XCS with Computed Prediction in Multistep Environments

Pier Luca Lanzi[*][†], Daniele Loiacono[*], Stewart W. Wilson[†][‡], David E. Goldberg[†]
[*]Artificial Intelligence and Robotics Laboratory (AIRLab)
Politecnico di Milano. P.za L. da Vinci 32, I-20133, Milano, Italy
[†]Illinois Genetic Algorithm Laboratory (IlliGAL)
University of Illinois at Urbana Champaign, Urbana, IL 61801, USA
[‡]Prediction Dynamics, Concord, MA 01742, USA
lanzi@elet.polimi.it, loiacono@elet.polimi.it, wilson@prediction-dynamics.com, deg@illigal.ge.uiuc.edu

ABSTRACT

XCSF extends the typical concept of learning classifier systems through the introduction of computed classifier prediction. Initial results show that XCSF's computed prediction can be used to evolve accurate piecewise linear approximations of simple functions. In this paper, we take XCSF one step further and apply it to typical reinforcement learning problems involving delayed rewards. In essence, we use XCSF as a method of generalized (linear) reinforcement learning to evolve piecewise linear approximations of the payoff surfaces of typical multistep problems. Our results show that XCSF can easily evolve optimal and near optimal solutions for problems introduced in the literature to test linear reinforcement learning methods.

Categories and Subject Descriptors

F.1.1 [**Models of Computation**]: Genetics Based Machine Learning, Learning Classifier Systems; I.2.6 [**Learning**]: Reinforcement Learning, Generalization

General Terms

Algorithms, Performance

Keywords

LCS, XCS, RL, Generalization

1. INTRODUCTION

Reinforcement Learning [6] deals with the problem of learning by interacting with an unknown environment which returns feedback in term of numerical reward. Solutions to reinforcement learning problems are represented by action-value functions which map state-action pairs to the expected payoff: state-action pairs identify the current situation (the state) and the possible system behavior (the action); the expected payoff estimates the amount of future reward that the system should expect. Most reinforcement learning algorithms assume that action-value functions are represented as look-up tables containing one entry for each possible state-action pair. However, look-up tables are infeasible when it comes to applications involving many states and many actions. Accordingly, when applying reinforcement learning to large problems approximation techniques are used and look-up tables are replaced by parametrized functions. Unfortunately, when moving to approximated approaches, the nice convergence properties of (exact) tabular algorithms do not hold anymore [1, 6]. Most important, results reported in the reinforcement learning literature suggest that function approximation does not always work well [1, 4], although some exceptional results have been obtained [7].

Learning classifier systems are a different approach to the same issue–the complexity of solutions represented as look-up tables. Learning classifier systems represent an action-value function with a set of condition-action-prediction rules, the classifiers. Each classifier associates a (constant) payoff value (the classifier prediction) to the set of states (matched by the condition) and to one action. Thus, from a reinforcement learning perspective, learning classifier systems evolve a *piecewise constant* approximation of the action-value function which represent the problem solution.

XCSF extends the typical concept of learning classifier systems through computed classifier prediction. In XCSF classifier prediction is not a parameter but it is computed as a linear combination of the current input and a weight vector associated to each classifier. Originally, XCSF was conceived as a pure function approximator [12]: classifiers did not have an action and computed prediction was used to produce piecewise linear approximations of target functions. Wilson [12] applied XCSF to simple function approximation problems showing that computed prediction can be used to evolve accurate piecewise linear approximation of the target functions. Recently, Wilson applied XCSF with discrete actions, dubbed XCS-LP, to a simple single step problem (the frog problem [13]) involving payoff functions continuous with respect to the problem space. The results reported in [13] show that XCSF with actions and linear prediction (XCS-LP) exhibits high performance and low error, as well as dramatically smaller solutions compared with XCS.

In this paper we take one step further and apply XCSF to problems involving delayed rewards to evolve piecewise linear approximation of payoff surfaces typical of multistep reinforcement learning problems. In this respect, we actually use XCSF as a method of (linear) generalized reinforcement learning [1, 6]. We apply XCSF to a set of problems inspired by those used in the reinforcement learning literature for linear approximators [1]. The results we present here show that, in similar problems, XCSF can easily reach optimal or near optimal performance in all the problems considered, suggesting that XCSF might be a very promising approach to tackle complex multistep problems providing solutions that are typical of generalized (linear) reinforcement learning methods. Note that, in this paper, we will simply refer to XCS with computed prediction as XCSF to abstract the concept of computed prediction (first introduced with XCSF [12]) from the more specific implementation with actions and linear prediction (XCS-LP in [13]).

2. REINFORCEMENT LEARNING

Reinforcement learning is defined as the problem of an *agent* that learns to perform a task through *trial and error interactions* with an unknown *environment* which provides feedback in terms of numerical *reward* [6]. The agent and the environment interact continually. At time t the agent senses the environment to be in state s_t; based on its current sensory input s_t the agent selects an action a_t in the set A of the possible actions; then action a_t is performed in the environment. Depending on the state s_t, on the action a_t performed, and on the effect of a_t in the environment, the agent receives a *scalar reward* r_{t+1} and a new state s_{t+1}. The agent's goal is to *maximize* the amount of reward it receives from the environment *in the long run*, or *expected payoff* [6].

