A New Approach for Value Function Approximation Based on Automatic State Partition

Jiaan Zeng, *Yinghua Han †

Abstract—**Value function is usually used to deal with the reinforcement learning problems. In large or even continuous states, function approximation must be used to represent value function. Much of the current work carried out, however, has to design the structure of function approximation in advanced which cannot be adjusted during learning. In this paper, we propose a novel function approximation called Fuzzy CMAC (FCMAC) with automatic state partition (ASP-FCMAC) to automate the structure design for FCMAC. Based on CMAC (also known as tile coding), ASP-FCMAC employs fuzzy membership function to avoid the setting of parameter in CMAC, and makes use of Bellman error to partition the state automatically so as to generate the structure of FCMAC. Empirical results in both mountain car and RoboCup Keepaway domains demonstrate that ASP-FCMAC can automatically generate the structure of FCMAC and agent using it can learn efficiently.**

Keywords: reinforcement learning, fuzzy CMAC, automatic state partition

1 Introduction

In reinforcement learning (RL) problems, each agent needs to learn an optimal policy to maximize the long term reward. Most of RL algorithms seek such policy using value function [1]. In complex tasks with large or even continuous states, function approximation approaches must be used to represent value function. Much of the work currently carried out uses linear function approximators such as CMAC, also known as tile coding, or neural network [1, 2]. Owing to simple structures, they have already been used widely in many issues [1, 3, 4, 5].

However, the structure of most function approximators is specified before hand by human designer. The process of designing a proper structure can be difficult and time consuming [6]. In addition, poor structure design can result in agent's poor learning performance. Recently, many researchers have devoted to the topic of generating the structure of function approximators automatically [6, 7, 8, 9].

In this paper, we present a novel function approximation for RL, called Fuzzy CMAC (FCMAC) with automatic state partition (ASP-FCMAC) which automate the structure design of FCMAC. FCMAC, which is based on CMAC, employs fuzzy membership functions instead of binary function used in CMAC to reduce the number of learning weights so as to lower computation load and avoid the setting of tiling number in CMAC [10, 11]. To generate the structure of FCMAC automatically, ASP-FCMAC analyzes Bellman error and learning weights to partition the state during learning. At the very beginning, the partition of state is very coarse and Bellman error remains high. Later on, ASP-FCMAC can partition the state gradually and find good representation of the state so as to reduce Bellman error and generate the structure. Empirical results in two RL domains which are mountain car and RoboCup Keepaway show that ASP-FCMAC can generate the structure of FCMAC and learn optimal policy efficiently.

2 Background

2.1 Markov Decision Process

Generally speaking, a RL problem can be represented as a Markov Decision Process (MDP). A MDP can be represented as a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$ where $\mathcal{S}$ is a state, $\mathcal{A}$ is an action set, function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ defines the probability for agent to transfer from one state to another, and $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ function defines the reward agent gets after executing an action. The goal for each agent is to maximize the following value function:

$$V^{\pi}(s) = E_{\pi}[\sum_{i=0}^{\infty} \gamma^{i} \mathcal{R}(s_t, a_t) | s_0 = s] \quad (1)$$

where s_t and a_t represent the state and action in step t respectively, γ is the discount factor. The Bellman error which displays the error between current reward and

*Jiaan Zeng is a graduate student in School of Computer Science and Engineering, South China University of Technology, Guangzhou, 510640, China, Email: l.allen09@gmail.com

†Yinghua Han is a graduate student in School of Computer Science and Engineering, South China University of Technology, Guangzhou, 510640, China, Email: hattieq@163.com

estimated reward can be represented as follows:

$$\Delta V = \max_a[\mathcal{R}(s,a) + \gamma R(P(s,a))] - V(s) \quad (2)$$

In large or even continuous states, the value function must be represented by approximated function. Linear function approximation is one of the methods mostly used. With it, (1) can be represented as follows:

$$\widetilde{V} = \Phi\theta \quad (3)$$

where $\widetilde{V}$ is the approximated value of V, Φ is a $|S| \times m$ matrix, $|S|$ is the size of state, m is the number of features for each state, θ is a vector with m dimensions and $m \ll |S|$ usually.

