grounded actions $\langle a_1,\ldots,a_N\rangle$ from demonstration
Input: Preceding and Succeeding states $\langle\{^-S_{a_1},^+S_{a_1}\}\ldots\{^-S_{a_N},^+S_{a_N}\}\rangle$ for each action
Output: Grounded actions $\langle a_1,\ldots,a_N\rangle$ with filled preconditions and effects

1: GROUP grounded actions into Operators $O_1,\ldots,O_k$, s.t. $\forall O_{op,op=1\ldots k}, \nexists a_j,a_k\{a_j,a_k \in op\}$, SUBSTITUTE$(a_j,a_k)$ is invalid
2: **For all** Operators op **do**
3: Collect the action states $\{^-S_{a_j},^+S_{a_j}\}\forall_j a_j \in op$.
4: Remove inconsistent action states.
5: **For all** Operators op **do**
6: Get $Preconditions^{op} \longleftarrow {}^-S_{a_1}\wedge\ldots\wedge{}^-S_{a_k}, a_1,\ldots,a_k \in op$
7: Effects:
8: **If** exists effect $e^x_{a_j}, a_j \in op \wedge \exists a_k \in op$ where $\forall_y e^y_{a_k}$ SUBSTITUTE$(e^x_{a_j}, e^y_{a_k})$ is invalid **then**
9: **If** exists predicates $c^{W=\{w_1,\ldots,w_m\}} \in {}^-S_{a_j}$ where $\arg(e^x_{a_j}) \cap \arg(c^W) \neq \emptyset \wedge \forall_{i,i\neq j}$, SUBSTITUTE$(e^x_{a_j}, e^y a_i)$ is invalid $\wedge\ c^W \ni {}^-S_{a_i}$, where $a_i, a_j \in op$ **then**
10: Add conditional Effect $condEffect^{op} \longleftarrow \{c^W, e^x_{a_j}\}$
11: **Else**
12: Add disjunctive Effect $Effects^{op} \longleftarrow e^x_{a_j} \vee Effects^{op}$
13: **Else If** $\exists$ effect $e^x_{a_j} \in \forall_i\{\triangle\langle{}^-S_{a_i},{}^+S_{a_i}\rangle\}a_i, a_j \in op$ **then**
14: Add conjunctive Effect $Effects^{op} \longleftarrow e^x_{a_j} \wedge Effects^{op}$
15: Fill in Preconditions and Effects for each action $a_j \in op$

will be more appropriate than the one presented in this work.

As shown in Algorithm 1, the inputs to the algorithm consists of the sequence of grounded actions obtained from the demonstration, and the corresponding states associated with each action. A state with negative '$-$' superscript such as $^-S_{a_j}$ corresponds to the state prior to executing the action a_j, while the state with positive superscript '$+$', such as $^+S_{a_j}$ corresponds to the state following the same action.

The first step in the Algorithm 1 is to group the grounded actions into their corresponding operators. The **SUBSTITUTE** procedure as shown in Line 1 of Algorithm 1 checks for the equality of two actions. In other words, substitute corresponds to replacing the parameters of one action for the other. So two grounded actions a_i, a_j correspond to the same operator when the result of their substitution is identical. Note that, here we are just comparing the action name and the arguments.

The next step of the algorithm is to collect the set of states corresponding to every action in each operator, following which, any inconsistent states are removed as depicted in Lines[2-4]. An inconsistent state is one where the intersection of the same (preceding) state corresponding to a specific grounded action with at least T (preceding) states corresponding to T grounded actions for the same operator is an empty set. In other words, for a pair of states $^-S_{a_k}, ^+S_{a_k}$ abbreviated as $^-S, ^+S$,

$$Inconsistent(^-S,^+S) = \begin{cases} \text{If} \quad \sum_{i=1}^{m}\{^-S_{a_k}\cap{}^-S_{a_j}\}\rightarrow\{\emptyset\} > T, & \text{true} \\ \text{Else,} & \text{false} \end{cases}$$

Only the states preceding the action are checked for inconsistency. However, both of $\{^-S_{a_j}{}^+S_{a_j}\}$ from an action a_j will be removed when $^-S_{a_j}$ is found to be inconsistent. The inconsistency check is performed to ensure that the precondition list of an operator is never an empty set. It is assumed that all preconditions for an action are conjunctive.

Next, the preconditions for the operator is obtained as the intersection of all the consistent preceding states for the associated actions as shown in Line 6 of the Algorithm 1. The procedure for obtaining the effects is depicted in Lines [8-14] of the Algorithm 1. The effects of a grounded action a_j corresponds to the set of predicates in the succeeding state $^+S_{a_j}$ occurring mutually exclusively from the preceding state $^-S_{a_j}$. This is represented as $\triangle\{^-S_{a_i},^+S_{a_i}\}$ on Line 13 of Algorithm 1. An effect $e^x_{a_j}$ of a grounded action a_j for operator op is added as a conditional effect when,

- there does not exist a substitution for the same effect $e^x_{a_j}$ in at least one other grounded action a_k in the same operator,
- there exists one or more predicates $c^W \in {}^-S_{a_j}$ where the intersection of the argument list of c^W with $e^x_{a_j}$ is not an empty set and there exists no substitution for the same predicates in any state $^-S_{a_k}$ where substitution for the effect $e^x_{a_j}$ is invalid.

