XCS with Computed Prediction in Continuous Multistep Environments

Pier Luca Lanzi†*, Daniele Loiacono†, Stewart W. Wilson‡*, David E. Goldberg*
†Artificial Intelligence Laboratory, Dip. di Elettronica e Informazione
Politecnico di Milano – Milano 20133, Italy
*Illinois Genetic Algorithms Laboratory, Dept. of General Engineering
University of Illinois at Urbana-Champaign, Urbana, Illinois, 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- We apply XCS with computed prediction (XCSF) to tackle multistep reinforcement learning problems involving continuous inputs. In essence we use XCSF as a method of generalized reinforcement learning. We show that in domains involving continuous inputs and delayed rewards XCSF can evolve compact populations of accurate maximally general classifiers which represent the optimal solution to the target problem. We compare the performance of XCSF with that of tabular Q-learning adapted to the continuous domains considered here. The results we present show that XCSF can converge much faster than tabular techniques while producing more compact solutions. Our results also suggest that when exploration is less effective in some areas of the problem space, XCSF can exploit effective generalizations to extend the evolved knowledge beyond the frequently explored areas. In contrast, in the same situations, the convergence speed of tabular Q-learning worsens.

1 Introduction

Computed prediction represents a major advance in learning classifier system research. XCS with computed prediction [16, 18, 19], or XCSF[1], has been recently applied (i) to evolve piecewise linear [18] and polynomial approximations of functions [4, 5]; (ii) to the learning of Boolean functions using piecewise linear approximation, as well as perceptron based and sigmoid based approximations [6]; (iii) to solve multistep reinforcement learning problems involving integer inputs and delayed rewards [7]; and also (iv) to solve *single-step* problems involving continuous inputs [19].

We now take XCS with computed prediction even further and apply it to *multistep* reinforcement learning problems involving continuous inputs. These types of problems represent an important testbed for XCSF and for learning classifier systems in general. Tabular reinforcement learning techniques such as Q-learning [13] can be applied to the simple multistep problems involving integer inputs, like those considered in [7]. However, they *cannot* be applied to problems involving continuous inputs, unless fine grained discretization is applied to preprocess the input space [10]. Unfortunately, such discretization process requires large tabular representations and most important, in large multistep problems, it is source of slow convergence. In contrast, XCSF can be directly applied to continuous domains [19]. Furthermore, the first results on a simple *single-step* problem involving continuous inputs (the frog problem [19]) demonstrate that XCS with computed prediction can (i) evolve populations which represent accurate approximations of the problem solutions, (ii) providing effective generalizations by evolving classifiers which cover large sections of the problem.

In this paper, we apply XCSF to a set of multistep problems, taken from the reinforcement learning literature [2], involving continuous inputs. The results we present show that XCSF can rapidly evolve populations of accurate maximally general classifiers which represent optimal solutions of the considered problems. We compare XCSF with a version of tabular Q-learning adapted to the continuous domains considered here. For this purpose, we follow the typical approach suggested in the reinforcement learning literaure [10]. First, we empirically determine an adequate discretization of the problem space (in this case a 100×100 grid). Then we apply tabular Q-learning on the discretized version of the problems considered and compare the results with those obtained with XCSF on the original problem. Our results show that, tabular Q-learning applied to the discretized problem domain tends to reach near optimal solutions very slowly while also requiring large Q-tables. In contrast, XCSF rapidly converges to optimal solutions that require much less space than the corresponding Q-tables.

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* [10]. 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* [10].

[1] XCS with computed prediction was first introduced as XCSF in [16] to approximate functions defined over integer domains and later extended to XCS-LP in [19] for single step problems defined over continuous domains involving discrete actions. In this paper we generally use the name XCSF to abstract the general concept of computed prediction from the specific implementation.

0-7803-9363-5/05/$20.00 ©2005 IEEE.