2.2 CMAC and Fuzzy CMAC

CMAC (also called tiling coding) is a widely used function approximator [1, 2]. The state is covered by several layers called tiling. Each tiling is consisted of some elements called tiles and overlaps its adjacent tilings with some offset. During learning, the function value is computed as the summation from learning weights of active tiles. The value function using CMAC can be represented as follows:

$$\begin{aligned} \widetilde{V} &= \sum_{i=1}^{n_1} \varphi_i(s) w_i \\ \varphi_i(s) &= \begin{cases} 1, \text{if tile i is active} \\ 0, otherwise \end{cases} \end{aligned} \quad (4)$$

where w_i is the learning weight of the $i-th$ tile, $\varphi_i(s)$ is the feature indicating whether a tile is active or not, n_1 is the storage of CMAC. The update rule for CMAC is as follows:

$$w_i \leftarrow w_i + \alpha\varphi_i\Delta V \quad (5)$$

where α is the learning rate.

Actually, the feature of CMAC is a binary function that shows whether a tile is active or not. To attain generalization, CMAC needs to have a large amount of tilings which will lead to high load of computation. In addition, Sherstov et. al. demonstrate that there is no optimal setting of tiling number. It should be adjusted dynamically during learning [8]. Moreover, in different domains, CMAC has to set the tiling number differently and manually. J.Nie et al. use fuzzy membership function instead of binary function for feature in CMAC [10] and name it FCMAC, and Su et. al. discuss the learning ability of both CMAC and FCMAC [11]. The fuzzy membership function reflects the active degree of a certain feature rather than just shows it active or not. Therefore, there is no need to set the tiling number and FCMAC can avoid the disadvantages mentioned above. Using FCMAC, the function approximation can be represented as follows:

$$\widetilde{V} = \sum_{i=1}^{n_2} \eta_i(s) w_i \quad (6)$$

where $\eta_i(s)$ is a fuzzy membership function, n_2 is the storage of FCMAC. The update rule for FCMAC is:

$$w_i \leftarrow w_i + \alpha\eta_i\Delta V \quad (7)$$

3 Fuzzy CMAC with Automatic State Partition (ASP-FCMAC)

FCMAC does not need to set the tiling number so that it is used widely [12]. However, it still needs to partition the state so that similar states can have similar values. Its structure bases on how the state is partitioned which influence the performance greatly [12]. Usually, the state partition of FCMAC is done in advance and cannot be adjusted during learning. To solve this problem, we present automatic state partition for FCMAC. This algorithm starts without any partition on the state. Gradually, it automatically partitions the state by analyzing the learning weights and Bellman error. To partition a state is to generate a new learning weight. In other words, it generates the structure for FCMAC during learning. Automatic state partition needs to solve two main problems: one is the occasion for partition, and the other one is the region to partition. Next, we will discuss the details.

3.1 The Occasion for Partition

The occasion for partition is the problem whether to exploit or explore, just like the one in evolutionary neural network [9, 5]. Partitioning the state too frequently cannot improve the effort of approximation but rather results in poor performance. This is because each partition needs time to adjust the learning weights, and if the state is partitioned inappropriately, this change cannot be reversed. But partitioning the state too late will lead to slow learning rate since the update of learning weights bases on coarse partitions. Therefore how to handle the tradeoff between structural exploration and structural exploitation is very important. In this paper, we use the Bellman error to make a decision when to partition the state. Since Bellman error is the criterion to judge whether the learning algorithm converges or not, it can reflect current status of function approximation. When function V approaches the optimal one gradually, $|\Delta V(s)|$ declines. Once $|\Delta V(s)|$ stops declining, the learning has been in plateau. Therefore, we partition the state when Bellman error stops declining so as to keep algorithm learning. But in actual tests, the quantity of $|\Delta V(s)|$ is very noisy. If partitioning the state according to $|\Delta V(s)|$ of each step, it will have so many meaningless partitions. Thus we use an average method to judge whether Bellman error declines. First, we set a sample size T which stands for the number of steps, and calculate the average Bellman error in this sample:

$$\overline{|\Delta V(s)|} = \frac{\sum_{i=1}^{T} |\Delta V_i(s)|}{T} \quad (8)$$

where $|\Delta V_i(s)|$ is the error of each step. Next is the condition judging whether to partition:

$$\overline{|\Delta V(s)|} \quad > \quad \overline{|\Delta V(s)_{last}|} \tag{9}$$

where $\overline{|\Delta V(s)_{last}|}$ is the last average of $\overline{|\Delta V(s)|}$. If $\overline{|\Delta V(s)|}$ is not less than the last one, then the learning has entered plateau stage and it is necessary to partition the state. Especially, when T is infinite, it does not partition the state; when T is 1, it partitions the state at each step. Therefore, the value of T influences how frequently to partition the state. In our implementation, T is not a fixed value, but has a function relationship with the number of partitions.

$$T = f(p) \tag{10}$$

where p is the number of partitions and f is an increasing function. The finer the partition of the state is, the larger the value of T will be. The reason for that is the approximation can be very quick at the beginning due to few partitions, but latter it requires a relatively long time to adjust the learning weights since more partitions are made. At that time, it is necessary to keep current partition structure rather than change it.