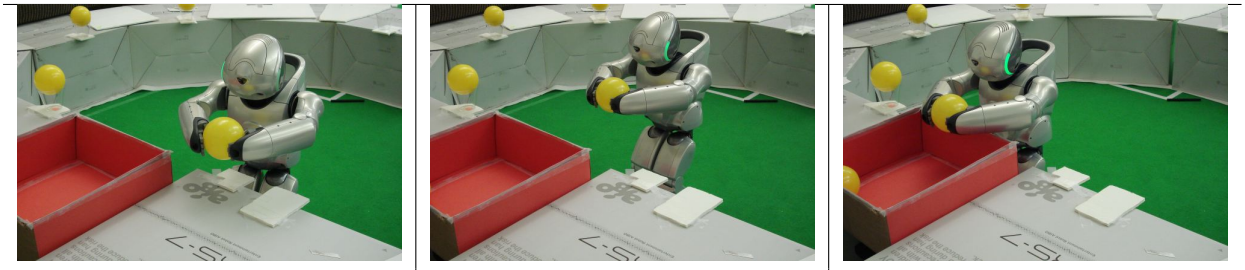
However, when no condition can be found, the effect $e^x_{a_j}$ is added as a disjunctive effect to the existing set of effects. Finally, the set of effects that occur in all the consistent action instantiations are added as conjuctive effects. At the end of this stage in learning, each action is represented in the classical PDDL format with preconditions and effects. As an example, the dropObject is represented as,

```
(:action dropObject_type1
 :parameters (?obj1 - type1 ?obj2 - type2)
 :preconditions (and (holdingObject_type1 obj1)
                     (closeToBasket_type2 obj2))
 :effects (and (in obj2 obj1)))
```

Figure 2: Representation of task specific dropObject action

4.3 Extracting Plans With Repetitions

Using the action definitions and the demonstration sequence, we then extract a partial ordering graph of the individual actions in the plan using [10] which links two steps in the demonstration that satisfy a producer-consumer relation where the producer step has an effect which is a precondition for the consumer step. The precondition is the rationale for the link. The last phase in learning from demonstration is to extract an executable planner from the demonstrated action sequence. In this work, we follow the similar definition as in programming languages for the *loops*. A sequence of actions forms a *loop* if and only if the same sequence of actions are repeated over different instances of the loop variable and there are no ordering constraints between actions occurring at different loop variable instances. The algorithm for learning the generalized plans from demonstration is depicted in Algorithm. 2. The first step transitively reduces the partial order graph for the action steps in the demonstration sequence to simplify computation for loop detection. In the next step, the actions are parameterized such that the grounded actions are replaced by actions with variables. Finally, the different steps and loops are arranged in the dependency order as obtained from the partial orderings. A set of actions forming a loop is merged with another loop when the actions in one loop are connected to the actions in the other loop through the producer-consumer orderings. Note that this merging still maintains the parallel execution of the steps along different branches of the merged loop.



Pick Object | Carry Object | Drop Object

Figure 3: Example sequence showing the execution of the task during demonstration.

Algorithm 2 Learning Looping Plans from Example

Input: Partial Order (PO) Graph
Output: Generalized Looping Plan
1: Transitively reduce PO Graph
2: Parameterize trace step actions
3: Detect LOOPS(Actions $a_1, \ldots, a_N$)
4: Order Steps by links.

5. ILLUSTRATIVE EXPERIMENT

Experiments were performed in indoor setting with the experimental setup as shown in Fig. 1. The objective of the experiment was to test whether the robot could learn a correct plan from the demonstrated sequence of actions. The domain is controlled such that no perceptual ambiguity such as occlusions or illumination variations occur. While restrictive for real world applications, robust sensing and planning with robust sensing is not the focus of this paper.

An example demonstration sequence consisted of the steps *searchObject, pickObject, carryObjectToBasket, dropObject* for two different balls repeated one after the other in the same order. An example of the task executed by the robot during the demonstration is depicted in Fig. 3. The plan learned from this demonstration sequence is depicted in Fig. 4. As shown, the robot is correctly able to learn an executable plan for the demonstrated sequence.

```
 while(?loopvar : type1)
if(haveObjectToSearch_type1 loopvar) then
   (searchObject_type1  loopvar)
if(closeToObject_type1 loopvar) then
   (pickObject_type1 loopvar)
if(holdingObject_type1 loopvar) then
  (carryObjectToBasket_type1_type2 loopvar obj2)
if(and (holdingObject_type1 loopvar)
       (closeToObject_type2 obj2))
   (dropObjectToBasket_type1_type2 loopvar obj2)
```

Figure 4: Example task specific plan.

6. CONCLUSIONS

In this work, we present an approach for teaching complex sequential tasks with repetitions through demonstration. We present a contribution where using the set of generic behaviors or skills and a demonstration, the robot can extract the task specific action definitions and finally a plan with repetitive execution of a sequence of actions. Additionally, the system has been implemented on a real world domain where the robot successfully transforms its executed actions and vision-based sensed measurements into an instantiated plan that can be used for learning an task specific planner.

7. ACKNOWLEDGEMENTS

The authors would like to SONY for making the QRIOs available to us for this project. The authors would also like to thank SONY for also making available the robot specific software libraries.

8. REFERENCES

[1] C. Breazeal, G. Hoffman, and A. Lockerd. Teaching and working with robots as a collaboration. In *Proc. Autonomous Agents and Multiagent Systems*, pages 1028–1035, 2004.

[2] Y. Gil. Learning by experimentation: incremental refinement of incomplete planning domains. In *Proc. Intl. Conf. on Machine Learning*, 1994.

[3] K. Haigh and M. Veloso. Interleaving planning and execution for asynchrnous user requests. *Autonomous Robots*, 5(1):79–95, 1998.

[4] N. Koenig and M. Mataric. Demonstration-based behavior and task learning. In *Working Notes AAAI Symposium To Boldly Go Where No Human-Robot Team Has Gone Before*, 2006.

[5] Y. Kuniyoshi, M. Inaba, and H. Inoue. Learning by watching: extracting reusable task knowledge from visual observation of human performance. *IEEE Trans. on Robotics and Automation*, 10(6):799–822, 1994.

[6] T. Lau, S. Wolfman, P. Domingos, and D. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.

[7] M. Nicolescu and M. Mataric. *Models and Mechanisms of Immitation and Social Learning in Robots, Humans, and Animals: Behavioral, Social and Communicative Dimensions*, chapter Task learning through immitation and human-robot interaction, pages 407–424. 2006.

[8] N. Nilsson. Shakey the robot. Technical Report 323, SRI International, AI Center, SRI International, Menlo Park, CA, 1984.

[9] P.E.Rybski, K. Yoon, J. Stolarz, and M. Veloso. Interactive robot task training through dialog and demonstration. In *Proc. Human Robot Interaction Conf.*, 2007.

