Citation for published version:
Drugowitsch, J & Barry, AM 2005, *XCS with eligibility traces*. Computer Science Technical Reports, no. CSBU-2005-01, University of Bath, Department of Computer Science.

Publication date:
2005

Link to publication

©The Author January 2005

University of Bath

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 07. Dec. 2019

Department of
Computer Science

UNIVERSITY OF BATH

Technical Report

XCS with Eligibility Traces

Jan Drugowitsch and Alwyn Barry

Technical Report 2005-01 **January 2005**

ISSN 1740-9497

Copyright ©January 2005 by the authors.

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

XCS with Eligibility Traces

Jan Drugowitsch
Department of Computer Science
University of Bath, UK
J.Drugowitsch@bath.ac.uk

Alwyn M Barry
Department of Computer Science
University of Bath, UK
A.M.Barry@bath.ac.uk

January 2005

Abstract

The development of the XCS Learning Classifier System has produced a robust and stable implementation that performs competitively in direct–reward environments. Although investigations in delayed–reward (i.e. multi–step) environments have shown promise, XCS still struggles to efficiently find optimal solutions in environments with long action–chains. This paper highlights the strong relation of XCS to reinforcement learning and identifies some of the major differences. This makes it possible to add Eligibility Traces to XCS, a method taken from reinforcement learning to update the prediction of the whole action–chain on each step, which should cause prediction update to be faster and more accurate. However, it is shown that the discrete nature of the condition representation of a classifier and the operation of the genetic algorithm cause traces to propagate back incorrect prediction values and in some cases results in a decrease of system performance. As a result further investigation of the existing approach to generalisation is proposed.

1 Introduction

Learning Classifier Systems (LCS) are a class of machine learning techniques that utilise evolutionary computation to provide the main knowledge induction algorithm. They are characterised by the representation of knowledge in terms of a population of simplified production rules (classifiers) in which the conditions are able to cover one or more inputs. The *Michigan* LCS [16] maintains a single population of production rules with a Genetic Algorithm (GA) operating with the population ... each rule maintains its own fitness estimate. LCS are general machine learners, primarily limited by the constraints in the representation adopted for the production rules (see, for example, [34]) and by the complexity of the solutions that can be maintained under the action of the GA [12].

LCS have been successfully applied to many application areas – most notably for Data Mining [22, 18, 6] but also more complex problems (e.g. [14, 26]). In particular the now commonly known XCS[1] brought significant improvements in the robustness and generalisation abilities of such classifier systems, as empirically demonstrated in direct reward environments in many cases (e.g. [32, 19, 34, 6, 13]). Solutions to delayed reward environments were found for simple environments (e.g. Woods2 [32, 33]), but XCS was shown to struggle to find the correct predictions in long chain environments, such as Woods14 [20]. Recently, due to XCS's similarity with reinforcement learning (RL) [21], performance in such environments has been improved by translating new developments in reinforcement learning to classifier systems. [7] has significantly improved XCS performance by adding direct and residual gradient descent [1, 2] to the prediction update. This allows the classifier system to find optimal predictions in Woods14 and other environments when using random start states and a low minimum error to avoid the problem of over–general classifiers identified in [5, 3].

Still, prediction update in XCS is limited to one time step (known as single–step temporal difference learning TD(0) in RL), resulting in slow reward distribution in large environments. To improve the speed of prediction update, RL provides a method called TD(λ) that keeps a decaying log of previous states (*Eligibility Traces*) which will be updated at each prediction update. This not only provides faster distribution of the reward to earlier states in the action–chain but also faster convergence to the optimal values [30, 31, 28]. Given XCS's similarity to Q-Learning[2] and the performance improvement due to Eligibility Traces in Q(λ), it is hypothesised that adding such traces to XCS will result in comparable prediction update improvements.

[1] XCS is a classifier system that uses prediction accuracy as basis for classifier fitness calculation. The interested reader is referred to [32, 33, 9].

[2] Q-Learning is off-policy TD(0) for optimal control [30].

Updating predictions for several state–action values at once is not a novel idea in classifier systems. [17] kept a record of all active classifiers during one episode and performed value update only at the end of each episode[3]. In [23], the speed of reward propagation was improved by adding static "bridging classifiers" that span over the whole action chain to update all classifiers in that chain at once. [11] even proposed TBB(λ), an improved bucket brigade algorithm [15] with truncated traces [10], but did not present any results. However, no such improvement was attempted since the emergence of XCS.

This paper investigates how Eligibility Traces can be included in XCS to allow faster prediction propagation in multi–step environments and enhance convergence properties in environments with long action–chains. The following section describe the areas where Q-Learning [30] and XCS are similar or differ and how XCS can be described as a Function Approximation (FA) technique for RL. This is followed by a more detailed description of TD(λ) and Q(λ) [30, 31], how traces are implemented in XCS and how this influences performance in a simple deterministic finite–state environment. The results are then discussed and their consequences generalised.