3.2 The Region to Partition

When an agent decides to partition the state, it needs to choose which region to partition. Before presenting our approach to choose the region, we need to state two points clearly. Principally, it is unnecessary to partition state along each dimension at one time. In our tests, we partition state along each dimension at one time and this approach works well. Therefore, for simplicity, all dimensions will involve in each partition operation. Another point is that different partitions should have different length so as to reflect different degree of representation. Each time, we divide a certain region equally. After many times, some regions have more partitions to have a finer representation, while other regions have fewer partitions to have a coarser representation. This can definitely reflect the importance of different regions. Next, we present two types of methods to find which region to partition.

1. *Partition the region visited frequently*
We use the following expression for partition.

$$p = \max_a v(p_i) \tag{11}$$

where p is the region we want to partition, $v(p_i)$ is the account to visit region p_i. If a region is visited frequently, it, we believe, plays an important role in agent's policy. Therefore, it is desirable to refine the representation of this region.

2. *Partition the region with maximal summation of learning weights.*
We use the following expression for partition.

$$p = \max_i(w_i + w_{i+1}) \tag{12}$$

where w_i and w_{i+1} denote the learning weights of region . We pick up the region with maximal summation $(w_i + w_{i+1})$ to partition. In FCMAC, $|\Delta V(s)|$ can be represented as follows:

$$|\Delta V(s)| = R(s,a) + \sum_{i=1}^{n_2} \eta_i w_i - V(s) \tag{13}$$

Thus $(w_i + w_{i+1})$ is part of $|\Delta V(s)|$. Using (13) for choice can cause agent to focus on the region where value function change rapidly. That is because the lager $(w_i + w_{i+1})$ is, the larger $|\Delta V(s)|$, which changes the learning weights directly, will be. This is just like the value criterion used by Whiteson [7]. Like evolutionary neural network, for ASP-FCMAC, we adopt the idea of structure mutation [9, 5]. Each time, when an agent decides to partition the state, it will choose a region using either of the approaches stated above or randomly choose a region. The possibility to choose randomly is ϵ_m. After each partition, a new learning weight is generated. Typically, it is initialized as 0, so as to avoid affecting overall performance of FCMAC. We present the entire procedure of ASP-FCMAC in next section.

3.3 ASP-FCMAC

Algorithm 1 presents the details of ASP-FCMAC. Function *GetFeatureValue* returns the fuzzy value corresponding to current state. Function *ShouldPartition* carries out algorithm stated in 3.1. Function *Partition* makes use of the method stated in 3.2 to divide the state so as to generate the structure of FCMAC. First in line 1, initial operation is done including setting 0 for feature vector and learning weights. The learning procedure is from line 2 to 12. In line 3 algorithm gains current state, in line 4 it calculates the Bellman error, and in line 5 it computes the fuzzy value. Next in line 7, it updates the learning weights according to (7). In line 9 we decide whether to partition the state and do it in line 10.

Algorithm 1 ASP-FCMAC

```
1: Initialization;
2: for t = 1,... do
3:    Input current state s_t;
4:    ΔV = max_a[R(s,a) + γR(P(s,a))] − V(s);
5:    η ← GetFeatureValue(s_t);
6:    for i = 1 to n_2 do
7:       w_i ← w_i + αφ_iΔV;
8:    end for
9:    if ShouldPartition(|ΔV(s)|) then
10:      Partition(s);
11:   end if
12: end for
```

4 Results and Discussion

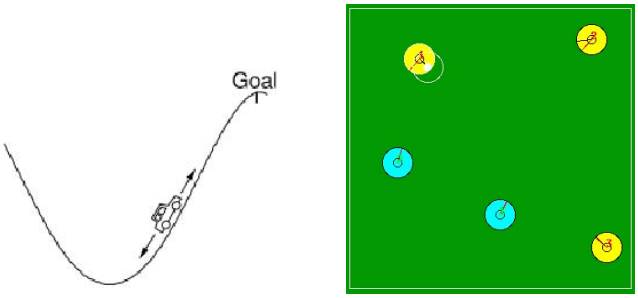
ASP-FCMAC is applied into two RL domains. One is mountain car. Since its state has only two dimensions, we can compare the policy of different learning algorithms deeply. The other one is RoboCup Keepaway. It is a complex multiagent system (MAS) with noise where the effort of ASP-FCMAC can be further verified.