[10] E. Winner and M. Veloso. Analyzing plans with conditional effects. In *Proc. Intl. Conf. Artificial Intelligence and Planning Systems*, pages 23–33, 2002.

[11] E. Winner and M. Veloso. Loopdistill: Learning domain-specific planners from example plans. In *In ICAPS Workshop on Planning and Scheduling*, 2007.

Adaptive Kanerva-based Function Approximation for Multi-Agent Systems

(Short Paper)

Cheng Wu and Waleed M. Meleis

ABSTRACT

In this paper, we show how adaptive prototype optimization can be used to improve the performance of function approximation based on Kanerva Coding when solving large-scale instances of classic multi-agent problems. We apply our techniques to the predator-prey pursuit problem. We first demonstrate that Kanerva Coding applied within a reinforcement learner does not give good results. We then describe our new adaptive Kanerva-based function approximation algorithm, based on prototype deletion and generation. We show that probabilistic prototype deletion with random prototype generation increases the fraction of test instances that are solved from 45% to 90%, and that prototype splitting increases that fraction to 94%. We also show that optimizing prototypes reduces the number of prototypes, and therefore the number of features, needed to achieve a 90% solution rate by up to 87%. These results demonstrate that our approach can dramatically improve the quality of the results obtained and reduce the number of prototypes required. We conclude that adaptive prototype optimization can greatly improve a Kanerva-based reinforcement learner's ability to solve large-scale multi-agent problems.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms, Experimentation

Keywords

Function approximation, Kanerva coding, Reinforcement learning, pursuit

1. INTRODUCTION AND RELATED WORK

Multi-agent problems can be difficult to solve by traditional machine learning techniques because the state space can be very large. The predator-prey pursuit problem [4] is a classic example of such a multi-agent problem. A general version of the problem takes place on a rectangular grid with one or more predator agents and one or more prey agents. Each grid cell is either open or closed, and an agent can only occupy open cells. Each agent has an initial position.

The problem is played in a sequence of time periods. In each time period, each agent can move to a neighboring open cell one horizontal or vertical step from its current location, or it can remain in its current cell. All moves are assumed to occur simultaneously, and more than one predator agent may not occupy the same cell at the same time. Each agent can observe the location of all other agents, and predator agents and prey agents can each communicate with agents of the same type. If a predator agent is in the same cell as a prey agent at the end of a time period, then that target has been caught. The goal is for the predator agents to catch all the prey agents in the shortest time.

Pursuit problems are difficult to solve in general. Closed-form solutions to restricted versions of the problem have been found [1, 7], but most such problems remain open. Researchers have used approaches such as genetic algorithms [5] and reinforcement learning [12] to develop solutions.

Reinforcement learning [11] is, in some respects, well-suited to solving multi-agent problems, and Q-learning [13] has emerged as one of the most successful reinforcement learning strategies. The algorithm works by combining state space exploration and exploitation to learn the value of each state-action pair. Through repeated trials, the estimates of the values of each state-action pair can gradually converge to the true value, and these can be used to guide the agent to maximize its reward. Under certain limited conditions, Q-learning has been shown to converge to an optimal policy.

A key limitation on the effectiveness of Q-learning is the size of the table needed to store the state-action values. The requirement that an estimated value be stored for every state-action pair limits the size and complexity of the learning problems that can be solved. Instead, function approximation [3] can be used to store an approximation of this table. Many approximation techniques exist, including coarse coding [6], and tile coding [2], and there are guarantees on their effectiveness in some cases [11].

Sparse distributed memories [8] can also be used to reduce the amount of memory needed to store the state-action value table. This approach applied to reinforcement learning, also called Kanerva Coding [11], represents a function approximation technique that is particularly well-suited to problem domains with high dimensionality. A collection of k *prototype state-action pairs*, (prototypes) is selected, each of which again corresponds to a binary feature. A state-action pair and a prototype are said to be *adjacent* if their bit-wise

Cite as: Adaptive Kanerva-based Function Approximation for Multi-Agent Systems (Short Paper), Cheng Wu, Waleed M. Meleis, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp. 1361-1364.

representations differ by no more than 1 bit. A state-action pair is represented as a collection of binary features, each of which equals 1 if and only if the corresponding prototype is adjacent. A value $\theta(i)$ is maintained for the ith feature, and an approximation of the value of a state-action pair is then the sum of the θ values of the adjacent prototypes. In this way, Kanerva Coding can greatly reduce the size of the value table that needs to be stored.

If the number of prototypes is very large relative to the number of state-action pairs, and the prototypes are uniformly distributed through the state space, each prototype will be adjacent to a small number of state-action pairs. In this case, the approximate state-action values will tend to be close to the true values, and the reinforcement learner will operate as usual. However if the number of prototypes is small, or if the prototypes themselves are not well chosen, the approximate values will not be similar to the true values and the reinforcement learner will give poor results.

Adaptively choosing prototypes appropriate to the particular application is an important way to contribute prior knowledge and experience to the reinforcement learner. There is therefore a need for algorithms to select prototypes that can span the state-space for a particular application. There have been few published attempts to apply Kanerva coding to multi-agent problems [9] or to evaluate and improve the quality of sets of prototypes.

Ratitch [10] has shown that sparse distributed memories can be used to represent the value table in a reinforcement learner. However, they add and delete locations only when the number of locations activated by an individual sample is below a fixed threshold. This approach may overreact to individual samples, in contrast to our approach which considers all samples and all prototypes in a training run before adding and deleting locations. Also, the deterministic nature of their decision to delete a prototype is less flexible than our probabilisitic approach.

2. PROTOTYPE OPTIMIZATION

When two different state-action pairs visited during Q-learning are mapped to the same subset of the prototypes, a prototype *collision* is said to have taken place. Both state-action pairs will necessarily have the same approximate value, at least one of which may be far from its true value. Selecting a set of prototypes that minimizes collisions will maximize the solver's ability to solve the problem.

However it is difficult to generate an optimal set of prototypes for several reasons: the space of possible subsets is very large and the state-action pairs encountered by the solver depend on the specific problem instance being solved. We therefore investigate several heuristic solutions to the prototype optimization problem.