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 values. 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. In addition, look-up tables cannot be applied to problems involving continuous state spaces. When continuous inputs are involved the only approach to apply tabular techniques consists of applying a discretization step to preprocess the inputs before look-up tables can be used. However, this approach raises some serious issues: while a coarse-grained discretization might prevent the learner from reaching the optimal performance, on the other hand, a fine-grained discretization might lead to large tables and slow convergence. 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: the action-value function is not represented as a table but as a function parameterized with a vector $\vec{\theta}$. 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 $\vec{\theta_t}$. The action-value function $Q(\cdot,\cdot)$ is viewed as a function parameterized by a vector $\vec{\theta}$ 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, $\vec{\theta_t} = \langle\theta_t(1)\ldots\theta_t(n)\rangle$, while the target action-value function is approximated by a smooth differentiable function of $\vec{\theta_t}$ for all the possible state-action pairs. At time step t, parameters $\vec{\theta_t}$ 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 [2, 9, 10]. With linear methods, value functions are represented by linear functions of the vector parameter $\vec{\theta_t}$. For any state s, a vector of n features, $\vec{\phi_s} = \langle\phi_s(1),\ldots,\phi_s(n)\rangle$ is extracted, the approximated value function for s is simply computed as $\vec{\theta_t}\vec{\phi_s}$, while the gradient simply corresponds to the vector $\vec{\phi_t}$. Linear methods are very computationally efficient in terms both of space and time; on the other hand, their effectiveness relies heavily on the choice of the feature vector $\vec{\phi_s}$, 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 relations between input features, e.g., coarse coding, tile coding, radial basis functions, and kanerva coding [10].

2.3 Convergence

The convergence of generalized reinforcement learning is a complex issue, still poorly understood (see [8] 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 [11], yet they have been shown to be generally unstable, even in simple problems [1]. In addition there are a large number of factors that influence the performance of function approximation approaches, such as: the algorithm (e.g., Q-learning or SARSA [10]), the approximation used (e.g., linear, or tile coding), or the window used for the update (e.g., eligibility traces). For instance, [12] suggests (in the case of Q-learning [10]), 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 [18]: (i) classifiers conditions are extended for numerical inputs, as done for XCSI [17]; (ii) classifiers are extended with a vector of weights w, that are used to compute classifier's prediction; finally, (iii) the weights 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 reals (whereas in the original XCSF they were integers [18]). The action specifies the action for which the payoff is predicted. The four parameters are: the weight vector $\vec{w}$, 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 $\vec{w}$ 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; covering is controlled by the parameter r_0 and it works as in XCSI [17, 18] but considers real values instead of integers. The weight vector w of covering classifiers is randomly initialized with values from [-1,1]; all the other parameters are initialized as in XCS.

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 [3], 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, $[M]|_a$ 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:

$$cl.p(s_t) = cl.w_0 \times x_0 + \sum_{i>0} cl.w_i \times s_t(i) \quad (2)$$

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 $[A]_{-1}$ corresponding to the previous time step, $t-1$. At time step t, the expected payoff P is computed as:

$$P = r_{-1} + \gamma \max_{a \in A} P(s_t, a) \quad (3)$$

where r_{-1} is the reward received at the previous time step. The expected payoff P is used to update the weight vector w of the classifier in $[A]_{-1}$ using a *modified delta rule* [14]. For each classifier $cl \in [A]_{-1}$, each weight $cl.w_i$ is adjusted by a quantity Δw_i computed as:

$$\Delta w_i = \frac{\eta}{|s_{t-1}(i)|^2}(P - cl.p(s_{t-1}))s_{t-1}(i) \quad (4)$$

where η is the correction rate and $|s_{t-1}|^2$ is the norm the input vector s_{t-1}, (see [18] for details). The values Δw_i are used to update the weights of classifier cl as:

$$cl.w_i \leftarrow cl.w_i + \Delta w_i \quad (5)$$

Then the prediction error ε is updated as:

$$cl.\varepsilon \leftarrow cl.\varepsilon + \beta(|P - cl.p(s_{t-1})| - cl.\varepsilon) \quad (6)$$

Finally, classifier fitness is updated as in XCS.

Discovery Component. In XCSF, the genetic algorithm works as in XCSI [17]; in the version used here the mutation of classifier conditions, controlled by parameter r_0, is based on real values as in [19] instead of integers as in [17].

4 Design of Experiments

To apply XCSF to continuous multistep problems, we follow the standard experimental design used in the literature [15]. 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 [2]. 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 1000 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 10 experiments.

We applied XCSF to two classes of real valued multistep environments: the one dimensional linear corridor and the *2D continuous gridworld* environments firstly introduced in [2].