2.1 Generalized Reinforcement Learning

In reinforcement learning the agent learns how to maximize the incoming reward by developing an action-value function $Q(\cdot,\cdot)$ (or a state value function $V(\cdot)$) that maps state-action pairs (or states) into the corresponding expected payoff value. Reinforcement learning methods assume that action-value functions (and value functions) are represented by *look-up tables* with one entry for each state-action pair (or one entry for each state in the case of value functions). However, look-up tables easily become infeasible in problems involving many states. Large look-up tables require more memory but, most important, they require more on-line experience to converge. To cope with the complexity of large problems the agent must be able to *generalize* over its experiences, i.e., to produce a good approximation of the optimal value function from a limited number of experiences, using a small amount of storage.

In reinforcement learning generalization is implemented by methods of function approximation techniques: the action-value function is not represented as a table but as a function parametrized with a vector $\boldsymbol{\theta}$. This means that at time step t, the value associated to a particular state-action pair (or to a particular state) depends on the current parameter vector $\boldsymbol{\theta}_\mathbf{t}$. The action-value function $Q(\cdot,\cdot)$ is viewed as a function parametrized by a vector $\boldsymbol{\theta}$ that maps state-action pairs into real numbers (i.e., the expected payoff).

2.2 Gradient-Descent Methods

These are the most widely used function approximation methods. In gradient-descent methods, the parameter vector has a fixed number of real-valued components, $\boldsymbol{\theta}_\mathbf{t} = \langle\theta_t(1)\ldots\theta_t(n)\rangle$, while the target action-value function is approximated by a smooth differentiable function of $\boldsymbol{\theta}_\mathbf{t}$ for all the possible state-action pairs. At time step t, parameters $\boldsymbol{\theta}_\mathbf{t}$ are adjusted to minimize the mean-squared error (MSE) between the new estimate of the action-value function and the previous estimate. Gradient descent methods do this by adjusting the parameter vector by a small amount in the direction that would reduce the error on that example.

Among gradient-descent approaches, linear methods represent probably the most important case in reinforcement learning [1, 5, 6]. With linear methods, value functions are represented by linear functions of the vector parameter $\boldsymbol{\theta}_\mathbf{t}$. For any state s, a vector of n features, $\boldsymbol{\phi}_s = \langle\phi_s(1),\ldots,\phi_s(n)\rangle$ is extracted, the approximated value function for s is simply computed as $\boldsymbol{\theta}_\mathbf{t}\boldsymbol{\phi}_s$, while the gradient simply corresponds to the vector $\boldsymbol{\phi}_\mathbf{t}$. Linear methods are very computational efficient in terms both of space and time; on the other hand, their effectiveness rely heavily on the choice of the feature vector $\boldsymbol{\phi}_\mathbf{s}$, in addition, they are limited in that they cannot express interactions between features. To improve linear methods, a number of approaches have been developed in which linear approximation is enriched with advanced representation of state features to allow the expression of relation between input features, e.g., coarse coding, tile coding, radial basis functions, and kanerva coding [6].

2.3 Convergence

The convergence of generalized reinforcement learning is a complex issue, still poorly understood (see [4] for a recent discussion). Most of the approximated reinforcement learning algorithms are not known to converge and there is no known way to extend the convergence proofs for tabular reinforcement learning to the case of function approximators. Even if function approximators proved successful in solving challenging reinforcement learning tasks [7], yet they have been shown to be generally unstable, even in simple problems [1]. For instance, [8] suggests (in the case of Q-learning [6]), that the approximators induce noise on the action-value function so that the system can overestimate the expected payoff even when noise has zero mean.

3. THE XCSF CLASSIFIER SYSTEM

XCSF extends XCS in three respects [12]: (i) classifier conditions are extended for numerical inputs, as done for XCSI [11]; (ii) classifiers are extended with a weight vector $\mathbf{w}$, that is used to compute classifier's prediction; finally, (iii) the classifier weight vector $\mathbf{w}$ are updated instead of the classifier prediction.

Classifiers. In XCSF, classifiers consist of a condition, an action, and four main parameters. The condition specifies which input states the classifier matches; it is represented by a concatenation of interval predicates, $int_i = (l_i, u_i)$, where l_i ("lower") and u_i ("upper") are integers, though they might be also real. The action specifies the action for which the payoff is predicted. The four parameters are: the weight vector $\mathbf{w}$, used to compute the classifier prediction as a function of the current input; the prediction error ε, that estimates the error affecting classifier prediction; the fitness F that estimates the accuracy of the classifier prediction; the numerosity *num*, a counter used to represent different copies of the same classifier. The weight vector $\mathbf{w}$ has one weight w_i for each possible input, and an additional weight w_0 corresponding to a constant input x_0, that is set as a parameter of XCSF.

Performance Component. XCSF works as XCS. At each time step t, XCSF builds a *match set* [M] containing the classifiers in the population [P] whose condition matches the current sensory input s_t; if [M] contains less than θ_{mna} actions, *covering* takes place as in XCSI [11, 12]. The weight vector $\mathbf{w}$ of covering classifiers is initialized with zero values (note that in [12], the weight vector is initialized with random values in [-1,1]); all the other parameters are initialized as in XCS [2].