We say that a prototype is *visited* during Q-learning if it is adjacent to the current state-action pair. If a specific prototype is rarely visited, it implies that few state-action pairs are adjacent to this prototype. This suggests that this prototype is inappropriate for the particular application. On the contrary, if a specific prototype is visited frequently, it implies that too many state-action pairs are adjacent to the prototype and collisions are more likely to occur. A necessary condition for collisions to be minimized is that most prototypes are visited an average number of times.

The frequency distribution of visits to prototypes over a sample run using Q-learning with Kanerva coding is shown in Figure 1 both before and after prototype optimization. This example is an instance of pursuit with a 32x32 grid, two predator agents, and one prey agent. The non-uniform distribution of visit frequencies across prototypes before prototype optimization indicates that some prototypes are frequently visited and others are rarely visited.

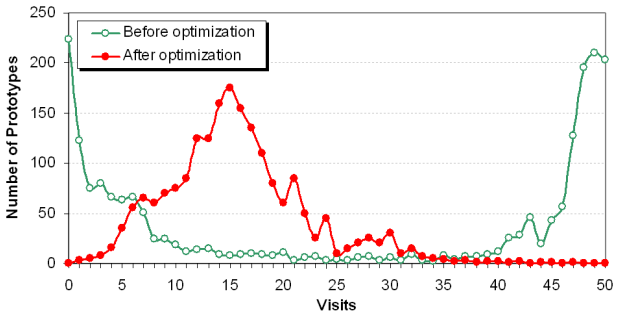


Figure 1: Distribution of the number of visits per prototype in a sample run.

We can optimize prototypes using visit frequencies. We divide the original prototypes into three categories: prototypes with low visit frequency, prototypes with high visit frequency, and the rest of the prototypes. Prototype optimization attempts to replace those prototypes with low or high frequency with prototypes that will have average visit frequencies, as shown in Figure 1.

We describe and evaluate different optimization mechanisms to achieve this goal. In each case, initial prototypes are selected randomly from the entire space of possible state-action pairs. Q-learning with Kanerva coding is used to develop policies for the predator agents, while keeping track of the number of visits to each prototype. After a fixed number of iterations, we update the prototypes using one of the mechanisms described below.

2.1 Prototype deletion

Prototypes that are rarely visited do not contribute to the solution of instances. Similarly, prototypes that are visited frequently are likely to cause many collisions. It makes sense to delete these prototypes and replace them with new prototypes with average frequencies. We evaluate the following two algorithms for deleting prototypes.

In the first approach, we periodically delete a fraction of prototypes whose visit frequency is lowest, and a fraction of prototypes whose visit frequency is highest. The fraction of prototypes that is deleted slowly decreases as the algorithm runs. The θ value and visit frequency of the new prototype is initially set to zero. We refer to this approach as deterministic prototype deletion.

An advantage of this algorithm is that it is easy to implement and it uses application- and instance-specific information to guide the deletion of rarely or heavily visited prototypes. However, this approach deletes prototypes deterministically which does not give the solver the flexibility to keep some prototypes that are rarely or frequently visited. For example, if the number of prototypes is very large, some prototypes that might become useful will not be visited in an early epoch and will be deleted.

In the second approach, we delete prototypes with a probability equal to an exponential function of the number of vis-

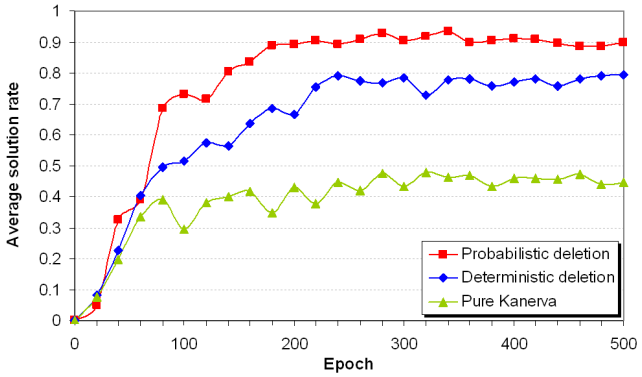


Figure 2: Effect of prototype deletion.

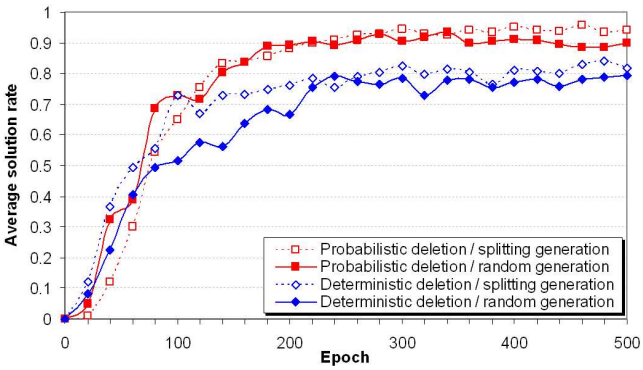


Figure 3: Effect of prototype generation.

its. I.e. the probability p_{del} of deleting a prototype whose visit frequency is v is $p_{del} = \lambda e^{-\lambda v}$, where λ is a parameter that can vary from 0 to 1. In this approach, prototypes that are rarely visited tend to be deleted with a high probability, while prototypes that are frequently visited are rarely deleted (we describe how we reduce the visit frequency of heavily visited prototypes in the next section). We refer to this approach as probabilistic prototype deletion.

2.2 Prototype generation

We replace prototypes that have been deleted with new prototypes that will tend to improve the behavior of the function approximation. We evaluate the following two algorithms for generating prototypes.

In the first approach, new prototypes are generated randomly from the entire state space. While this approach aggressively searches the state space for useful prototypes, it does not use domain- or instance-specific information.

In the second approach, we create new prototypes by applying prototype splitting. A prototype s_1 that has been visited the most times is selected, and a new prototype s_2 that is a neighbor of s_1 is created by inverting a fixed number of bits in s_1. The prototype s_1 remains unchanged.