2 XCS and Q–Learning

2.1 Prediction and Policy

The classifier predictions of XCS are updated using the technique of Q–learning [30]. This technique is based on off–policy Temporal Difference (TD) control to directly approximate the optimal action–value function Q^*, using the update procedure[4]

$$Q(s_{t-1}, a_{t-1}) \leftarrow Q(s_{t-1}, a_{t-1}) + \beta \left(r + \gamma \max_{a \in A(s_t)} Q(s_t, a) - Q(s_{t-1}, a_{t-1}) \right) \tag{1}$$

for each action a performed in state s at time $t-1$. β is the learning rate ($0 < \beta \leq 1$), γ is the discount factor ($0 \leq \gamma \leq 1$) and r is the reward received for performing action a_{t-1} in state s_{t-1}.

The action–value function $Q(s, a)$ represents the expected return for each state–action pair. The discount factor γ effects how much future rewards are valued at the current state. Additionally, discounting of the reward acts as motivation for action selection towards rewarding states (assuming that non–rewarding actions within the environment are not penalised) as the agent's action selection is based on maximising the expected return when exploiting the learned knowledge. The policy π describing this action selection is called the greedy policy π_{greedy}, where from each state s the selected action is $a_t = \text{argmax}_{a \in A(s_t)} Q(s_t, a)$. ϵ–greedy is another commonly used policy in reinforcement learning that allows early focus on optimal control but also provides some degree of exploration. It conforms to a greedy policy with a probability of $1 - \epsilon$ and performs random action selection from $A(s_t)$ otherwise.

In XCS, in contrast, the action–value for one state–action pair is given by the accumulated, fitness–weighted prediction of the classifiers in the action set. Policy–wise, XCS commonly performs one exploration episode with purely random action selection, followed by one exploitation episode with greedy action selection. In both cases actions are performed until either a terminal state is visited or the maximum number of steps per episode is exceeded. Performance is usually only reported for exploitation episodes, which makes it hard to compare XCS and Q–Learning performance reports.

The sequence of policy evaluation and prediction–value update in Q–Learning is to firstly selects its action depending on the current policy. Then it performs another greedy policy evaluation while updating the Q–function, as can be seen in the update function (1). In comparison, XCS calculates the prediction array (PA), which is the fitness–weighted accumulation of predictions for each action in $\mathcal{M}$. The next action is selected by applying the current policy to PA. Thereafter, the prediction values of the $\mathcal{A}^{-1}$ are updated using the maximum value of PA, which is equivalent to performing a π_{greedy} evaluation on the current $\mathcal{M}$. This shows that in each subsequent step the policy evaluation is based on prediction values before they have been updated with the expected return of the previous step. Therefore, when compared with Q–Learning, XCS always lags one step behind with its prediction update. As, however, the generalised policy iteration (GPI) [28] does not put restrictions on the sequence of policy evaluation and policy improvement, XCS without generalisation can still be said to converge to optimal values with $t \to \infty$. Still, in a strict sense XCS does not perform Q–Learning.

[3]This update procedure is comparable to Monte Carlo Markov Chain methods, where predictions are updated when a terminal state is reached.

[4]Notation: Using the notation of [9], with the following differences: $\mathcal{A}$ is the action set, $\mathcal{M}$ is the match set, $\mathcal{P}$ is the full population of classifiers, S is the set of all states in the state space, $A(s)$ is the set of possible actions for state s. The environment is modelled by a Markov Decision Process (MDP), defined by a finite set of states $s \in S$, actions $a \in A(s)$, a transition function $T(T : S \times A \to \Pi(S))$ to assign each state–action pair the probability distribution $\Pi(S)$, and a reward function $R(R : S \times A \to \mathbb{R})$.

2.2 Value Function Approximation and Generalisation

Tabular Q–Learning needs to store one prediction value per state–action pair, or $\sum_{s \in S} |A(s)|$ prediction values overall. This can result in excessive storage usage for large environments. Therefore, attempts are made to reduce the number of parameters of the learner and still retain acceptable performance. In reinforcement learning, this is done by the use of known function approximation techniques, like linear approximation, gradient descent, or different tiling methods to approximate the action–value function [28].

XCS can be seen as some form of function approximator that segments the action–value function into overlapping regions. Each of these regions is defined by the condition of one classifier, which covers a region of a size that depends on the level of generality of the classifier's condition. Perhaps the closest resemblance to such a function approximator in RL is soft state aggregation. Ordinary state aggregation is a simple form of generalising function approximator where states are grouped together with one table entry used for each group [28]. In soft state aggregation, the states are mapped into $M > 0$ aggregates or clusters from cluster space $\mathcal{X}$, where each state s belongs to cluster x with probability $P(x|s)$ [24]. This allows each state to be covered by several clusters. The value of a cluster generalises to all states in proportion to the clustering probabilities.