For each action a_i in [M], XCSF computes the *system prediction*. As in XCS, in XCSF the *system prediction* of action a is computed by the fitness-weighted average of all matching classifiers that specify action a. In contrast with XCS, in XCSF classifier prediction is computed as a function of the current state s_t and the classifier vector weight w. Accordingly, in XCSF system prediction is a function of both the current state s and the action a. Following a notation similar to that in [2], the system prediction for action a in state s_t, $P(s_t, a)$, is defined as:

$$P(s_t,a) = \frac{\sum_{cl\in[M]|_a} cl.p(s_t) \times cl.F}{\sum_{cl\in[M]|_a} cl.F} \quad (1)$$

where cl is a classifier, $[\mathrm{M}]|_a$ represents the subset of classifiers in [M] with action a, $cl.F$ is the fitness of cl; $cl.p(s_t)$ is the prediction of cl in state s_t, which is computed as:

$$cl.p(s_t) = cl.w_0 \times x_0 + \sum_{i>0} cl.w_i \times s_t(i)$$

where $cl.w_i$ is the weight w_i of cl. The values of $P(s_t, a)$ form the *prediction array*. Next, XCSF selects an action to perform. The classifiers in [M] that advocate the selected action are put in the current *action set* [A]; the selected action is sent to the environment and a reward r is returned to the system together with the next input state s_{t+1}

Reinforcement Component. XCSF uses the incoming reward to update the parameters of classifiers in action set $[\mathrm{A}]_{-1}$ corresponding to the previous time step. Note that, when XCSF is used for function approximation (a single step problem) the reinforcement component acts on the current action set. At time step t, the expected payoff P is computed as:

$$P = r_{-1} + \gamma \max_{a\in A} P(s_t, a) \quad (2)$$

where r_{-1} is the reward received at the previous time step. The expected payoff P is used to update the weight vector $\mathbf{w}$ of the classifier in $[\mathrm{A}]_{-1}$ using a *modified delta rule* [9]. For each classifier $cl \in [A]_{-1}$, each weight $cl.w_i$ is adjusted by a quantity Δw_i computed as:

$$\Delta w_i = \frac{\eta}{|s_{t-1}(i)|^2}(P - cl.p(s_{t-1}))s_{t-1}(i) \quad (3)$$

where η is the correction rate and $|s_{t-1}|^2$ is the norm the input vector s_{t-1} [12]. The values Δw_i are used to update the weights of classifier cl as:

$$cl.w_i \leftarrow cl.w_i + \Delta w_i \quad (4)$$

Then the prediction error ε is updated as:

$$cl.\varepsilon \leftarrow cl.\varepsilon + \beta(|P - cl.p(s_{t-1})| - cl.\varepsilon)$$

Finally, classifier fitness is updated as in XCS [2].

Discovery Component. The genetic algorithm in XCSF works as in XCSI [11]. On a regular basis depending on the parameter θ_{ga}, the genetic algorithm is applied to classifiers in [A]. It selects two classifiers with probability *proportional to their fitness*, copies them, and with probability χ performs crossover on the copies; then, with probability μ it mutates each allele. Crossover and mutation work as in XCSI [11, 12]. The resulting offspring are inserted into the population and two classifiers are deleted to keep the population size constant.

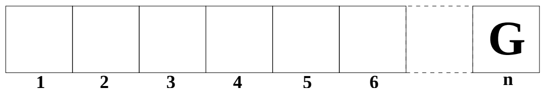


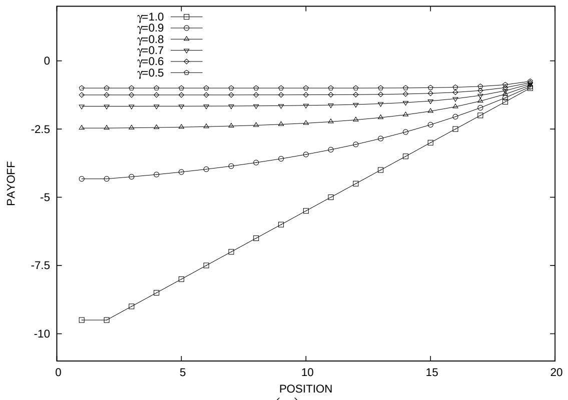
Figure 1: The `Corr(`n`)` environment. The symbol "G" denotes the goal position.

4. DESIGN OF EXPERIMENTS

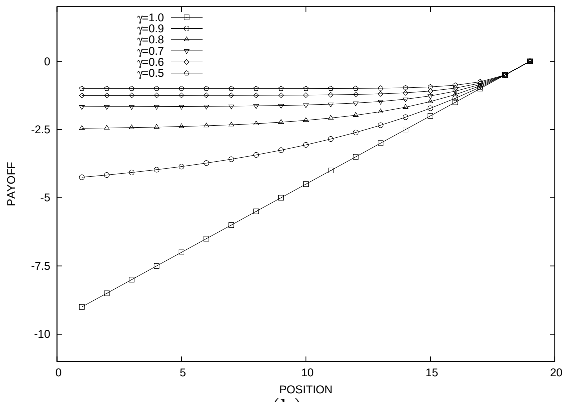
To apply XCSF to multistep problems, we follow the standard experimental design used in the literature [10]. Each experiment consists of a number of problems that the system must solve. Each problem is either a *learning* problem or a *test* problem. In *learning* problems, the system selects actions randomly from those represented in the match set. In *test* problems, the system always selects the action with highest prediction. The genetic algorithm is enabled only during *learning* problems, and it is turned off during *test* problems. The covering operator is always enabled, but operates only if needed. Learning problems and test problems alternate. The reward policy we use is the one used in the reinforcement learning literature when studying linear approximators [1]. When XCSF solves the problem correctly, reaching the goal position, it receives a constant reward equal to 0; otherwise it receives a constant reward equal to -0.5. The performance is computed as the average number of steps needed to reach goal or food positions during the last 100 test problems. To speed up the experiments, problems can last at most 500 steps; when this limit is reached the problem stops even if the system did not reach the goal. All the statistics reported in this paper are averaged over 20 experiments.