This approach creates new prototypes near prototypes with the highest visit frequencies. These prototypes are similar but distinct which tends to reduce the number of visits to nearby prototypes, and therefore the number of collisions they cause.

3. EXPERIMENTAL EVALUATION

We evaluate our prototype optimization algorithms by applying them to random predator-prey pursuit instances on a 32x32 grid with two non-communicating predator agents and one prey agent. Each predator agent can see the position of the prey agent. Each agent can select one of 9 possible actions, moving one step in any of 8 directions, or not moving. Each grid instance has 32 random closed cells.

In each epoch, we apply each learning algorithm with 1984 prototypes to 40 random training instances followed by 40 random test instances. Prototype optimization is applied after every 20 epochs. For every 20 epochs, we record the average fraction of test instances within those epochs that are solved within a maximum of 64 moves.

The effect of different prototype deletion algorithms is shown in Figure 2. The figure shows the average fraction of test instances solved over a series of epochs for three algorithms: the pure Kanerva coding algorithm that uses no prototype optimization, deterministic deletion, and probabilistic deletion algorithms. These deletion algorithms use random prototype generation.

The algorithms converge after about 200 epochs, and the results show that the pure Kanerva algorithm solves approximately 45% of the test instances, the deterministic-deletion algorithm solves approximately 79% of the test instances, and the probabilistic-deletion algorithm solves approximately 90% of the test instances. These results indicate that dynamically deleting and regenerating prototypes can significantly increase the quality of the results. The results also indicate that probabilistic prototype deletion significantly outperforms deterministic deletion.

The effect of different prototype generation algorithms is shown in Figure 3. The figure shows the average fraction of test instances solved over a series of epochs for all four combinations of deletion and generation algorithms.

The algorithms converge after about 240 epochs, and the results show that prototype splitting raises the fraction of test instances solved from 79% to 82% with deterministic prototype deletion, and from 90% to 94% with probabilistic prototype deletion. These results indicate that prototype splitting can improve the quality of the results by a small but noticeable amount.

The effect of varying the parameter λ in the exponential distribution used to delete prototypes in the probabilistic deletion algorithm is shown in Table 1. The table shows the average fraction of test instances solved over a range of λ values with either random prototype generation or prototype splitting. The results show that the best results are achieved when $\lambda = 1$ for both prototype generation algorithms.

Figure 4 shows the minimum number of prototypes needed to solve an average of 90% of test instances over a range of grid sizes. The results compare the pure Kanerva algorithm with the probabilistic-split algorithm with $\lambda = 1$. The al-

λ	0	0.5	0.8	1
Random generation	56.25%	77.00%	86.38%	90.40%
Splitting generation	60.51%	82.63%	94.13%	94.25%

Table 1: The effect of λ under probabilistic deletion

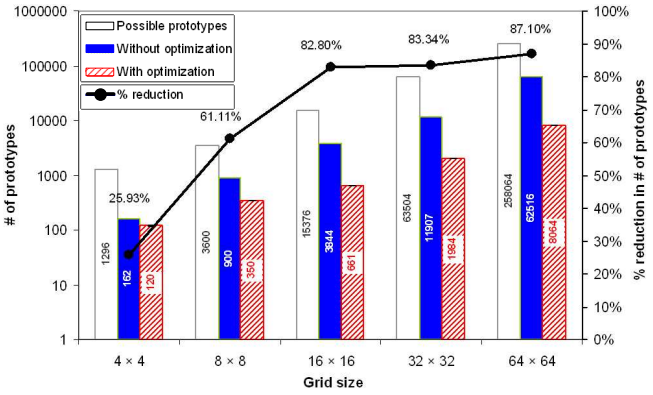


Figure 4: Minimum number of prototypes to solve an average of 90% of instances, and % reduction.

gorithm is run for 500 epochs and the average solution rate is measured over the next epoch. The results are computed by initially setting the number of prototypes equal to the total number of possible prototypes. After 500 epochs, if the result is greater than 90%, the number of prototypes is gradually decreased and the results are recomputed. This process continues until the solution rate is less than 90%. We report the minimum number of prototypes needed to solve an average of 90% of the test instances, which is shown on a logarithmic scale. Figure 4 also shows the total number of possible prototypes and the percent reduction in the number of prototypes needed.

The results show that prototype optimization dramatically reduces the number of prototypes needed to achieve a 90% solution rate. For example, on a 64x64 grid the number of prototypes needed is reduced from 62,516 to 8,064, a reduction of 87%.

We show an example of the policy learned after 500 epochs using our adaptive Kanerva-based function approximation algorithm in Figure 5. This example is an instance of pursuit with a 32x32 grid, one prey agent which starts on the left, and two predator agents.

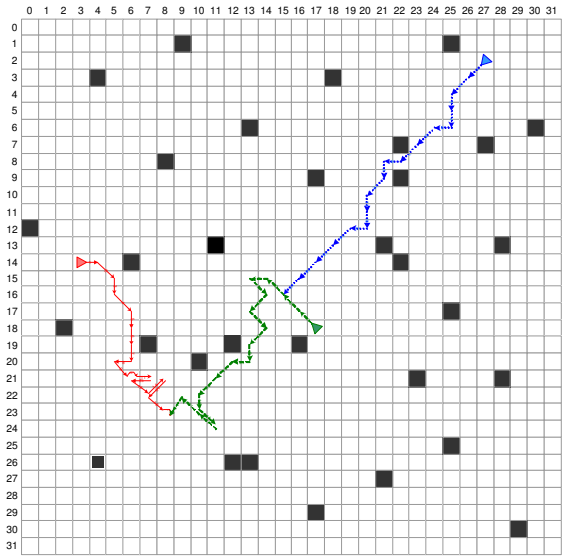


Figure 5: Sample policy

4. CONCLUSIONS

We have shown that pure Kanerva-based function approximation applied within a reinforcement learner does not give good results. We described our new adaptive Kanerva-based function approximation algorithm, based on prototype deletion and generation. We showed that probabilistic prototype deletion with random prototype generation increases the fraction of test instances that are solved from 45% to 90%, and that prototype splitting increases that fraction to 94%. We also showed that optimizing prototypes reduces the number of prototypes, and therefore the number of features, needed to achieve a 90% solution rate by up to 87%.