Agglomeration of classifier predictions in XCS can be seen as soft state aggregation with one cluster being represented by one classifier. The state–action value for state s and action a is given by the fitness–weighted prediction of all classifiers matching this state, or more formally

$$Q(s,a) = \frac{\sum_{cl \diamond s,a} p_{cl} f_{cl}}{\sum_{cl \diamond s,a} f_{cl}} \tag{2}$$

where the $\diamond$ operators indicates that for any classifier $cl \in \mathcal{P}$ its condition matches the current state s and the action it promotes matches the current action a[5], i.e. $cl \diamond s, a$ resembles the action set $\mathcal{A}$ matching state s for the chosen action a. In soft state agglomeration the action–value is given by the classifier prediction weighted by its probability to cover the state–action pair s, a, or formally $Q(s,a) = \sum_{cl \in \mathcal{P}} P(cl|s,a) p_{cl}$. That gives

$$P(cl|s,a) = \begin{cases} \frac{f_{cl}}{\sum_{cl \diamond s,a} f_{cl}} & \text{if } cl \diamond s, a \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

for the probability of a classifier belonging to a particular state–action pair. Using the above probabilities, one state of the classifiers of XCS can be mapped onto reinforcement learning with soft state aggregation.

2.3 Eligibility Traces

Eligibility Traces are a way of improving the speed of prediction update in reinforcement learning by performing action–value backups over more than one state. As shown in expression (1) standard Q–Learning, like any other TD(0) method, performs single backup, i.e. it bases the update magnitude on one next expected return. Monte Carlo methods, in contrast, perform value update only after the whole sequence of rewards are observed, which is at the end of one episode. Eligibility traces allow a mixture of TD(0) and Monte Carlo by basing prediction updates on n steps of real reward and the estimated value of the nth next state, all appropriately discounted [28].

In TD(λ), several backups of different lengths are mixed to form a complex backup, i.e. if the return of an n–step backup is given as $R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n})$, then TD($\lambda$) mixes these backups by weighting the n–step backup by λ^{n-1}, or more formally

$$R_t^{\lambda} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)} \tag{4}$$

Although it might not seem feasible to apply this approach, its implementation is rather trivial, as shown below.

From a different viewpoint, TD(λ) can be seen as a decaying log of visited state–action pairs. These can be implemented by adding an additional memory $e(s,a)$ to each state–action pair that gets refreshed every time this pair is visited and decays with every other step taken. Originally, traces were accumulating, i.e. every time the corresponding state–action pair was visited they were increased by

[5] For a more formal definition of matching of classifiers see [21]

1. Later, it was found that limiting the trace–magnitude to 1 (called *replacing traces*) gives better performance, in particular if some actions do not perform a state transition [25], or more formally

$$e_t(s,a) = \begin{cases} 1 & \text{if } s = s_t; a = a_t \\ \lambda\gamma e_{t-1}(s,a) & \text{otherwise} \end{cases} \tag{5}$$

for each s, a at each step. Different to reinforcement learning, XCS commonly deals with environments where $A(s)$ is not known and so $A = \bigcup_{s \in S} A(s)$ has to be assumed for each state. For this reason, replacing traces is a better choice for such environments.

To use eligibility traces in Q–Learning, (1) is changed to

$$Q(s,a) \leftarrow Q(s,a) + \beta\epsilon e(s,a) \forall s \in S, a \in A(s) \tag{6}$$

$$\text{with } \epsilon_t = r_{t+1} + \gamma \max_{a \in A(s)} Q(s_{t+1}, a) - Q(s_t, a_t) \tag{7}$$

where ϵ is the current error of expected return. At every step all traces decay by $e(s,a) \leftarrow \lambda\gamma e(s,a)$, as given in (5). Due to Q–Learning's off–policy nature, special care is required to only keep the traces as long as the learner follows a greedy policy. As soon as a non–optimal action (based on the state of current knowledge) is selected, all traces have to be set back to zero to avoid misleading value updates. This update procedure conforms to Watkins' Q(λ), as described in [30].

In XCS the current action–value is formed by the classifiers in the current action set $\mathcal{A}$. Therefore, each classifier eventually causes a different prediction error. Eligibility Traces require one error value ϵ to be propagated back to all classifiers with traces. The approach chosen here is to weight the error produced by each $cl \in \mathcal{A}$ by the macroclassifier's fitness[6], or

$$\bar{\epsilon} = \frac{\sum_{cl \in \mathcal{A}} (P - p_{cl}) f_{cl}}{\sum_{cl \in \mathcal{A}} f_{cl}} \tag{8}$$

This causes each classifier to contribute the same fraction to the overall error as it contributed to the prediction.

It is hypothesised that adding Eligibility Traces to XCS will increase the speed of convergence towards the correct predictions, similar to the benefits seen in Q(λ). What might interfere with this hypothetical speed increase is generalisation within the classifier's condition, which causes the classifier to cover several states that might be visited during one episode. Another influential factor is the mean error calculation, because it may be misleading to weight each classifier's error by its fitness which in the case of a new classifier will always be low.

3 Implementation

The effectiveness of eligibility traces is tested by adapting Barry's XCS v2.3, creating a new XCS(λ). His implementation is similar to the one described in [9] with the following differences:

- The covering operator only introduces a new classifier if $\mathcal{M} = \emptyset$. A new classifier is introduced at the end of each exploration step if $\sum_{cl \in \mathcal{M}} p_{cl} num_{cl} < \zeta p_I$, where ζ is the covering multiplier constant and usually set to 0.5. The new classifier matches the current message with "#"s introduced with probability $P_\#$ and promotes a random action not present in $\mathcal{M}$.