5. THE LINEAR CORRIDOR

We begin with the very simple environment depicted in Figure 1. `Corr(`n`)` is a linear corridor with n positions labeled from 1 to n; the only goal is in position n; the system input is the integer associated to the current position; there are two possible actions: *left*, coded with 0, and *right*, coded with 1; the system can start in any empty position; when the system reaches position n (the goal), it receives zero reward, in all the other cases it receives -0.5. Figure 2 reports the optimal value functions in `Corr(20)` for the two actions



(a)



(b)

Figure 2: Optimal action-value functions for `Corr(20)` for the action (a) *left* and (b) *right*.

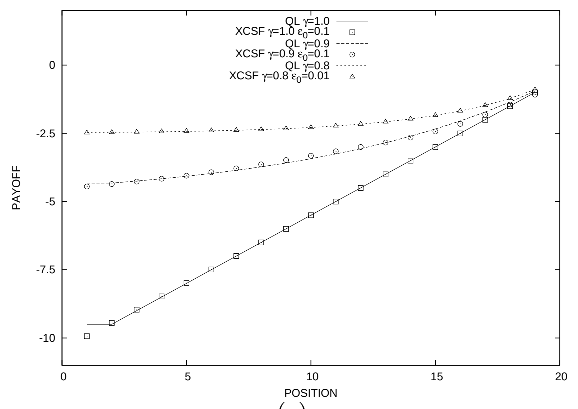
and different values of γ; no value is associated to the final position since when the agent reaches final position the problem ends and no action is available. As can be noted from the plots (Figure 2), when $\gamma = 1$ the state-values (identified by the square dots) are placed on a straight line and in each state there is a rough distinction between the values of the two actions. When γ is near to one (e.g., $\gamma = 0.9$), the state-values are placed on a smooth curve and still in each state there is a rough distinction between the values of the two actions. When γ is small (e.g., $\gamma = 0.7$), almost all the state-values stay on a flat line since there is almost no distinction between the values of the different action in the same positions For instance, when $\gamma = 0.7$, in position 1, the difference between the values of action *left* and action *right* is around 10^{-4}, three order of magnitude less than the immediate reward (0.5). This is a general issue for function approximators in multistep problems like XCSF. If γ is small we need accurate approximations, that is small values of the error threshold ϵ_0, to distinguish actions far from the goal; on the other hand, such small values of ϵ_0 generally limit generalization capabilities near the goal position where generalizations are possible only allowing higher error thresholds. Accordingly, most of the literature on function approximators for multistep problems use values of γ between 0.9 and 1.0, e.g., [1, 5].

We apply XCSF in `Corr(20)` with the following parameters setting (see [2] for details): $N = 400$, $\beta = 0.2$; $\alpha = 0.1$; $\nu = 5$; $\chi = 0.8$, $\mu = 0.04$, $p_{\text{explr}} = 0.5$, $\theta_{del} = 50$, $\theta_{GA} = 50$, and $\delta = 0.1$; GA-subsumption is on with $\theta_{sub} = 50$; while action-set subsumption is off; the parameters for integer conditions are $m_0 = 5$, $r_0 = 5$ [11]; the parameter x_0 for XCSF is 10, η is 0.2 [12]. We tested XCSF in `Corr(20)` on three different configurations of γ and ϵ_0: $\gamma = 1.0$, $\epsilon_0 = 0.1$; $\gamma = 0.9$, $\epsilon_0 = 0.1$; $\gamma = 0.8$, $\epsilon_0 = 0.01$; Figure 3a and Figure 3b compare the optimal action-value functions (reported as lines) with the solutions evolved by XCSF (reported as dots) for action *left* (Figure 3a) and action *right* (Figure 3b). In all the three cases, XCSF reaches optimal performance; the solutions evolved (reported with dots in Figure 3a and Figure 3b) represent the optimal action-value functions accurately. In particular, when $\gamma = 0.9$, the ϵ_0 is small enough to guarantee XCSF's optimal performance but not to guarantee a "*perfect fit*". A similar case shows up for action *left* with $\gamma = 1.0$ in position 1 (square dots in Figure 3a). Here XCSF approximation is rather far from the optimal action-value function, although the overall policy is optimal for the problem. This is easily explained by noting that the error threshold ϵ_0 represents an *average error* over all the states matched by each condition. When $\gamma = 1.0$ XCSF tends to generate highly general classifiers (matching most of the states) that are very accurate in the oblique part of the action value function, but quite inaccurate in position 1; however, on the average error of such classifiers is below the ϵ_0 threshold. When considering the number of macroclassifiers in the population we note that the final solutions are very compact, ranging from an average of the 3.3% of N (16 classifiers) when $\gamma = 1.0$, to an average of the 7% of N (28 classifiers) when $\gamma = 0.8$. As we should expect, smaller values of ϵ_0 correspond to more specific and therefore larger solutions.

