

XCSR with Computed Continuous Action

Muhammad Iqbal, Will N. Browne, and Mengjie Zhang

Evolutionary Computation Research Group,
School of Engineering and Computer Science,
Victoria University of Wellington,
PO Box 600, Wellington 6140, New Zealand
{muhammad.iqbal,will.browne,mengjie.zhang}@ecs.vuw.ac.nz

Abstract. Wilson extended XCS with interval based conditions to XCSR to handle real valued inputs. However, the possible actions must always be determined in advance. Yet domains such as robot control require numerical actions, so that neither XCS nor XCSR with their discrete actions can yield high performance. In the work presented here, genetic programming-based representation is used for the first time to compute continuous action in XCSR. This XCSR version has been examined on a simple one-dimensional but non-linear testbed problem – the “frog” problem – and compared with two continuous action based systems, GCS and XCSFCA. The proposed approach has consistently solved the frog problem and outperformed GCS and XCSFCA.

Keywords: Learning Classifier Systems, XCS, XCSR, Code Fragment, Computed Action, Continuous Action.

1 Introduction

A learning classifier system (LCS) represents a rule-based agent that incorporates evolutionary computing and machine learning to solve a given task by interacting with the environment. The rules are of the form “if *state* then *action*”. Usually the possible number of actions are known in a learning classifier system, but for problems requiring continuous real valued outputs, it is not possible. Therefore discrete action based systems like XCS and XCSR show some limits. A solution is to use a generalized classifier system (GCS) [20] in which the input x is linked to the action a in the process of matching the environmental instance. The difficulty of GCS is to evolve a condition form $t(x, a)$ so that the action is continuous. Tran et al. [14] implemented XCSF [19] having computed continuous actions, named as XCSFCA, where the action is computed directly as a linear combination of the input state and a vector of action weights. XCSFCA has outperformed GCS and produced very promising results for the frog problem, but could not achieve the 100% performance.

In this paper, a new approach will be investigated in which the discrete action in XCSR is replaced by a code fragment. A code fragment is a tree-expression, similar to a tree generated in genetic programming. The action value is determined using the input state values as terminals of the code fragmented action.

Thus the action is continuous with respect to the input state in this version of XCSR, named XCSRCFA (*XCSR with Code Fragmented Action*). The proposed approach will be examined and compared with GCS and XCSFCA on the frog problem described in [20].

The rest of the paper is organised as follows. Section 2 describes genetic programming, learning classifier systems, accuracy based learning classifier systems (XCS), and real valued XCS (XCSR). In section 3 the novel implementation of XCSR using code fragmented actions is detailed. Section 4 describes the frog problem and experimental setup used in the experimentation. In section 5 experimental results are presented and compared with GCS and XCSFCA. In the last section this work is concluded and the future work is outlined.

2 Background

2.1 Genetic Programming

Commonly, genetic programming (GP) uses a rich alphabet to encode the solution, i.e. symbols that can express functions as well as numbers. A GP like alphabet to describe the problem is used in the LCS developed here, so the GP technique is described to aid understanding.

GP is an evolutionary approach to generating computer programs for solving a given problem automatically [10]. The task to be solved is represented by a primitive set of operations, known as the *function set*, and a set of operands, known as the *terminal set*.

The GP generated computer program is normally represented as a tree, which may contain unnecessary terms (bloat). The bloat problem in GP is usually addressed by limiting maximal allowed depth for an individual tree and/or using a fitness measure that punishes excess sized individuals [12], or using specific bloat control operators [2]. LCSs are a well structured rule-based learning technique that have various evolutionary pressures [4] to produce maximally general, optimal and accurate classifier rules, implicitly avoiding bloat.

A GP system produces a tree as a ‘single’ solution, rather than a co-operative set of rules as in an LCS. It generally requires supervised learning with the whole training set, rather than on-line, reinforcement learning as in LCS.

2.2 Learning Classifier Systems

Traditionally, an LCS represents a rule-based agent that incorporates evolutionary computing and machine learning to solve a given task, enacting in an unknown environment via a set of sensors for input and a set of effectors for actions. The rules are of the form “if *state* then *action*”. After observing the current state of the environment, the agent performs an action, and the environment provides a reward, as depicted in Figure 1.