4.1 Results of Mountain Car

In the mountain car (MCAR) domain [1], a car initially stays at the bottom of a "U" shape mountain and starts to climb up one side of the mountain. Since its engine is not powerful enough, the car cannot climb up only through driving forward. Instead, the car needs to drive backward to another side and uses the energy generated by going down. The main evaluation criterion is the number of steps needed for the car to climb up the mountain. The less the number is, the better the policy is. Figure 1(a) depicts the scene of MCAR.

Four learning algorithms, including FCMAC, ASP-FCMAC with partition on region visited most frequently (ASP-FCMAC-Visit), ASP-FCMAC with partition on region having maximal summation of learning weights (ASP-FCMAC-Max), and CMAC, are tested. The parameters of RL in MCAR are $\alpha = 0.25, \epsilon = 0.01, \lambda = 1, \gamma = 0.9$. For simplicity, triangular functions are used here. The settings for ASP-FCMAC are $\beta = 0.1, T = 10p, \epsilon_m = 0.01$. For CMAC, the tiling number is set as 10; tile number in each tiling is set as 8 General speaking, we have infinite choices how the state is partitioned in CMAC. After several trials, we decide on these setting since they perform well in Mountain Car. For FCMAC, the state is divided into 8 partitions. For consistency and comparison, we divide the state for FCMAC just as CMAC did. Sarsa(λ) is used to calculate value function for all learning algorithms. Each learning algorithm is run 5000 episodes and all of the curves plotted in Figure 2 and 3 are average of 8 different runs.

Figure 2 displays the number of steps needed for a successful achievement. The x-axis is the episode number and the y-axis is the steps to goal. The results of these four algorithms vary clearly. The steps of FCMAC and ASP-FCMAC-Visit are about 155, but the ones of CMAC and ASP-FCMAC-Max are nearly 120. This strongly proves that ASP-FCMAC definitely can generate the structure and approximate value function with automatic state partition. CMAC learns fastest among all the four algorithms and has a flatten curve. FCMAC fluctuates during learning. This is consistent to the conclusion of Tokarchuck: although FCMAC stores fewer learning weights than CMAC does, its generalization is worst than CMAC [12]. Moreover, ASP-FCMAC-Max can get the result just as CMAC has. Thus we believe the generalization ability can be raised by improving the partition of



(a) Mountain Car [1]. (b) 3vs.2 Keepaway in progress.

Figure 1: Test bed domains.

state, namely the structure of FCMAC. According to the steps achieved, we believe ASP-FCMAC-Max represents the state more efficient than ASP-FCMAC-Visit does.

To see whether the approach of automatic state partition has an impact on the convergence of FCMAC, we plot average Bellman error of FCMAC, ASP-FCMAC-Max, and ASP-FCMAC-Visit in Figure 3. We can see that all of these algorithms converge at the end of learning. Therefore, automatic state partition does not influence the convergence of FCMAC. Besides, the error of both FCMAC and ASP-FCMAC-Max declines faster than ASP-FCMAC-Visit. This further verifies that ASP-FCMAC-Max is superior to ASP-FCMAC-Visit since it can reduce Bellman error quickly.

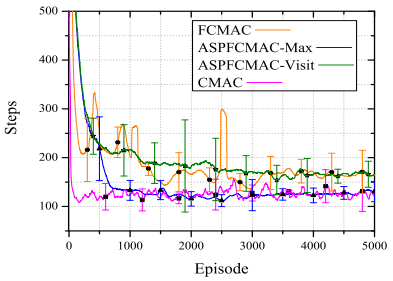


Figure 2: Steps to goal.