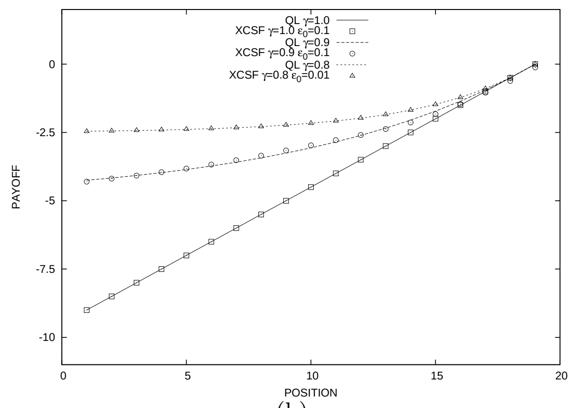
We extend the previous experiments and apply XCSF to `Corr(40)`; we use the same parameter setting and two values of γ and ϵ_0: $\gamma = 1.0$ and $\epsilon_0 = 0.1$; $\gamma = 0.9$ and $\epsilon_0 = 0.01$. In both cases the solutions evolved (not shown here) approximate the optimal action-value functions accurately. Figure 4a reports XCSF's performance while Figure 4b reports the number of macroclassifiers in the population. The results confirms those for `Corr(20)`: in both cases, XCSF reaches optimal performance (Figure 4c) and the population size is still rather compact, the 5% for $\gamma = 1.0$, the 10% when $\gamma = 0.9$.

6. THE EMPTY ROOM

In the second set of experiments we apply XCSF to the `Room(n)` environment (Figure 5), originally introduced in [1] where it has been generically dubbed *2D gridworld*. `Room(n)` is an $n \times n$ grid, with a goal in position $\langle n, n\rangle$; the system can start in any empty position; the system input consists of a pair of integers representing the current system position in the grid; there are four possible actions (up, down, left, and right) coded with two bits; as in `Corr(n)`, when the system reaches the goal (position $\langle n, n\rangle$) it receives a zero reward, otherwise it receives -0.5. Figure 6 shows the optimal value function for `Room(20)` when $\gamma = 0.9$. It is worth noting that while `Room(n)` is rather simple, it does not allow many generalizations for other models of learning classifier systems like XCSI, which for this problem would evolve very specific solutions mainly consisting of four clas-

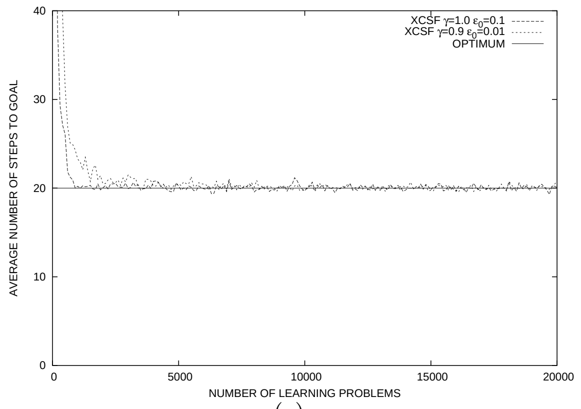


(a)

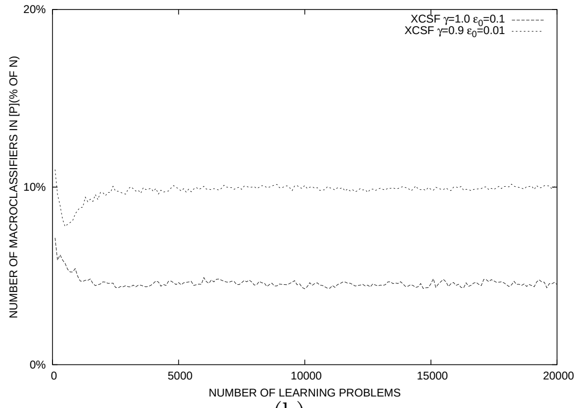


(b)

Figure 3: XCSF in Corr(20) for the following setting of γ and ϵ_0: $\gamma = 1.0$ and $\epsilon_0 = 0.1$; $\gamma = 0.9$ and $\epsilon_0 = 0.1$; $\gamma = 0.8$ and $\epsilon_0 = 0.01$. (a) action-value function for action left; (b) action-value function for action right. Statistics are averages over 20 runs.



(a)



(b)

Figure 4: XCSF in Corr(40) when $\gamma = 1.0$ and $\gamma = 0.9$: (a) performance; (b) number of macroclassifiers in the population ($/N$). Curves are averages over 20 runs.

sifiers for each position (result now shown here). Figure 6 shows even clearer what we already note in the Corr(20) environments: as the number of steps increases, even if a large value of γ is considered, there is a dramatic difference between the values of positions in areas far from the goal and positions in areas near to the goal. This has a serious influence on the generalization capability of XCSF approaches: the error threshold ϵ_0 is the same on all the problem space; to distinguish the values of action in areas far from the goal, we need small error thresholds; however, such small error thresholds prevent the evolution of large generalizations in areas near to the goal where the values of actions are largely different. For this reason, as we suggest in the concluding section, we believe that it would be interesting to extend XCSF approaches with adaptive error thresholds for classifiers, as done in [3].