- The only form of subsumption used is between the parents and its children in the GA.

Traces are added by introducing an additional parameter e to each macroclassifier. Initially 0, at each occurrence of a classifier in $\mathcal{A}$ its trace is updated to $e_{cl} = 1$. At the end of each step, the traces of all classifiers in the population decay by $e_{cl} \leftarrow \lambda\gamma e_{cl}$. After each episode and whenever a non–greedy action is performed, all traces are reset back to zero.

The prediction update for $\mathcal{A}^{-1}$ is performed in the usual way. However, directly after updating $\mathcal{A}^{-1}$, the prediction of each classifier $cl \notin \mathcal{A}^{-1}$ is updated by $p_{cl} \leftarrow p_{cl} + \beta\bar{\epsilon}e_{cl}$, where $\bar{\epsilon}$ is given by (8). On encountering a terminal state, prediction update in $\mathcal{A}$ is performed in the same way as in $\mathcal{A}^{-1}$.

One point of inconsistency of this implementation is the incompatibility of the prediction update within $\mathcal{A}$ and the update through traces outside of $\mathcal{A}$. For classifiers in $\mathcal{A}$ the MAM update method [29] is applied, where unexperienced classifiers calculate the average of the previous and the current

[6] As noted in [19], the fitness already accounts for the numerosity, the count of equal classifiers within a population, in a macroclassifier.

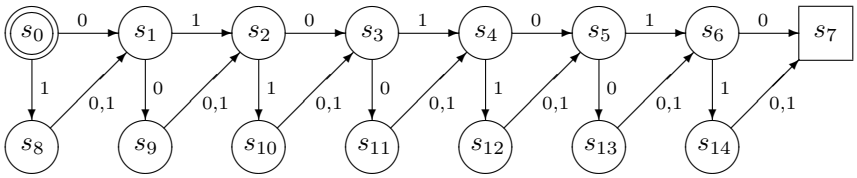
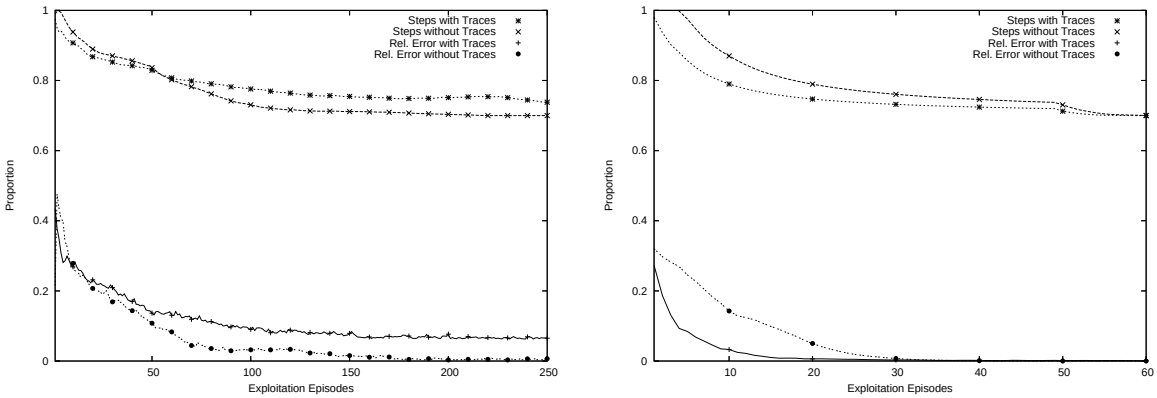


Figure 1: A 7–Step Finite State World. The learner always starts at state s_0. Each episode terminates when terminal state s_7 is reached. The states are encoded in binary by their state numbers using 4 bits.



(a) (b)

Figure 2: The number of steps per episode and the relative error w.r.t. the current prediction for XCS with and without traces. In (a) XCS starts with an empty population and uses covering, induction and deletion to find the optimal classifiers. (b) shows prediction learning in XCS when it starts with the optimal population and provides neither induction nor deletion. The episode steps are given by their moving average over 50 exploit episodes. All curves are averaged over 10 runs.

values and experienced classifiers use the delta–rule for its prediction update. This requires the number of updates to be stored within each macroclassifier. Update through traces facilitates the delta–rule independent of the classifier experience and causes an update error when mixed with the MAM method. This error, however, is negligible and does not influence the general finding of the experiments.