These results demonstrate that our approach can dramatically improve the quality of the results obtained and reduce the number of prototypes required. We conclude that adaptive prototype optimization can greatly improve a Kanerva-based reinforcement learner's ability to solve large-scale multi-agent problems.

5. REFERENCES

[1] M. Adler, H. Racke, N. Sivadasan, C. Sohler, and B. Vocking. Randomized pursuit-evasion in graphs. In *Proc. of the Intl. Colloq. on Automata, Languages and Programming*, 2002.

[2] J. Albus. *Brains, Behaviour, and Robotics*. McGraw-Hill, 1981.

[3] L. Baird. Residual algorithms: Reinforcement learning with function approximation. In *Proc. of the 12th Intl. Conf. on Machine Learning*. Morgan Kaufmann, 1995.

[4] M. Benda, V. Jagannathan, and R. Rodhiawalla. On optimal cooperation of knowledge sources. *Technical Report, Boeing Computer Services*, 1985.

[5] T. Haynes and S. Sen. The evolution of multiagent coordination strategies. *Adaptive Behavior*, 1997.

[6] G. Hinton. Distributed representations. *Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh*, 1984.

[7] V. Isler, S. Kannan, and S. Khanna. Randomized pursuit-evasion with local visibility. *SIAM Journal on Discrete Mathematics*, 20(1):26–41, 2006.

[8] P. Kanerva. *Sparse Distributed Memory*. MIT Press, 1988.

[9] K. Kostiadis and H. Hu. KaBaGe-RL: Kanerva-based generalisation and reinforcement learning for possession football. In *Proc. of IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, 2001.

[10] B. Ratitch and D. Precup. Sparse distributed memories for on-line value-based reinforcement learning. In *Proc. of the European Conf. on Machine Learning*, 2004.

[11] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. Bradford Books, 1998.

[12] M. Tan. Multi-agent reinforcement learning: Independent vs. cooperative learning. In M. N. Huhns and M. P. Singh, editors, *Readings in Agents*, pages 487–494. Morgan Kaufmann, CA, 1997.

[13] C. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1989.

Efficient Multi-Agent Reinforcement Learning through Automated Supervision

(Short Paper)

Chongjie Zhang
Computer Science Department
140 Governors Drive
University of Massachusetts
Amherst, MA 01002-9264
chongjie@cs.umass.edu

Sherief Abdallah
Institute of Informatics
British University in Dubai
Knowledge Village, Block 17
Dubai, United Arab Emirates
sherief.abdallah@buid.ac.ae

Victor Lesser
Computer Science Department
140 Governors Drive
University of Massachusetts
Amherst, MA 01002-9264
lesser@cs.umass.edu

ABSTRACT

Multi-Agent Reinforcement Learning (MARL) algorithms suffer from slow convergence and even divergence, especially in large-scale systems. In this work, we develop a supervision framework to speed up the convergence of MARL algorithms in a network of agents. The framework defines an organizational structure for automated supervision and a communication protocol for exchanging information between lower-level agents and higher-level supervising agents. The abstracted states of lower-level agents travel upwards so that higher-level supervising agents generate a broader view of the state of the network. This broader view is used in creating supervisory information which is passed down the hierarchy. We present a generic extension to MARL algorithms that integrates supervisory information into the learning process, guiding agents' exploration of their state-action space.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence

General Terms

Algorithms, Experimentation

Keywords

Reinforcement Learning, Multiagent Systems, Supervision, Heuristics

1. INTRODUCTION

The main contribution of this paper is the development of a framework that speeds up the convergence of Multi-Agent Reinforcement Learning (MARL) algorithms [2, 6] in a network of agents. Each agent's learning occurs in the context of a limited set of agents. We call this set of agents the agent's neighborhood that is specified as an overlay network.

Cite as: Efficient Multi-Agent Reinforcement Learning through Automated Supervision (Short Paper), Chongjie Zhang, Sherief Abdallah, Victor Lesser, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons(eds.),May,12-16.,2008,Estoril,Portugal,pp. 1365-1368.

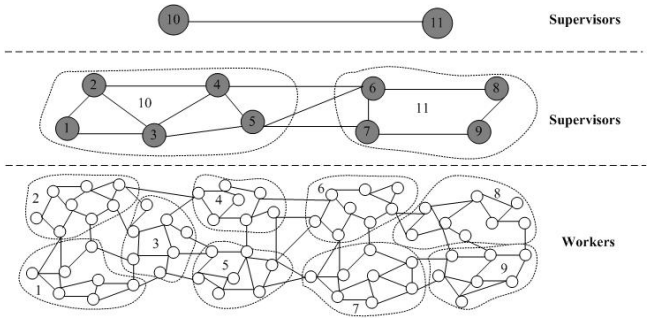


Figure 1: An organization structure for multi-level supervision

The slowness of MARL convergence is due to the large policy search space. Each agent's policy not only includes its local state and actions but also some characteristics of the states and actions of its neighboring agents [2], or the state size of each agent may be proportional to the size of the system [6]. Convergence is also affected by the non-stationarity of the environment (other agents are simultaneously learning their own policies).

Our framework consists of three main components: a multi-level supervision organization (a meta-organization built on top of the agents' overlay network), a communication protocol for exchanging information between lower-level agents and higher-level supervising agents, and a generic extension to MARL algorithms that integrates supervisory information into the learning process. The key idea of our framework is as follows. Each level in the supervising organization is an overlay network in itself. For example, Figure 1 shows a three-level supervision organizational structure. The abstracted states of lower-level agents travel upwards so that higher-level supervising agents can generate a broader view of the state of the network. This broader view comes from not only information about the states of lower-level agents but also information from neighboring supervising agents. In turn, this broader view results in creating supervisory information which is passed down the hierarchy. The supervisory information guides agents in collectively exploring their state-action spaces more efficiently, and consequently results in faster convergence.