5 The Continuous Linear Corridor

We start from a very simple environment, a one dimensional linear corridor `Corr(s)`, in which the system inputs are defined over the interval $[0, 1]$, the goal is in position 1, and there are two possible actions: *left*, coded with 0, which corresponds to a change of "$-s$" in the system position (when the system tries to move below position 0, the system reaches position 0); *right*, coded with 1, which corresponds to a change of "$+s$" in the system position. The step-size parameter s determines the size of the environment with respect to the system movement capabilities, the smaller the s, the larger the environment, the larger the s, the smaller the environment. The system can start anywhere but in the goal position (i.e., in the real interal $[0, 1)$) and it reaches the goal when moving in 1 or beyond 1 (i.e., the goal is reached if the system position is greater or equal than 1). When the system reaches the goal it receives zero reward, in all the other cases it receives -0.5. Given the step-size s the average number of steps to reach the goal position is computed as $(s+1)/2s$ (see Appendix A).

In the first experiment we apply XCSF to the continuous corridor with a step size of 0.05, `Corr(0.05)`, with the following parameter settings: $N = 400$, $\epsilon_0 = 0.01$, $\beta = 0.2$; $\alpha = 0.1$; $\nu = 5$; $\chi = 0.8$, $\mu = 0.04$, $p_{\text{explr}} = 0.5$, $\theta_{del} = 50$, $\theta_{GA} = 50$, and $\delta = 0.1$; GA-subsumption is on with $\theta_{sub} = 50$; while action-set subsumption is off; the parameters for real-valued conditions are $m_0 = 0.5$, $r_0 = 0.25$;

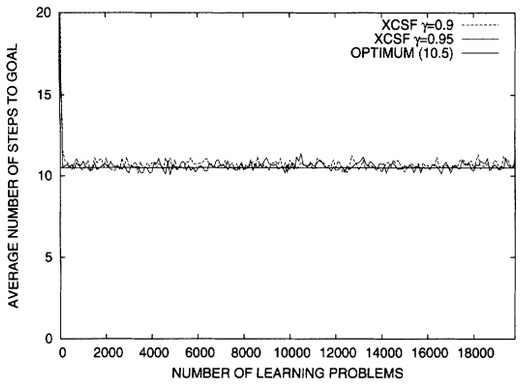

Figure 1: XCSF in `Corr(0.05)` when ϵ_0=0.01, $\gamma = 0.9$, and $\gamma = 0.95$.

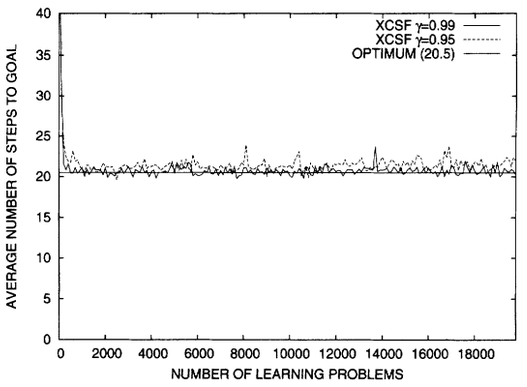

Figure 2: XCSF in `Corr(0.025)` when ϵ_0=0.01, $\gamma = 0.95$, and $\gamma = 0.99$.

the parameter x_0 for XCSF is 1 [18]. Figure 1 compares the performance of XCSF on the `Corr(0.05)` for two values of discount factor, $\gamma = 0.9$ and $\gamma = 0.95$. As shown, when $\gamma = 0.95$ XCSF reaches optimal performance while with $\gamma = 0.9$ the performance of XCSF is slightly over the optimum. With respect to the population size, in both cases the final populations contain less than 40 classifiers: when $\gamma = 0.9$, the final populations contain an average of 36.4 classifiers (the 9.09% of N), when $\gamma = 0.95$ the final populations contain an average of 34.2 macroclassifiers (the 8.55% of N).

In the second experiment, we reduce the step size s from 0.05 to 0.025, making the environment larger with respect to the system actions. Figure 2 reports the performance of XCSF in `Corr(0.025)` for two values of the discount factor, $\gamma = 0.95$ and $\gamma = 0.99$. In both cases, the performance of XCSF rapidly converges towards the optimum, but the performance is fully optimal only for $\gamma = 0.99$. Likewise to the previous experiment, also in this case final populations contain on the average less than 40 classifiers: 37.52 (the 9.38% of N) when $\gamma = 0.95$, 28.36 (the 7.09% of N) when $\gamma = 0.99$. As the γ approaches to one, the optimal value function for the continuous corridor approaches a line (see [7]) and thus is better approximated by XCSF. Accordingly, on the average the evolved solutions are smaller when $\gamma = 0.99$.