First, we apply XCSF to Room(10) with $\gamma = 0.9$, $N = 2500$, and $\epsilon_0 = 0.05$; the parameters for integer conditions are $m_0 = 10$, $r_0 = 5$; the parameter x_0 for XCSF is 10, η is 0.2 [12]. Figure 7a reports XCSF's performance in Room(10) which is optimal. The evolved solutions are also compact containing an average of 80 macroclassifiers (the 3.3% of N). Note that XCSI in this case would generate populations of around 400 classifiers (not shown). Then, we apply XCSF to Room(20) which noteworthy is to our knowledge the largest *multistep* problem used for XCS models so far; in fact, Room(20) is much larger than other Markov grid environments tried, consisting of 399 distinct states, i.e., 1596 distinct state-action pairs. Figure 8 reports XCSF's performance in Room(20) for different values of N. For $N = 2500$ and $N = 5000$, XCSF's performance is very near to the optimum and becomes fully optimal when $N = 7500$. With respect to the size of the evolved solutions, when $N = 2500$ the final populations consist on the average of 315 macroclassifiers (the 12.6% of N), when $N = 5000$ the final populations consist on the average of 475 macroclassifiers (the 9.5% of N), when $N = 7500$ the final populations consist on the average of 563 macroclassifiers (the 7.5% of N). These results (Figure 8) evidence a typical behavior of XCSF which in our opinion represent an important improvement with respect to the performance of other XCS models. XCSF appears to be quite robust. Even if the population size is dramatically reduced or the error threshold raised, XCSF's performance, in the experiments we performed so far, did not show sudden breakdowns. Instead, XCSF's performance usually decreases rather smoothly as the available resources (e.g., the number of available macroclassifiers) diminish.

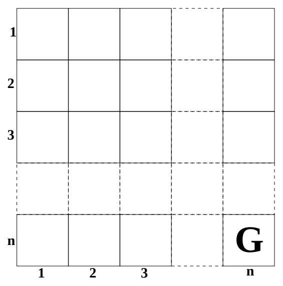


Figure 5: The Room(n) environment. The G denotes the goal position.

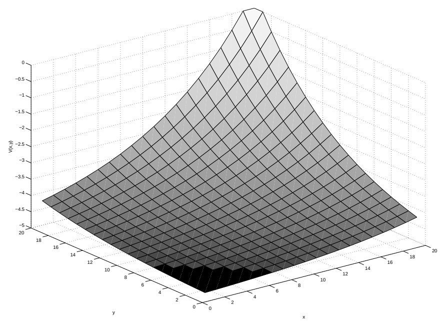


Figure 6: Optimal value function for Room(20) with $\gamma = 0.9$

7. ROOMS WITH OBSTACLES

Finally, we apply XCSF to two environments, depicted in Figure 9, Holes10 and Side10, derived from Room(10) by adding obstacles. As in [1], obstacles are positions in which there is an additional cost for moving: when the system enters an obstacle position it receives a reward of -2 (an empty position returns a reward of -0.5).

In the first experiment we apply XCSF to Holes10 (Figure 9a) for $\gamma = 0.9$ and two values of the population size, $N = 2500$ and $N = 1000$; all the other parameters are set as in the previous experiments. Figure 10 compares the performance of tabular Q-learning (which provides reference to optimal performance) with that of XCSF. As can be noted, XCSF performs optimally when $N = 2500$, the average number of macroclassifiers in the evolved populations is 434.5, the 17% of N; when N is decreased to 1000, the average performance is nearly optimal, with an average solution consisting of 200 classifiers, the 20% of N. Again, we note that even when the computation resources are drastically reduced XCSF's performance does not experience sudden breakdowns as other models, instead it smoothly decreases, as the available resources diminish. Overall, when $N = 2500$ the average solution appears to be larger than the size of the Q-table (which is 400), however, final populations contain many newly created classifiers that may also overlap with other ones. Thus, the size of the average evolved solution is actually smaller that the Q-table.

In the second experiment we apply XCSF to Side10 (Figure 9b), which is derived from Room(10) by adding a region of negative reward on one side. The size of the region and the cost associated to the obstacle positions are such that

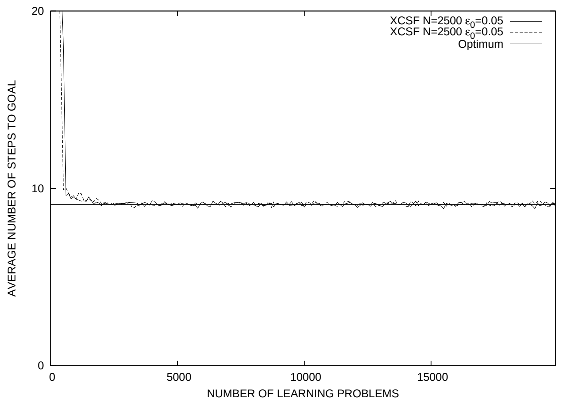


Figure 7: Performance of XCSF in Room(10) with $N = 2500$, $\gamma = 0.9$, and $\epsilon_0 = 0.05$.