4 XCS(λ) Performance

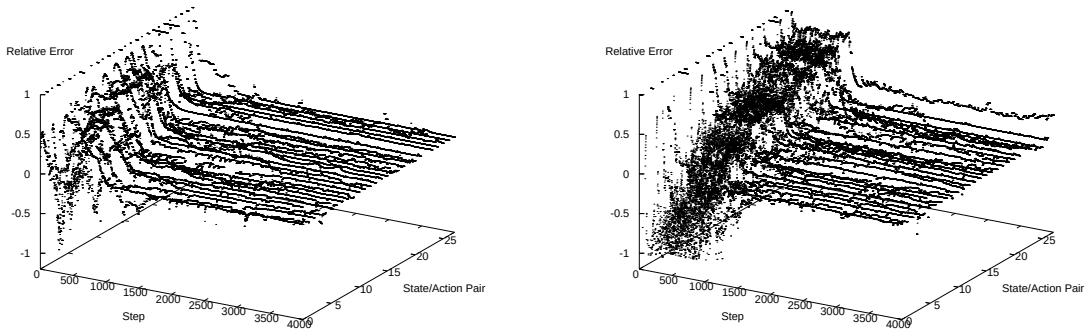
4.1 Experiments

To keep analysis simple, XCS(λ) is tested in a small, deterministic finite state world which is shown in Fig. 1, eliminating a number of confounding variables that exist in Woods–like environments [4]. The learner has to find the shortest sequence of actions from state s_0 to the terminal state s_7, where a reward of 1.0 is given. All states are encoded in 4–bit binary, allowing for generalisation of state n and $n + 1000_2$ without introducing any error. The optimal population $\mathcal{O}$ therefore consists of 21 classifiers. Optimal generalisation covers a fraction of $P_{\#\mathcal{O}} = 0.083$ of all condition alleles of $\mathcal{O}$. The minimum number of steps is 7.

The XCS parameters are set to the following values: $N = 336$, $\beta = 0.2$, $\alpha = 0.1$, $\epsilon_0 = 0.01$, $\nu = 5.0$, $\gamma = 0.71$, $\Theta_{GA} = 25$, $\chi = 0.8$, $\mu = 0.04$, $\Theta_{del} = 20$, $\delta = 0.1$, $\Theta_{sub} = 20$, $P_\# = 0.2$, $p_I = 0.01$, $\epsilon_I = 0.01$, $f_I = 0.01$, $\zeta = 0.5$. Explore and exploit episodes follow each other alternately, performance is only reported in exploit mode. For XCS(λ) the additional trace decay constant is set to $\lambda = 0.9$. It is assumed that $\mathcal{O}$ is not known in advance and so $P_\#$ is kept high when compared to $P_{\#\mathcal{O}}$ although this might influence the results negatively.

To honour prediction errors independent of their position in their environment, the relative error w.r.t. the current prediction or payoff[7] or the relative error w.r.t. the optimal prediction are reported.

[7]This method was first proposed in [4]



(a) (b)

Figure 3: (a) shows the relative error w.r.t. the optimal prediction for each state–action pair for the first 4000 steps, using standard XCS. The state–action pairs are shown in the sequence $s_0/0$, $s_0/1$, $s_8/0$, $s_8/1$, $s_2/0$, $s_2/1$, $s_9/0$, $s_9/1$, ..., $s_{14}/0$, $s_{14}/1$. (b) shows the same results for XCS(λ).

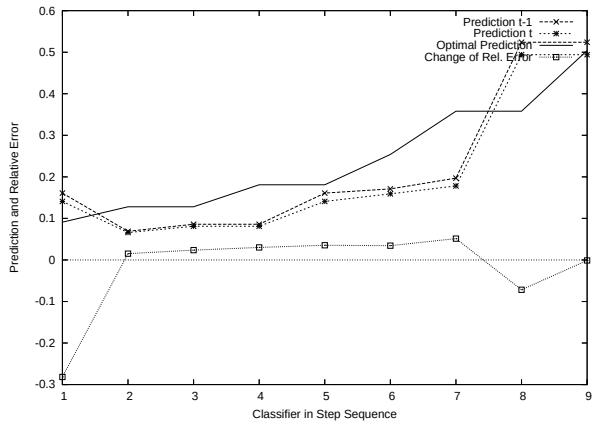


Figure 4: The prediction per state–action pair before and after the prediction update. The change of relative error shows how the relative prediction error w.r.t. the optimal prediction changes with the prediction update. The sequence of state–action pairs is given in the text.

Figure 2 shows a comparison of steps per episode and relative error w.r.t. the current prediction for XCS and XCS(λ). It can be clearly seen in (a) that although traces cause a faster decrease of error and episode steps in the early stages, they perform worse by the time that standard XCS converges to the optimal policy. However, when starting XCS with the optimal population and deactivating any covering or induction (see (b)), traces show the expected effect of improved prediction update.

A comparative plot of the relative error w.r.t. the optimal prediction for each state–action pair is shown in Fig. 3. One feature that is already visible in Fig. 2 but can also be seen in this plot, is that standard XCS in (a) reduces the relative error faster that XCS(λ) in (b). Another interesting effect is that for XCS(λ) the relative error moves below zero very rapidly, particularly for the early states in the sequence. This indicates that the combination of classifiers for these state–action pairs predict a value that is higher than the optimal value – for the first few states even double the optimal value.

4.2 Analysis

Although it seems intuitive that adding traces to XCS should increase the speed of learning as they do in RL, they seem to have the opposite effect. A comparison of the graphs in Fig. 2 suggests that the reduction of performance is caused by the covering and induction operator. A more detailed analysis of one prediction update in an early episode allows insight into why traces as implemented above do not improve performance.