6 The 2D Continuous Gridworlds

This class of environments has been introduced in [2] when studying reinforcement learning with function approximators. They are two dimensional environments in which the current state is defined by a pair of real valued coordinates $\langle x, y\rangle$ in $[0, 1]^2$, the only goal is in position $\langle 1, 1\rangle$, and there are four possible actions (left, right, up, and down) coded with two bits; each action corresponds in a step of size s in the corresponding direction; actions that would take the system outside the domain $[0, 1]^2$ take the system to the nearest position of the grid border. The system can start *anywhere* but in the goal position and it reaches the goal position when *both* coordinates are equal or greater than one. When the system reaches the goal it receives 0, in all the other cases it receives -0.5.

6.1 Empty Continuous Gridworlds

We begin from the simplest environment an empty gridworld, `Grid(s)`, that XCSF has to solve with a step size s. Given the step-size s the average number of steps to reach the goal in `Grid(s)` is computed as $(s + 1)/s$ (see Appendix A for details). In the first experiment we apply XCSF to `Grid(0.05)` (which requires an average of 21 steps to reach the goal) with different values of population size N and the following parameters: $\epsilon_0 = 0.005$; $\beta = 0.2$; $\alpha = 0.1$; $\gamma = 0.95$; $\nu = 5$; $\chi = 0.8$, $\mu = 0.04$, $p_{\text{expl}} = 0.5$, $\theta_{del} = 50$, $\theta_{GA} = 50$, and $\delta = 0.1$; GA-subsumption is on with $\theta_{sub} = 50$; while action-set subsumption is off; the parameters for integer conditions are $m_0 = 0.5$, $r_0 = 0.25$ [17]; the parameter x_0 for XCSF is 1 [18].

Figure 3 compares the performance of XCSF when $N = 5000$, $N = 7500$, and $N = 10000$. As can be noted, with all the three values of N, XCSF converges to the optimum; more precisely, with $N = 10000$ the performance is perfectly optimal, while with $N = 5000$ and $N = 7500$, the performance is slightly above the optimum. Noticeably, the results also show that the decrease of population size has almost no influence in XCSF performance, with all the three population sizes XCSF rapidly converges near the optimum. The populations evolved are in all the three cases rather compact and always below the 10% of N; more precisely, final populations contain on the average, 991.8 classifiers (the 9.91% of N) when $N = 10000$, 791.3 classifiers (the 10.55% of N) when $N = 7500$, and 606.5 classifiers (the 12.13% of N) when $N = 5000$.

It is interesting to compare XCSF to tabular Q-learning. For this purpose, before applying Q-learning, we need to select an adequate discretization of the state space; we tried different discretizations (e.g., 50×50, 75×75, and 100×100) and found that Q-learning would perform best when the state space $[0, 1]^2$ is discretized according to a 100×100 grid; in coarser grids Q-learning would perform much worse; note that in this case, the step size $s = 0.05$ corresponds to a move of 5 positions on the discretized grid. For this comparison, both XCSF and Q-learning use the

same explore-exploit strategy. Figure 4 compares the performance of XCSF with $N = 10000$ to tabular Q-learning applied with the 100×100 discretization. The performance curve of XCSF covers only the very first section of the plot, since XCSF converges to the optimum much faster than the tabular Q-learning applied to the 100×100 grid. Noticeably, even after 250000 problems, the performance of Q-learning is still slightly over the optimum; to reach full optimality, Q-learning would need more accurate discretization of the state space, though the convergence would be even slower. These results confirm what discussed in Section 2.3: in problems involving continuous state spaces, only fine grained discretizations allow the convergence to optimal performance but this involves a very slow learning process. With respect to generalization it is also interesting to note that Q-learning applied to `Grid(0.05)` with a 100×100 discretization requires a Q-table of 40000 entries, whereas XCSF requires less than 1000 classifiers.

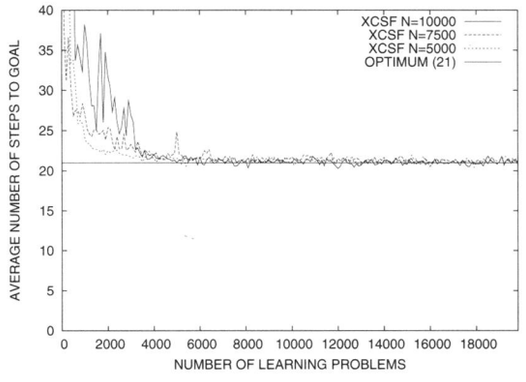

Figure 3: XCSF in `Grid(0.05)` for $N \in \{5000, 7500, 10000\}$; curves are averages over 10 runs.