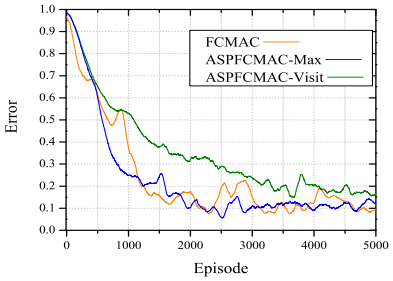
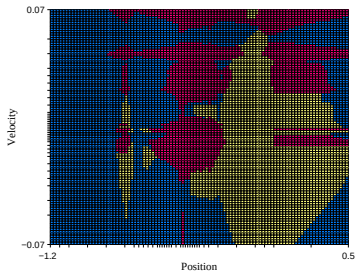
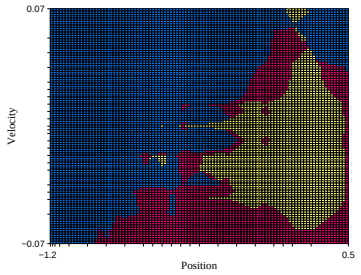


Figure 3: Average Bellman error.



(a) ASP-FCMAC-Max



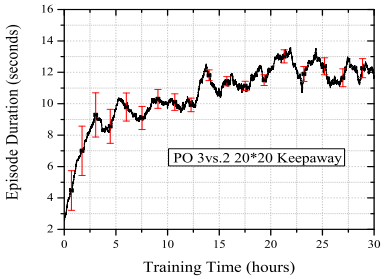
(b) ASP-FCMAC-Visit

Figure 4: Policy space of different algorithms.

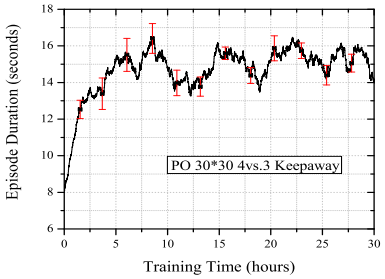
To investigate how the state is partitioned finally and compare the policy space of different algorithms, we plot the final policy for ASP-FCMAC-Max and ASP-FCMAC-Visit in Figure 4. The x-axis is the position and the y-axis is the velocity. The ticks in both x-axis and y-axis represent the final partition structure. ASP-FCMAC-Max has a fine representation near the center of state, while ASP-FCMAC-Visit seems to divide the state equally. It is reasonable for ASP-FCMAC to devote finer representation in the area near the center of state than other regions. That is because the car usually has to select an action from forward, backward or stop in the bottom of the mountain. Refining representation of this area can help the car to get a better policy. But in the areas adjacent to the top of both sides, there are less choice for the car to take action and there is no need to have a fine representation on these areas. Considering the learning results of Figure 2, we can safely conclude that it is smarter for ASP-FCMAC-Max to find the important region in state than ASP-FCMAC-Visit does.

4.2 Results of Keepaway

Keepaway is a sub-problem of RoboCup Soccer Simulation (RCSS) [3, 4]. Two teams are played in a fixed rectangular region. A team of keepers tries to keep the ball away from the opposing team of takers as much as possible. Usually, there are x keepers and y takers ($x > y$). The goal of keepaway is to make keeper to keep the ball as long as possible. The main evaluation criterion is the average episode duration. The longer it is, the better policy agent has. Figure 1(b) shows 3vs.2 keepaway in progress.



(a) 3vs.2 Keepaway in 20×20 field



(b) 4vs.3 Keepaway in 30×30 field

Figure 5: PO Keepaway.

The parameters of RL in keepaway are $\alpha = 0.125, \epsilon = 0.01, \lambda = 1, \gamma = 0.65$. As it is stated in 4.1, we use triangular function as fuzzy membership function. The settings for ASP-FCMAC are $\beta = 0.1, T = 10p, \epsilon_m = 0.01$. Keeper uses Sarsa($\lambda$) to calculate value function, as well as ASP-FCMAC-Max for function approximation. All players including keeper and taker have 360° vision. Experiments are carried out in partially observable (PO) keepaway which is the standard version of keepaway. Both sense information and action information in PO keepaway have noises. All the learning curves are average of 4 different runs.

Figure 5(a) plots the results of PO 3vs.2 Keepaway in 20×20 field. Average episode duration is 12.2s, standard variation is 0.2s. That is consistent with the results using CMAC [3] and hence verifies the effort of ASP-FCMAC in large scale, noisy problem. To further study the learning ability of ASP-FCMAC, we perform PO 4vs.3 Keepaway in 30×30 field. Since more players are added, ASP-FCMAC needs to handle larger state (There are 13 state variables in 3vs.2, 19 state variables in 4vs.3). Figure 5(b) shows the result. Average episode duration is 14.1s, standard variation is 0.3s. That is also similar to the result using CMAC [3]. Therefore, we can safely conclude that ASP-FCMAC performs the same as CMAC does in PO Keepaway. But ASP-FCMAC is superior to CMAC with less storage, generating structure automatically.