To examine this further, let us consider an example transition, in this case in the form of a prediction update in episode 7 (explore) after the learner has performed the following sequence of steps: $s_0/1 \rightarrow$

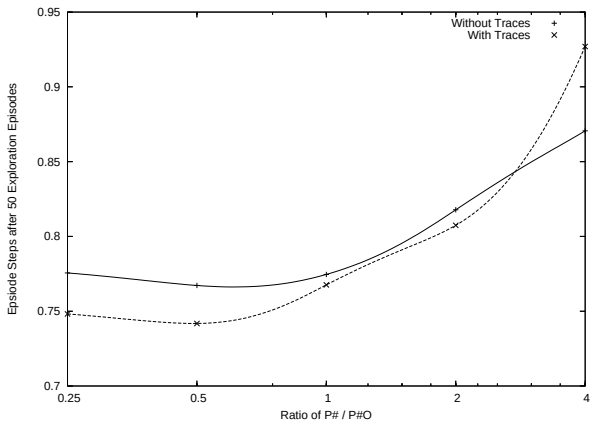


Figure 5: Change of the average steps per episode after 50 exploitation episodes by changing of the parameter $P_\#$. $P_\#$ is displayed as the ratio of $P_\#$ to $P_{\#\mathcal{O}}$. The steps per episodes are averaged over 10 runs.

$s_8/0 \rightarrow s_1/0 \rightarrow s_9/0 \rightarrow s_2/1 \rightarrow s_{10}/1 \rightarrow s_3/1 \rightarrow s_4$. The chosen action is 1, which lets the learner proceed to state s_{12}. Figure 4 shows the prediction for each state–action pair in the performed sequence before and after the prediction update caused by the transition to state s_{12}. The state–action pairs $s_4/1$ and $s_{12}/1$ are exclusively covered by an over–general classifier $\#1\#\#/1$, which causes their prediction to be equal. As visualised in Fig. 4, the prediction for $s_4/1$ is higher than the optimal prediction, which was caused by an earlier update in $s_{12}/1$ with a higher value. This causes $s_4/1$ to produce a negative error at the current update which is reasonable for $s_4/1$ due to its too high prediction. However, propagating this error back through all previously visited states causes their prediction also to be lowered. This back–propagation causes a drop in performance for 6 out of 7 states and subsequently reduces overall prediction quality.

Clearly, traces fail as soon as over–general classifiers cause negative prediction errors to occur. Updating the prediction in a state closer to the terminal state can cause the prediction of a state–action pair with a lower optimal value to overshoot this optimal value. Hence, negative errors emerge from classifiers that cover state–action pairs with different optimal values. When traces are applied, such negative errors are then propagated back to all classifiers in the action sequence and reduce their prediction. If their prediction was lower than or equal to the optimal prediction – which is the case for all less than optimally or optimally general classifiers – their prediction is degraded. This effect is strongest when the over–general classifier covers several states close to the terminal state where the difference in magnitude of the optimal value is highest between the states.

Experimental validation for this explanation is given in Fig. 3b where the predictions for early states in the chain show a negative error of high magnitude in early runs. These result from either over–general classifiers that cover these early states and other states closer to the terminal state or from traces that propagate negative errors back through the action chain. The former also explains the minor negative error in Fig. 3a for standard XCS but the significant reduction of performance in XCS(λ) can only be accounted for by the latter.

If negative error back–propagation is found to cause such a loss of performance, one could argue to avoid negative errors by taking the prediction P instead of the mean error $\bar{\epsilon}$ to perform update in all classifiers with traces. This would bypass the problem of how to define the mean error and subsequently avoids the back–propagation of negative errors. This, however, is a fallacy as the value of earlier states is always based on later states in the action–chain, which can, due to over–generals, still cause a too high prediction and subsequent reduction of performance. The only viable solution would be exclusive use of traces when a terminal state is reached (and the external reward is guaranteed to be correct) but experiments revealed only a minor speed increase that does not compare with speed decrease caused by the maintenance of the traces themselves.

Another workaround for the loss in performance is a reduction of generality as controlled by $P_\#$ to avoid the high amount of over–general classifiers. Indeed, a reduction of $P_\#$ can improve the performance of XCS(λ) significantly as visualised in Fig. 5. This graph shows that $P_\# < P_{\#\mathcal{O}}$ causes XCS(λ) to evolve a correct policy faster than standard XCS. However, at $P_\# > 3P_{\#\mathcal{O}}$ the increased introduction of over–general classifiers reduces the performance of XCS(λ) visibly below that of the standard system. With these findings one might want to suggest that keeping $P_\#$ low would solve

the problem. Yet, a too low $P_{\#}$ introduces too many over–specific classifiers that cause early random deletion due to population pressure, as identified in [8]. The result is an infinite "covering – random deletion" cycle with sub–optimal performance.