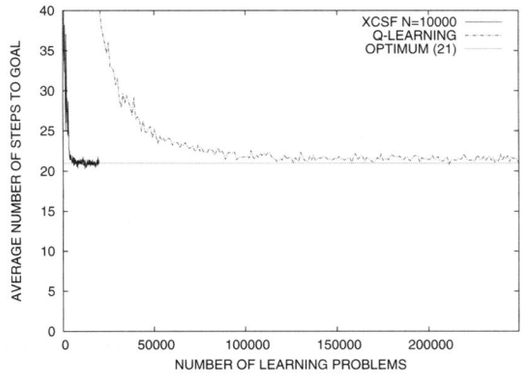

Figure 4: XCSF and the Q-learning in `Grid(0.05)`. Curves are averages over 10 runs.

6.2 Continuous Gridworld with Puddles

We now add obstacles to the empty continuous gridworld discussed in the previous section. We follow the approach in [2] and represent obstacles as areas in which there is an additional cost for moving. These areas are called "puddles" [2], since they actually create a sort of puddle in the optimal value function. Figure 5 depicts the `Puddles(s)` environment that is derived from `Grid(s)` by adding two puddles (the gray areas). When the system is in a puddle, it receives an additional negative reward of -2, i.e., the action has an additional cost of -2; in the area where the two puddles overlap, the darker gray region, the two negative rewards add up, i.e., the action has a total additional cost of -4. Note that for this environment there is not a simple expression of the average number of steps required to reach the goal.

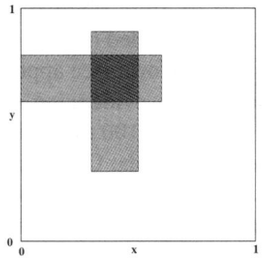

Figure 5: The `Puddles(s)` environment: the light gray regions represent the two puddles; the dark gray region is where the two puddles overlap; the goal is in position $\langle 1, 1 \rangle$.

In the first experiment, we apply XCSF to `Puddles(0.1)` and compare its performance with tabular Q-learning by discretizing the state space according to the same 100×100 grid used in the previous section. We set XCSF parameters as follows: ϵ_0=0.005, $\beta = 0.2$; $\alpha = 0.1$; $\gamma = 0.95$; $\nu = 5$; $\chi = 0.8$, $\mu = 0.04$, $p_{\text{explr}} = 0.5$, $\theta_{del} = 50$, $\theta_{GA} = 50$, and $\delta = 0.1$; GA-subsumption is on with $\theta_{sub} = 50$; while action-set subsumption is off; the parameters for integer conditions are $m_0 = 0.5$, $r_0 = 0.25$ [17]; the parameter x_0 for XCSF is 1 [18]. Figure 6 compares the performance of XCSF in `Puddles(0.1)` when $N = 5000$, $N = 7500$, and $N \leftarrows 10000$ with the performance of tabular Q-learning obtained after 250000 learning problems, using the 100×100 discretization. As can be noted, all the three versions of XCSF converge rapidly to a performance that is slightly better than tabular Q-learning. After 4000 learning problems the performance of all the three versions of XCSF is below Q-learning; note however that when $N = 5000$ spikes in the performance of XCSF appear. Figure 7 compares the performance of XCSF when $N = 10000$ with Q-learning performed on the 100×100 discretization. As in the case of the empty grid, XCSF converges much faster than tabular Q-learning applied to the discretized version of the same environment. The solutions evolved by XCSF are also rather compact, containing an average of 1270 classifiers (the 12.7% of N) when $N = 10000$, 1027.5 classifiers (the 13.7% of N) when $N = 7500$, and 420 classifiers (the 8.4% of N) when $N = 5000$.

In the second experiment, we extend previous results and we apply XCSF to `Puddles(0.05)` with the same settings used in the previous experiment. Figure 8 compares the performance of XCSF in `Puddles(0.05)` when $N = 5000$, $N = 7500$, and $N = 10000$ with the performance of tabular Q-learning obtained after 250000 learning problems, using

the 100×100 discretization. Figure 9 compares the performance of XCSF when $N = 10000$ with Q-learning performed on the 100×100 discretization. Figure 11 reports an example of optimal value function evolved by XCSF for `Puddles(0.05)`; to report the value function we sample the state space with a resolution of 0.05. Figure 10 reports an example of value function developed by Q-learning on the 100×100 discretization.