5 Conclusion and Future Work

This paper presents a novel approach for value function approximation — ASP-FCMAC. We discuss two main parts of it: the occasion for partition and the region to partition deeply. ASP-FCMAC is applied into two RL domains. In mountain car, we compare FCMAC, ASP-FCMAC, and CMAC on different aspects including steps to goal, Bellman error, and policy space. Empirical results demonstrate that the learning ability of FCMAC can be improved by finding a better structure and ASP-FCMAC can efficiently generate the structure for FCMAC. Besides we find that partitioning on the region having maximal summation of learning weights is better than on region visited frequently. In RoboCup Keepaway, the results show that ASP-FCMAC can learn as efficiently as CMAC in continuous state with noise. But ASP-FCMAC is better than CMAC in regard to less storage, generating structure automatically. Next, we will focus on the algorithm finding a better occasion for partition and extend ASP-FCMAC to different versions of keepaway such as 5vs.4 keepaway.

References

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

[2] J. C. Santamaria, R. S. Sutton, and A. Ram, "Experiments with reinforcement learning in problems with continuous state and action spaces," tech. rep., Amherst, MA, USA, 1996.

[3] P. Stone, R. S. Sutton, and G. Kuhlmann, "Reinforcement learning for RoboCup-soccer keepaway," *Adaptive Behavior*, vol. 13, no. 3, pp. 165–188, 2005.

[4] P. Stone, G. Kuhlmann, M. E. Taylor, and Y. Liu, "Keepaway soccer: From machine learning testbed to benchmark," in *RoboCup-2005: Robot Soccer World Cup IX* (I. Noda, A. Jacoff, A. Bredenfeld, and Y. Takahashi, eds.), vol. 4020, (Berlin), pp. 93–105, Springer Verlag, 2006.

[5] S. Whiteson and P. Stone, "Evolutionary function approximation for reinforcement learning," *Journal of Machine Learning Research*, vol. 7, pp. 877–917, May 2006.

[6] P. W. Keller, S. Mannor, and D. Precup, "Automatic basis function construction for approximate dynamic programming and reinforcement learning," in *Proceedings of the 23rd International Conference on Machine Learning*, 2006.

[7] S. Whiteson, M. E. Taylor, and P. Stone, "Adaptive tile coding for value function approximation," Tech. Rep. AI-TR-07-339, University of Texas at Austin, 2007.

[8] A. A. Sherstov and P. Stone, "Function approximation via tile coding: Automating parameter choice," in *SARA 2005* (J.-D. Zucker and I. Saitta, eds.), vol. 3607 of *Lecture Notes in Artificial Intelligence*, (Berlin), pp. 194–205, Springer Verlag, 2005.

[9] J. H. Metzen, M. Edgington, Y. Kassahun, and F. Kirchner, "Analysis of an evolutionary reinforcement learning method in a multiagent domain," in *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, (Richland, SC), pp. 291–298, International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[10] J. Nie and D. A. Linkens, "Fcmac: a fuzzified cerebellar model articulation controller with self-organizing capacity," *Automatica*, vol. 30, no. 4, pp. 655–664, 1994.

[11] S. Su, Z. Lee, and Y. Wang, "Robust and fast learning for fuzzy cerebellar model articulation controllers," *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics*, vol. 36, no. 1, pp. 203 – 208, 2006.

[12] L. Tokarchuk, "Fuzzy and tile coding approximation techniques for coevolution in reinforcement learning," tech. rep., University of London, 2006.

[13] P. L. Lanzi, D. Loiacono, S. W. Wilson, and D. E. Goldberg, "Classifier prediction based on tile coding," in *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, (New York, NY, USA), pp. 1497–1504, ACM, 2006.

[14] S. Mahadevan, "Samuel meets amarel: Automating value function approximation using global state space analysis," in *Proceedings of AAAI*, pp. 1000–1005, 2005.

[15] I. Menache, S. Mannor, and N. Shimkin, "Basis function adaptation in temporal difference reinforcement learning," *Annals of Operations Research*, vol. 134, pp. 215–238, 2005.