2. RELATED WORK

Two paradigms have been studied to speed up the learning process. The first paradigm is to reduce the policy search space. For example, the TPOL-RL [10] reduced the state space by mapping states onto a limited number of action-dependent features. The hierarchical multi-agent reinforcement learning [7] used the explicit task structure to restrict the space of policies, where each agent learned joint abstract action-values by communicating with each other only the state of high-level subtasks. The second paradigm is to use heuristics to guide the policy search. The work described in [11] used both local and global heuristics to accelerate the learning process in a decentralized multirobot system. The local heuristic used only the local information and the global heuristic used the information that was shared and required to be exactly the same among robots. The Heuristically Accelerated Minimax-Q (HAMMQ) [4] incorporated heuristics into the Minimax-Q algorithm to speed up its convergence rate, which shared the convergence property with Minimax-Q. HAMMQ was intended for a two-agent configuration and further the authors had no discussion about how heuristics were constructed.

Our approach follows the second paradigm that uses heuristics to guide the policy search. However, it differs from other approaches in a key respect that it defines a decentralized hierarchical supervision mechanism to automate the generation of heuristics and integrates heuristics into existing unsupervised MARL algorithms (e.g., ReDVaLeR [3], WoLF-GIGA [5], WPL [1], etc.) in a generic manner to speed up their convergence.

3. ORGANIZATIONAL SUPERVISION

Supervision mechanisms commonly exist in human organizations (e.g., enterprises and governments), whose purpose is to run an organization effectively and efficiently to fulfill organizational goals. Supervision involves gathering information, making decisions, and providing directions to regulate and coordinate actions of organization members. The practical effectiveness of the supervision in human organizations, especially in large organizations, inspired us to introduce a similar mechanism into multi-agent systems to improve the efficiency of MARL algorithms.

To add a supervision mechanism to a MAS with an overlay structure, we adopt a multi-level, clustered organizational structure. Agents in the original overlay network, called workers, are clustered based on some measure (e.g., geographical distance). Each cluster is supervised by one agent, called the supervisor, and its member agents are called subordinates. The supervisor role can be played by a dedicated agent or one of the workers. If the number of supervisors is large, higher-level supervisors can be added, and so on, forming a multi-level supervision structure.

Two supervisors at the same level are adjacent if and only if at least one subordinate of one supervisor is adjacent to at least one subordinate of the other. Communication links, which can be physical or logical, exist between adjacent workers, adjacent supervisors, and subordinates and their supervisors. Figure 1 shows a three-level organizational structure. The bottom level is the overlay network of workers which forms 9 clusters. A shaded circle represents a supervisor, which is responsible for a corresponding cluster. Note that links between subordinates and their supervisors are omitted in this figure.

4. COMMUNICATION PROTOCOL

Three types of communication messages, *report*, *suggestion*, and *rule*, are used. A worker's report passes its activity data upwards to provide its supervisor with a broader view. A supervisor's report aggregates the information of reports from its subordinates. A supervisor sends its report to its adjacent supervisors at the same level in addition to its immediate supervisor (if any). The supervisor's view is based on not only the agents that it supervises (directly or indirectly) but also its neighboring supervisors. This peer-supervisor communication allows each supervisor to make rational local decisions when directions from its immediate supervisor are unavailable. To prevent supervisors from being overwhelmed and reduce the communication overhead in the network, the information is summarized (abstracted) in reports. Furthermore, reports are only sent periodically.

Based upon this information, a supervisor employs its expertise, integrates directions from its superordinate supervisor, and provides supervisory information to its subordinates. As in human organizations, rules and suggestions are used to transmit supervisory information. A *rule* is defined as a tuple $\langle c, F \rangle$, where

- c: a condition specifying a set of satisfied states
- F: a set of forbidden actions for states specified by c

A *suggestion* is defined as a tuple $\langle c, A, d \rangle$, where

- c: a condition specifying a set of satisfied states.
- A: a set of actions
- d: the suggestion degree, whose range is $[-1, 1]$.

A suggestion with a negative degree, called a *negative suggestion*, urges a subordinate not to do the specified actions. In contrast, a suggestion with a positive degree, called a *positive suggestion*, encourages a subordinate to do the specified action. The greater the absolute value of the suggestion degree, the stronger the impact of the suggestion on the supervised agent.

Each rule contains a condition specifying states where it can be applied. Subordinates are required to obey rules from their supervisors. Due to their imperativeness, correct rules greatly improve the system efficiency, while incorrect rules can lead to inefficient policies. In contrast, suggestions are used to express a supervisor's preference for subordinates' behavior, which may not be completely correct. Therefore, a subordinate does not rigidly adopt suggestions. The effect of a suggestion on a subordinate's local decision making may vary, depending on its current policy and state. A supervisor will refine or cancel rules and suggestions as new or updated information from its subordinates become available.

A set of rules are in conflict if they forbid all possible actions on some state(s). Two suggestions are in conflict if one is positive and the other is negative and they share some state(s) and action(s). A rule conflicts with a suggestion if a state-action pair is forbidden by the rule but is encouraged by the suggestion. In our supervision mechanism, we assume each supervisor itself is rational and will not generate rules and suggestions that are in conflict. However, in a multi-level supervision structure, a supervisor's local decision may

conflict with its superordinate direction. Rules have higher priority than suggestions. There are several strategies for resolving conflicts between rules or between suggestions, such as always taking its superordinate or local rule, stochastically selecting a rule, or requesting additional information to make a decision. The strategy choice depends on the application domain. Note that it may not always be wise to select the superordinate decision, because, although the superordinate supervisor has a broader view, its decision is based on abstracted information. Our strategy for resolving conflicts picks the most constraining rule and combines suggestions by summing the degrees of the strongest positive suggestion and the strongest negative suggestion.

5. MARL UNDER SUPERVISION

Using MARL, each agent gradually improves its action policy as it interacts with other agents and the environment. A *pure* policy deterministically chooses one action for each state. A *mixed* policy specifies a probability distribution over the available actions for each state. Both can be represented as a function $\pi(s, a)$, which specifies the probability that an agent will execute action a at state s. As argued in [9], mixed policies can work better than pure policies in partially observable environments, if both are limited to act based on the current percept. Due to partial observability, most MARL algorithms are designed to learn mixed policies. The rest of this section shows how MARL algorithms learning mixed policies can take advantage of higher-level information specified by rules and suggestions to speed up convergence.