The results for `Puddles(0.05)` confirm the ones obtained for `Puddles(0.1)`. XCSF can rapidly converge to a solution that appears to be fully optimal when a sufficient number of classifiers is provided ($N = 10000$); while with fewer classifiers XCSF performance appears slightly worse and sometimes more noisy evidencing some spikes. On the other hand, on the 100×100 discretization of the state space, the convergence of tabular Q-learning is much slower than that of XCSF. Note that the convergence of Q-learning mainly depends on the complexity of the state space and not on the problem itself. In fact, the convergence speed of Q-learning in `Puddles(0.1)` is almost slower than in `Puddles(0.05)`, although the latter environment, with respect to the system actions, is larger than the former one. Also in `Puddles(0.05)` the solutions evolved by XCSF are rather compact in that they contain an average of 1720 classifiers (the 17.2% of N) when $N = 10000$, 892.5 classifiers (the 11.9% of N) when $N = 7500$, and 410 classifiers (the 8.2% of N) when $N = 5000$.

Noticeably, even if for XCSF `Puddles(0.05)` is virtually *four times* larger than `Puddles(0.1)` (with a smaller step size more actions are required to reach the goal position), the populations evolved by XCSF in the two cases contain on the average the same number of classifiers. For instance, when $N = 10000$ in `Puddles(0.1)` the evolved solutions contain an average of 1270 classifiers, in `Puddles(0.05)` the evolved solutions contain an average of 1700 classifiers. I.e., XCSF has been able to partition the state space so as to produce effective generalizations, more or less independently from the action effect. In contrast, if we compare the performance of Q-learning in `Puddles(0.1)` and `Puddles(0.05)` (Figure 7 and Figure 9, respectively) we note that the convergence of Q-learning is slightly slower. This because with a large step size the exploration of the state space is less effective (the systems jumps around too much). Since tabular Q-learning has no generalization capabilities, the reduction of exploration in `Puddles(0.1)` corresponds to a decrease in the convergence speed. In contrast, XCSF can exploit effective generalizations to extend the evolved knowledge beyond the highly explored areas.

7 Conclusions

We have applied XCSF to multistep problems involving continuous inputs. We have presented results showing that XCSF can easily converge toward optimal performance while also producing compact representation of the solutions. The comparison with tabular Q-learning adapted to continuous domains shows that XCSF can converge faster than such tabular methods and requires less memory to store

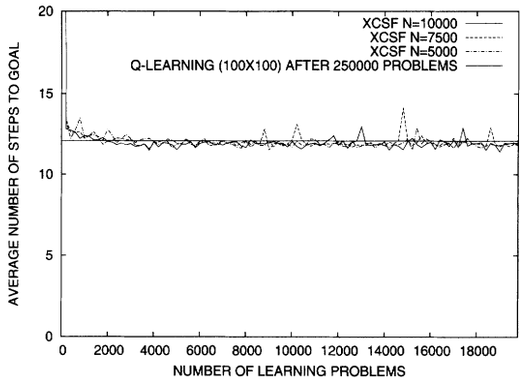

Figure 6: XCSF in `Puddles(0.1)`. Curves are averages over 10 runs.

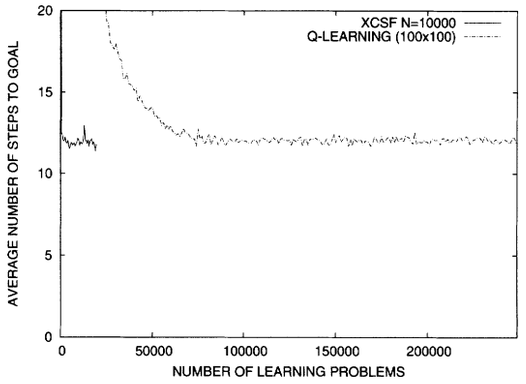

Figure 7: XCSF and Q-learning in `Puddles(0.1)`. Curves are averages over 10 runs.

the final solutions. Noticeably, XCSF appears to be rather robust: even if the number of available classifiers is drastically reduced (e.g., it is halved), XCSF can still converge near to optimal performance. Future research directions include the extension to domains involving noise and to more difficult multistep problems.

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.