5 Discussion and Conclusion

It has been shown that XCS(λ) does perform worse than standard XCS if over–general classifiers reduce performance due to negative error back–propagation. Setting coverage and mutation generality $P_{\#}$ close to or lower than the generality of $\mathcal{O}$ avoids this effect but can cause XCS to be trapped in an infinite "covering – random deletion" cycle. As in a real–word setting $P_{\#\mathcal{O}}$ is not known, so no recommendations can be given for the setting of $P_{\#}$. Due to such sensitivity to system parameters and the increased effort of implementation, additional storage requirements and increased runtime, adding Eligibility Traces to XCS does not justify the effort.

Recently it has been shown [7] that adjusting the prediction update to reflect on gradient descent can significantly improve the speed and accuracy of learning the correct prediction values. This improvement is achieved by weighting the prediction update in addition to the learning rate β by the classifier's fitness w.r.t. the fitness of $\mathcal{A}$, or $\frac{F_{cl}}{\sum_{cl \in \mathcal{A}^{-1}} F_{cl}}$. Therefore, high–fitness classifiers will converge to the correct prediction quicker, while low–fitness classifiers will keep their errors high. It is questionable if this extension would improve the performance of XCS(λ), as it will not be able to avoid over–general classifiers. In addition, fitness–weighting prediction updates for classifiers with traces require to additionally store $\sum_{cl \in \mathcal{A}} F_{cl}$ for the last $\mathcal{A}$ the classifier appeared in, which introduces additional bookkeeping.

Over–general classifiers cannot be avoided in the trial–and–error approach that characterises the reinforcement learning framework. If the optimal values of all state-action pairs are larger than the initial prediction of a classifier, then any prediction update to a lower classifier prediction is a strong indicator for an over–general classifier. Such information could be used in further XCS development to quickly identify such overly general classifiers and remove them from the population. The problem with such an approach is that before the too–high prediction was identified, it has already been used to update the prediction of the previous classifier in the action–chain. As a result, the previous classifier's prediction could also overshoot its optimal value and would incorrectly be removed at its next update. Another way is to initially keep more general classifiers in a "nursery" that allows prediction update but no contribution to the final prediction value. Only if a nursery's classifier did not cause a negative error after a certain amount of updates will it act as a part of the population. This again would allow the addition of traces, as over–general classifiers are much less likely to disrupt the prediction update. Still, the impact of such a nursery on overall XCS performance is unclear and remains topic of further investigation.

In the light of the results of XCS(λ) it seems surprising that RL with FA is still able to improve performance when using traces (e.g. [27]). The most obvious difference in how XCS and RL perform their searches is how they traverse the error surface. Function approximation methods in RL either operate on fixed generalisation (e.g. by tiling) or adjust generalisation while also updating the value function. Only in the latter case could generalisation be an issue when using traces. Still, generalisation is usually performed smoothly, by descending in the direction of the negative gradient of the error surface. Adaptive State Aggregation (ASA, [24]), for example, applies soft state aggregation and adjusts the cluster probability distribution softly over all states by minimising the Bellman error. The GA in XCS with ternary condition coding, in contrast, treats state coverage as a switch rather than a degree. Therefore, when mapping state coverage of XCS to soft state aggregation (as demonstrated in section 2.2), its probability distribution per classifier changes with a much higher degree of discontinuity when compared to ASA. Eligibility Traces propagate errors back to states that have been visited before. Rapid changes cause higher errors and their propagation results in bigger disruption of the system.

By interpreting XCS as a reinforcement learner with function approximation, the genetic algorithm performs the function of searching possible ways of covering the action–value function. Currently this is done by flipping bits and introducing a "don't care" symbol to allow some form of generalisation. It might be possible to find other approaches to perform this search, possibly by further investigating recent developments in RL, with softer adaptation of how the search space is traversed and still reach a global optimum. Naturally, that would require another way of expressing generalisation and as such deviation from the binary representation. If soft adaptation can be achieved, further investigation of performance improvement with eligibility traces should be conducted.

References

[1] L. C. Baird. Residual algorithms: Reinforcement learning with function approximation. In *Machine Learning: Proceeding of the Twelfth International Conference*. Morgan Kaufmann Publishers, July 1995.

[2] L. C. Baird. *Reinforcement Learning Through Gradient Descent*. PhD thesis, School of Computer Science. Carnigie Mellon University, Pittsburgh, PA 15213, 1999.

[3] A. Barry. Limits in Long Path Learning with XCS. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 1832–1843. Springer-Verlag, 2003.

[4] A. M. Barry. *XCS Performance and Population Structure within Multiple-Step Environments*. PhD thesis, Queens University Belfast, September 2000.

[5] A. M. Barry. The Stability of Long Action Chains in XCS. *Journal of Soft Computing*, 6(3–4):183–199, 2002.

[6] E. Bernadó, X. Llorà, and J. M. Garrell. XCS and GALE: a Comparative Study of Two Learning Classifier Systems with Six Other Learning Algorithms on Classification Tasks. In *Proceedings of the 4th International Workshop on Learning Classifier Systems (IWLCS-2001)*, pages 337–341, 2001.

[7] M. V. Butz, D. E. Goldberg, and P. L. Lanzi. Gradient Descent Methods in Learning Classifier Systems: Improving XCS Performance in Multistep Problems. Technical Report 2003028, Illinois Genetic Algorithms Laboratory, December 2004.