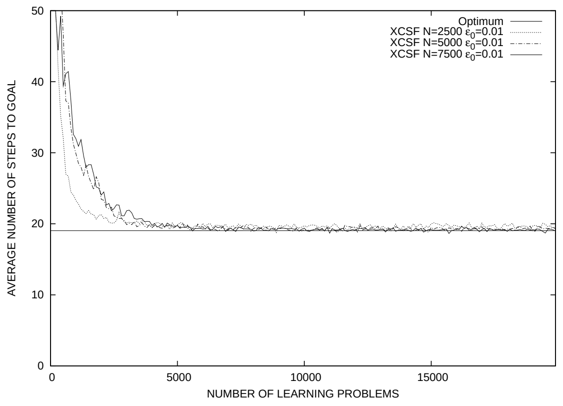
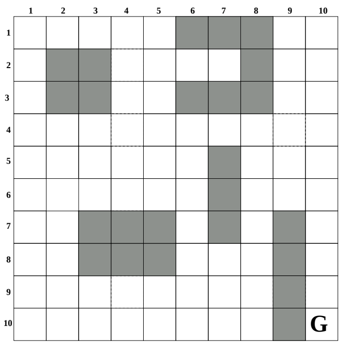
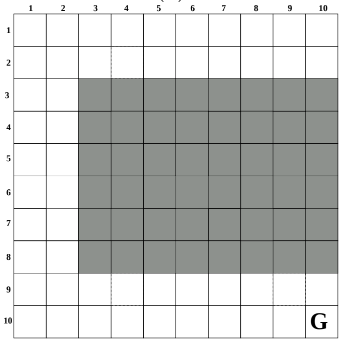


Figure 8: Performance of XCSF in Room(20) when $\gamma = 0.9$, $\epsilon_0 = 0.01$, and three values of N: 2500, 5000, 7500.

the optimal policy requires the agent to move around the obstacle border, instead of traverse it. We apply XCSF to Side10 with $\gamma = 0.9$, the same two values of N and the same parameter setting used in the previous problem. Figure 11 shows the performance of XCSF; the results confirm what found for Holes10: when $N = 2500$, XCSF's performance is optimal, with solutions consisting on the average of 404.25 macroclassifiers, the 16.18% of N; when $N = 1000$, XCSF's performance is nearly optimal, with solutions consisting on the average of 165.58 macroclassifiers, the 16.55% of N. Again, even if the population is highly reduced XCSF's performance smoothly degrades. As an example, in Figure 12 we report one of the policies evolved by XCSF for Side10 when $N = 2500$: for each position, arrows represent the direction of the best action. In Figure 13 we also report an action-value function evolved by XCSF for $N = 2500$ (Figure 13a) and an action-value function evolved by XCSF for $N = 1000$ (Figure 13b). With a population of 1000 classifiers, XCSF evolves a solution (Figure 13b) very similar to the optimal one, evolved with a population of 2500 classifiers. The only noticeable difference is found in the central section of the obstacle region (the darker area in Figure 13a) where, with the smaller population, XCSF tends to overestimate action values (as evidenced by the lighter color of the inner obstacle region in Figure 13b).



(a)



(b)

Figure 9: The environments (a) `Holes10` and (b) `Side10`. Grey positions represent obstacles.

8. CONCLUSIONS

We have applied XCSF, the extension of XCS to a computed prediction, to typical multistep problems taken from the reinforcement learning literature. In this respect, we have used XCSF as a method of generalized reinforcement learning, based on linear approximators such as those used in [1, 6]. The results we have discussed here show that XCSF can always converge to optimal or nearly optimal policies suggesting that XCSF might be a very promising approach to tackle complex multistep problems. Our experiments also suggest that XCSF is more robust than other XCS models. In fact, the reduction of the computational resources available to XCSF does not lead to sudden performance break downs. Instead, XCSF's performance smoothly degrades as the resources are reduced.

Our experience with XCSF suggests that its generalization capabilities may be further improved. As it happens in most XCS models, the degree of generalization evolved by XCSF depends on the error threshold ϵ_0 which is actually fixed right from the beginning and does not change throughout the learning process. On the other hand, in many problems, such as `Room(20)`, XCSF might reach better generalizations if different error thresholds could be used in different areas of the problem space. Accordingly, we believe that a major future research direction for XCSF consists of extending it with adaptive error thresholds which, in our opinion, would improve generalization.

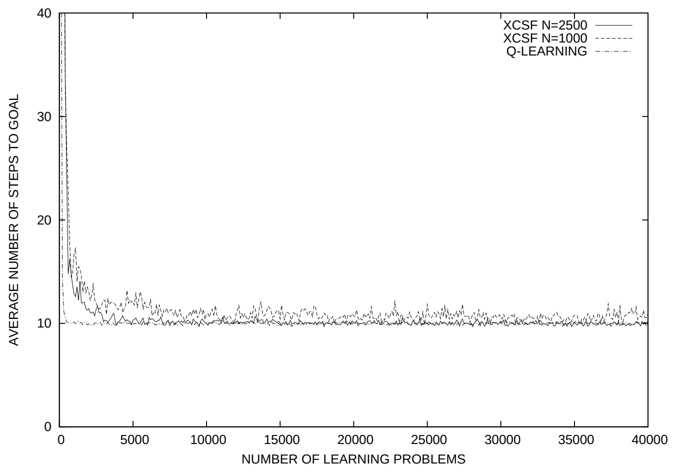


Figure 10: Performance of Q-learning and XCSF in `Holes10` for $\gamma = 0.9$. Curves are averages over 20 runs.