Bibliography

[1] J. Boyan and A. Moore. Generalization in reinforcement learning: Safely approximating the value function. In *Proceedings of Neural Information*

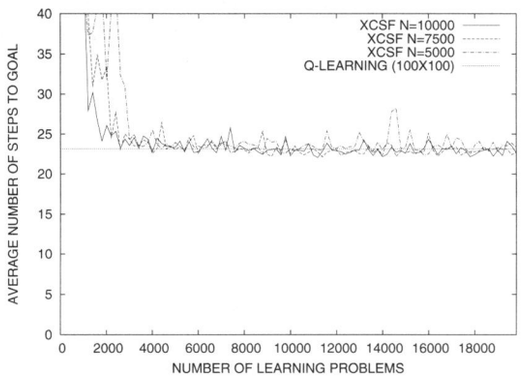

Figure 8: XCSF in `Puddles(0.05)`. Curves are averages over 10 runs.

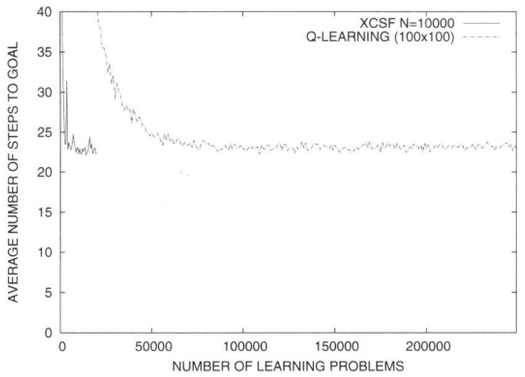

Figure 9: XCSF and Q-learning in `Puddles(0.05)`. Curves are averages over 10 runs.

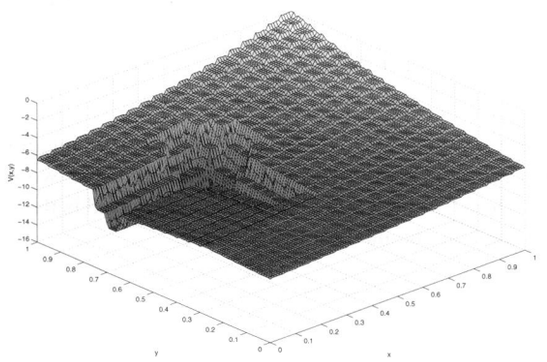

Figure 10: An example of value function obtained with tabular Q-learning in `Puddles(0.05)`.

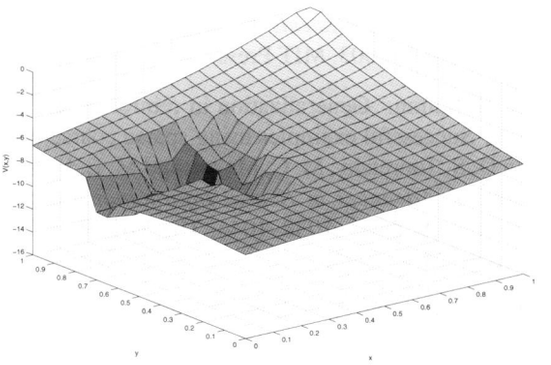

Figure 11: An example of value function evolved by XCSF for `Puddles(0.05)`.

Processings Systems 7. Morgan Kaufmann, 1995. http://www.cs.cmu.edu/~awm/papers.html.

[2] 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.

[3] M. V. Butz and S. W. Wilson. An Algorithmic Description of XCS. In P. L. Lanzi, W. Stolzmann, and S. W. Wilson, editors, *Advances in Learning Classifier Systems*, volume 1996 of *LNAI*, pages 253–272. Springer-Verlag, Berlin, 2001.

[4] P. L. Lanzi, D. Loiacono, S. W. Wilson, and D. E. Goldberg. Extending xcsf beyond linear approximation. Technical Report 2005006, Illinois Genetic Algorithms Laboratory – University of Illinois at Urbana-Champaign, 2005.

[5] P. L. Lanzi, D. Loiacono, S. W. Wilson, and D. E. Goldberg. Generalization in the xcsf classifier system: Analysis, improvement, and extension. Technical Report 2005012, Illinois Genetic Algorithms Laboratory – University of Illinois at Urbana-Champaign, 2005.

[6] P. L. Lanzi, D. Loiacono, S. W. Wilson, and D. E. Goldberg. Xcs with computable prediction for the learning of boolean functions. Technical Report 2005007, Illinois Genetic Algorithms Laboratory – University of Illinois at Urbana-Champaign, 2005.