[8] M. V. Butz, T. Kovacs, P. L. Lanzi, and S. W. Wilson. How XCS Evolves Accurate Classifiers. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *GECCO-2001: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 927–934. Morgan Kaufmann, 2001.

[9] M. V. Butz and S. W. Wilson. An Algorithmic Description of XCS. Technical Report 2000017, Illinois Genetic Algorithms Laboratory, 2000.

[10] P. Cichosz and J. J. Mulawka. Fast and Efficient Reinforcement Learning with Truncated Temporal Differences. In *International Conference on Machine Learning*, 1995.

[11] P. Cichosz and J. J. Mulawka. Faster Temporal Credit Assignment in Learning Classifier Systems. In *Proceedings of the First Polish Conference on Evolutionary Algorithms*, 1996.

[12] M. Compiani, D. Montanari, R. Serra, and P. Simonini. Learning and Bucket Brigade Dynamics in Classifier Systems. *Special issue of Physica D (Vol. 42)*, 42:202–212, 1990.

[13] L. Davis, C. Fu, and S. W. Wilson. An Incremental Multiplexer Problem and Its Uses in Classifier System Research. In P. L. Lanzi, W. Stolzmann, and S. W. Wilson, editors, *Advances in Learning Classifier Systems*, volume 2321 of *LNAI*, pages 23–31. Springer-Verlag, Berlin, 2002.

[14] M. Dorigo and M. Colombetti. *Robot Shaping: An Experiment in Behavior Engineering*. MIT Press/Bradford Books, 1998.

[15] J. H. Holland. Properties of the Bucket Brigade. In J. J. Grefenstette, editor, *Proceedings of the 1st International Conference on Genetic Algorithms and their Applications (ICGA85)*, pages 1–7. Lawrence Erlbaum Associates: Pittsburgh, PA, July 1985.

[16] J. H. Holland, K. J. Holyoak, R. E. Nisbett, and P. R. Thagard. *Induction: Processes of Inference, Learning, and Discovery*. MIT Press, Cambridge, 1986.

[17] J. H. Holland and J. S. Reitman. Cognitive systems based on adaptive algorithms. In D. A. Waterman and F. Hayes-Roth, editors, *Pattern-directed Inference Systems*. New York: Academic Press, 1978.

[18] J. H. Holmes. A genetics-based machine learning approach to knowledge discovery in clinical data. *Journal of the American Medical Informatics Association Supplement*, 1996.

[19] T. Kovacs. Evolving Optimal Populations with XCS Classifier Systems. Master's thesis, School of Computer Science, University of Birmingham, Birmingham, U.K., 1996.

[20] P. L. Lanzi. A Study of the Generalization Capabilities of XCS. In T. Bäck, editor, *Proceedings of the 7th International Conference on Genetic Algorithms (ICGA97)*, pages 418–425. Morgan Kaufmann, 1997.

[21] P. L. Lanzi. Learning Classifier Systems from a Reinforcement Learning Perspective . Technical Report 00-03, Dipartimento di Elettronica e Informazione, Politecnico di Milano, 2000.

[22] A. Parodi and P. Bonelli. The Animat and the Physician. In J. A. Meyer and S. W. Wilson, editors, *From Animals to Animats 1. Proceedings of the First International Conference on Simulation of Adaptive Behavior (SAB90)*, pages 50–57. A Bradford Book. MIT Press, 1990.

[23] R. L. Riolo. Bucket Brigade Performance: I. Long Sequences of Classifiers. In J. J. Grefenstette, editor, *Proceedings of the 2nd International Conference on Genetic Algorithms (ICGA87)*, pages 184–195, Cambridge, MA, July 1987. Lawrence Erlbaum Associates.

[24] S. P. Singh, T. Jaakkola, and M. I. Jordan. Reinforcement Learning with Soft State Aggregation. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 361–368. The MIT Press, 1995.

[25] S. P. Singh and R. S. Sutton. Reinforcement Learning with Replacing Eligibility Traces. *Machine Learning*, 22(1–3):123–158, 1996.

[26] R. E. Smith, B. A. Dike, R. K. Mehra, B. Ravichandran, and A. El-Fallah. Classifier Systems in Combat: Two-sided Learning of Maneuvers for Advanced Fighter Aircraft. *Computer Methods in Applied Mechanics and Engineering*, 186(2–4):421–437, 2000.

[27] R. Sutton. Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding. *Advances in Neural Information Processing Systems*, 8:1038–1044, 1996.

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

[29] G. Venturini. *Apprentissage Adaptatif et Apprentissage Supervisé par Algorithme Génétique.* PhD thesis, Université de Paris-Sud, 1994.

[30] C. J. Watkins. *Learning from delayed rewards.* PhD thesis, University of Cambridge, Psychology Department, 1989.

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

[32] S. W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

[33] S. W. Wilson. Generalization in the XCS Classifier System. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 665–674. Morgan Kaufmann, 1998.

[34] S. W. Wilson. Mining Oblique Data with XCS. Technical Report 2000028, University of Illinois at Urbana-Champaign, 2000.