A typical MARL algorithm contains two components: policy (or action-value function) updating and action choice based on the learned policy. One common method to speed up learning is to supply an agent with additional reward to encourage some particular actions [8]. The use of the special reward affects both policy updating and action choice. In a multi-agent context, special rewards may generate a policy that is undesirable in that they may distract from the main goal, which is supported by the normal reward. In contrast, our approach directly biases the action selection for exploration without changing the policy update process. Hence its effect on the final learned policy is transient (can be turned off at any time), while reward shaping has a permanent effect.

As described previously, a rule explicitly specifies undesirable actions for some states and is used to prune the state-action space. Suggestions, on the other hand, are used to bias agent exploration. The strategy adopted for integrating suggestions into MARL is that the lower the probability of a state-action pair, the greater the effect a positive suggestion has on it and the less the effect a negative suggestion has on it. The underlying idea is intuitive. If the agent's local policy already agrees with the supervisor's suggestions, it is going to change its local policy very little (if at all); otherwise, the agent follows the supervisor's suggestions and make a more significant change to its local policy.

Let R and G be the rule set and suggestion set, respectively, that a worker received and π be its policy. We define $R(s, a) = \{r \in R|$ state s satisfies the condition $r.c$ and $a \in r.F\}$[1] and $G(s, a) = \{g \in G|$ state s satisfies the condition $g.c$ and $a \in g.A\}$. Then a new function π^{AC} for the action choice is defined as:

$$\pi^A(s,a) = \begin{cases} 0 & \text{if } R(s,a) \neq \emptyset \\ \pi(s,a) + \pi(s,a) * \eta(s) & \\ \qquad * \, deg(s,a) & \text{else if } deg(s,a) \leq 0 \\ \pi(s,a) + (1 - \pi(s,a)) & \\ \qquad * \, \eta(s) * deg(s,a) & \text{else if } deg(s,a) > 0 \end{cases}$$

where $deg(s, a)$ and $\eta(s)$ are defined as following.

The function $deg(s, a)$ determines the impact of suggestions. We define $deg(s, a) = max(\{g.d > 0 | g \in G(s, a)\}) + min(\{g.d < 0 | g \in G(s, a)\})$.[2] With this definition, a worker only considers the strongest suggestion, either positive or negative. This definition is also used to resolve conflicting suggestions (in a multi-level supervision organization) by summing the degrees of the strongest positive suggestion and the strongest negative suggestion.

The function $\eta(s)$ is state-dependent and ranges from $[0, 1]$. It determines the receptivity for suggestions and allows the agent to selectively accept suggestions based on its current state. For instance, if an agent becomes more confident in the effectiveness of its local policy on state s because it has more experience with it, then $\eta(s)$ decreases as learning progresses. For example, we set $\eta(s) = k/(k + visits(s))$ where k is a constant and $visits(s)$ returns the number of visits on the state s.

To normalize π^{AC} such that it sums to 1 for each state, the *limit* function from GIGA [13] is applied with minor modifications so that every action is explored with minimum probability ϵ:

$$\pi^{AC} = limit(\pi^{AC}) = argmin_{x:valid(x)} |\pi^{AC} - x|$$

i.e., $limit(\pi^{AC})$ returns a valid policy that is closest to π^{AC}.

We have tested our approach in a distributed task allocation problem. Experimental results show that our approach incorporated with some simple domain knowledge not only dramatically speeds up the convergence rate, but also increases the likelihood of convergence when an unsupervised MARL algorithm fails to converge. Due to the space limit, we describe our experiments in the technical report [12].

6. CONCLUSIONS

This work presents a scalable and robust framework that enables efficient learning in large-scale multi-agent systems. In our framework, the automated supervision mechanism fuses activity information of lower-level agents and generates supervisory information that guides and coordinates agents' learning process. This supervision mechanism continuously interacts with the learning process to accelerate the convergence.

7. REFERENCES

[1] S. Abdallah and V. Lesser. Learning the task allocation game. In *AAMAS'06*, 2006.

[2] S. Abdallah and V. Lesser. Multiagent reinforcement learning and self-organization in a network of agents. In *AAMAS'07*, 2007.

[3] B. Banerjee and J. Peng. Performance bounded reinforcement learning in strategic interactions. In *AAAI'04*, pages 2–7, 2004.

[1] We use "." as a projection operator. For example, $r.c$ returns the rule condition of rule r.

[2] If $G(s, a)$ is empty, then $deg(s, a) = 0$.

[4] R. A. C. Bianchi, C. H. C. Ribeiro, and A. H. R. Costa. Heuristic selection of actions in multiagent reinforcement learning. In *IJCAI'07*, Hyderabad, India, 2007.

[5] M. Bowling. Convergence and no-regret in multiagent learning. In *NIPS'05*, pages 209–216, 2005.

[6] J. A. Boyan and M. L. Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. In *NIPS'94*, volume 6, pages 671–678, 1994.

[7] R. Makar, S. Mahadevan, and M. Ghavamzadeh. Hierarchical multi-agent reinforcement learning. In *Autonomous Agents'01*, pages 246–253, 2001.

[8] A. Y. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: theory and application to reward shaping. In *ICML'99*, pages 278–287, 1999.

[9] S. P. Singh, T. Jaakkola, M. L. Littman, and C. Szepesvari. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38(3):287–308, 2000.

[10] P. Stone and M. Veloso. Team-partitioned, opaque-transition reinforcement learning. In *Autonomous Agents'99*, pages 206–212, 1999.

[11] P. Tangamchit, J. Dolan, and P. Khosla. Learning-based task allocation in decentralized multirobot systems. In *DARS'00*, pages 381–390, 2000.

[12] C. Zhang, S. Abdallah, and V. Lesser. Improving multi-agent learning through automated supervisory policy adaptation. In *University of Massachusetts Amherst Computer Science Technical Report #08-03*, 2008.

[13] M. Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *ICML'03*, pages 928–936, 2003.