[7] P. L. Lanzi, D. Loiacono, S. W. Wilson, and D. E. Goldberg. Xcs with computable prediction in multistep environments. Technical Report 2005008, Illinois Genetic Algorithms Laboratory – University of Illinois at Urbana-Champaign, 2005.

[8] T. J. Perkins and D. Precup. A convergent form of approximate policy iteration. In S. T. S. Becker and

K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 1595–1602, Cambridge, MA, 2003. MIT Press.

[9] 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.

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

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

[12] S. Thrun and A. Schwartz. Issues in Using Function Approximation for Reinforcement Learning. In M. Mozer, P. Smolensky, D. Touretzky, J. Elman, and A. Weigend, editors, *Proceedings of the 1993 Connectionist Models Summer School*, Hillsdale, NJ, 1993. Lawrence Erlbaum.

[13] C. Watkins. *Learning from delayed reward*. PhD thesis, 1989.

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

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

[16] S. W. Wilson. Function approximation with a classifier system. In L. S. et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 974–981, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

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

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

[19] S. W. Wilson. Classifier systems for continuous payoff environments. In K. D. et al., editor, *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.

A Optimal Average Performance

We now compute the average number of steps required by an optimal policy in the continuous linear corridor `Corr(s)` and in the 2D continuous gridworld.

A.1 The linear corridor

To compute the optimal average number of steps for `Corr(s)` we first compute the minimum number of steps to reach the goal position starting from position $x \in [0, 1)$, as

$$\left\lceil \frac{1-x}{s} \right\rceil, \tag{7}$$

Now the average number of step can be obtained integrating Equation 7 on all the input space:

$$\overline{steps^*} = \frac{\int_0^1 \left\lceil \frac{1-x}{s} \right\rceil dx}{\int_0^1 dx} \tag{8}$$

Applying the substitution $z = \frac{1-x}{s}$ and considering that $\int_0^1 dx = 1$, Equation 8 becomes:

$$\overline{steps^*} = s \int_0^{\frac{1}{s}} \lceil z \rceil \, dz \tag{9}$$

Integral in Equation 9 can be easily solved using the additive property of integrals:

$$\begin{aligned} \overline{steps^*} &= s \sum_{i=0}^{\frac{1}{s}-1} \int_i^{i+1} \lceil z \rceil \, dz = s \sum_{i=0}^{\frac{1}{s}-1} \int_i^{i+1} (i+1) dz \\ &= s \sum_{i=0}^{\frac{1}{s}-1} (i+1) = s \cdot \frac{\frac{1}{s} \cdot (\frac{1}{s}+1)}{2} \\ &= \frac{(\frac{1}{s}+1)}{2} = \frac{(s+1)}{2s} \end{aligned} \tag{10}$$

where we make the hypothesis[2] that $\frac{1}{s} \in \mathbb{N}$.

A.2 The Empty 2D Continuous Grid

The extension to `Grid(s)` is straightforward. In this case, the minimum number of steps to reach the goal starting from position $\langle x, y \rangle$ is:

$$\left\lceil \frac{1-x}{s} \right\rceil + \left\lceil \frac{1-y}{s} \right\rceil. \tag{11}$$

As in the previous case, the average number of step can be obtained integrating Equation 11 on all the input space:

$$\overline{steps^*} = \frac{\int_0^1 \int_0^1 \left(\left\lceil \frac{1-x}{s} \right\rceil + \left\lceil \frac{1-y}{s} \right\rceil \right) dxdy}{\int_0^1 \int_0^1 dxdy}. \tag{12}$$

Considering that $\int_0^1 \int_0^1 dxdy = 1$ and that the integral at numerator of Equation 12 can be separated in the two variables, $\overline{steps^*}$ in the empty continuous gridworld is the double of the one obtained for the corridor:

$$\overline{steps^*} = \frac{s+1}{s} \tag{13}$$

[2] This assumption it is not strict and it can be easily relaxed. In fact if $\frac{1}{s} \notin \mathbb{N}$ Equation 9 can be written as,

$$\overline{steps^*} = s \int_0^{\lfloor \frac{1}{s} \rfloor} \lceil z \rceil \, dz + \int_{\lfloor \frac{1}{s} \rfloor}^{\frac{1}{s}} \left\lceil \frac{1}{s} \right\rceil dz,$$

and therefore $\overline{steps^*} = \frac{\lfloor \frac{1}{s}+1 \rfloor}{2} + \lceil \frac{1}{s} \rceil \cdot \left(\frac{1}{s} - \lfloor \frac{1}{s} \rfloor \right)$