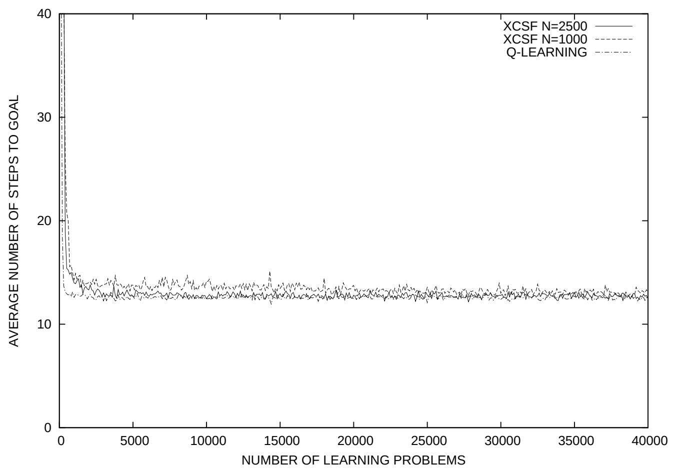


Figure 11: Performance of Q-learning and XCSF in `Side10` for $\gamma = 0.9$. Curves are averages over 20 runs.

Acknowledgments

This work was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF (F49620-03-1-0129), and by the Technology Research, Education, and Commercialization Center (TRECC), at University of Illinois at Urbana-Champaign, administered by the National Center for SupercomputingApplications (NCSA) and funded by the Office of Naval Research (N00014-01-1-0175). The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

9. REFERENCES

[1] J. A. Boyan and A. W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 369–376, Cambridge, MA, 1995. The MIT Press.

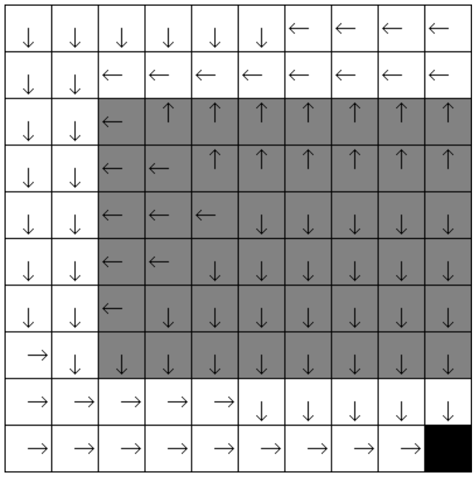


Figure 12: Example of an optimal policy evolved by XCSF for `Side10` with $N = 2500$ and $\gamma = 0.9$.

[2] M. V. Butz and S. W. Wilson. An algorithmic description of xcs. *Journal of Soft Computing*, 6(3–4):144–153, 2002.

[3] P. L. Lanzi and M. Colombetti. An Extension to the XCS Classifier System for Stochastic Environments. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 353–360, Orlando (FL), July 1999. Morgan Kaufmann.

[4] T. J. Perkins and D. Precup. A convergent form of approximate policy iteration. pages 1595–1602, 2003.

[5] R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*, pages 1038–1044. The MIT Press, Cambridge, MA., 1996.

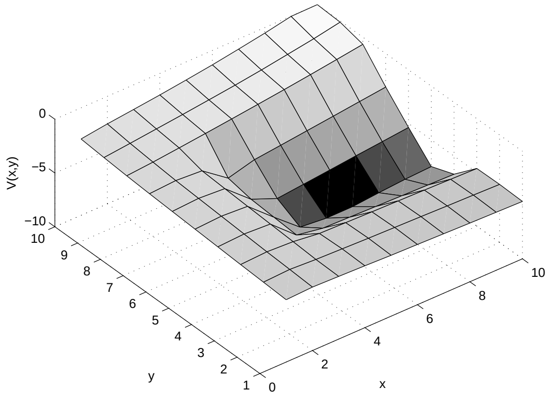
[6] R. S. Sutton and A. G. Barto. *Reinforcement Learning – An Introduction.* MIT Press, 1998.

[7] G. Tesauro. TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

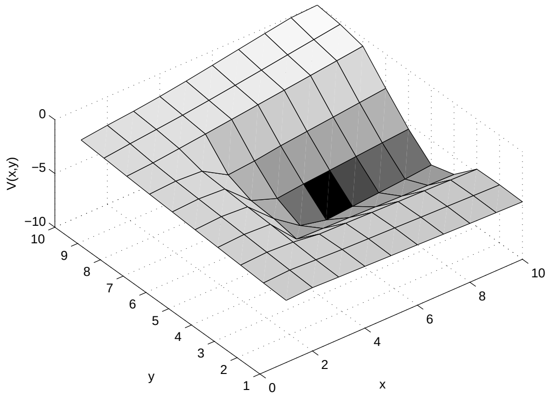
[8] S. Thrun and A. Schwartz. Issues in Using Function Approximation for Reinforcement Learning. 1993.

[9] B. Widrow and M. E. Hoff. *Adaptive Switching Circuits*, chapter Neurocomputing: Foundation of Research, pages 126–134. The MIT Press, Cambridge, 1988.

[10] S. W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995. http://prediction-dynamics.com/.



(a)



(b)

Figure 13: Examples of action-value functions evolved by XCSF for `Side10` when (a) $N = 2500$ and (b) $N = 1000$.

[11] S. W. Wilson. Mining Oblique Data with XCS. volume 1996 of *Lecture notes in Computer Science*, pages 158–174. Springer-Verlag, Apr. 2001.

[12] S. W. Wilson. Classifiers that approximate functions. *Journal of Natural Computating*, 1(2-3):211–234, 2002.

[13] S. W. Wilson. Classifier systems for continuous payoff environments. In K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E. Burke, P. Darwen, D. Dasgupta, D. Floreano, J. Foster, M. Harman, O. Holland, P. L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, and A. Tyrrell, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 824–835, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.