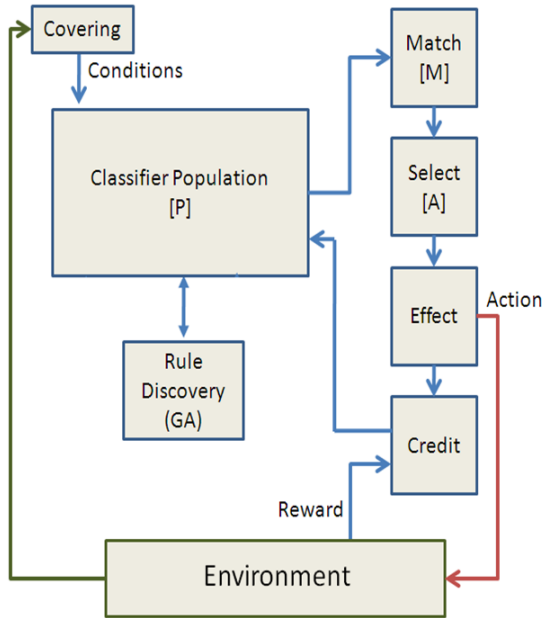


Fig. 1. Schematic depiction of a learning classifier system. In the proposed approach of computed continuous action, a classifier will become a member of M if its condition part matches the input as usual and its action belongs to the allowed action range. The action set $[A]$ is governed by the classifiers in $[M]$ that have the highest fitness-weighted average prediction value, see section 3.

Accuracy Based Learning Classifier System. XCS is a formulation of LCS that uses accuracy-based fitness to learn the problem by forming a complete mapping of states and actions to rewards.¹ In XCS, the learning agent evolves a population $[P]$ of classifiers, where each classifier consists of a rule and a set of associated parameters estimating the quality of the rule. Each rule is of the form ‘if *condition* then *action*’, having two parts: a condition and the corresponding action. Commonly, the condition is represented by a fixed length bitstring defined over the ternary alphabet $\{0, 1, \#\}$, and the action is represented by a numeric constant.

Each classifier has three main parameters: (1) prediction p , an estimate of the payoff that the classifier will receive if its action is selected, (2) prediction error ϵ , which estimates the error between the classifier’s prediction and the received payoff, and (3) fitness F , computed as an inverse function of the prediction error. In addition, each classifier keeps an experience parameter *exp*, which is a count of the number of times it has been updated, and a numerosity parameter n , which is a count of the number of copies of each unique classifier.

¹ For a detailed review of different types and approaches in LCS refer to [15].

The agent has two modes of operation, explore (training) and exploit (application). In the following, XCS operation is concisely described. For a complete description, the interested reader is referred to the original XCS paper by Wilson [16], and to the algorithmic details by Butz et al. [5].

In the explore mode the agent attempts to obtain information about the environment and describe it by creating the decision rules:

1. observes the current state of the environment, $s \in S$ where S is the set of all possible environmental states. The current state s is usually represented by a fixed length bitstring defined over the binary alphabet $\{0, 1\}$.
2. selects classifiers from the classifier population $[P]$ that have conditions matching the state s , to form the match set $[M]$.
3. performs covering: for every action $a_i \in A$ in the set of all possible actions, if a_i is not represented in $[M]$ then a random classifier is generated with a given generalization probability such that it matches s and advocates a_i , and added to the population (termed covering)².
4. forms a system prediction array $P(a_i)$ for every $a_i \in A$ that represents the system's best estimate of the payoff should the action a_i be performed in the current state s . Commonly, $P(a_i)$ is a fitness weighted average of the payoff predictions of all classifiers advocating a_i .
5. selects an action a to explore (probabilistically or randomly) and selects all the classifiers in $[M]$ that advocated a to form the action set $[A]$.
6. performs the action a , records the reward r from the environment, and uses r to update the associated parameters of all classifiers in $[A]$.
7. when appropriate, implements rule discovery by applying an evolutionary mechanism, commonly a genetic algorithm (GA), in the action set $[A]$ to introduce new classifiers to the population.

Additionally, the explore mode may perform subsumption, to merge specific classifiers into any more general and accurate classifiers.

In contrast to the explore mode, in the exploit mode the agent does not attempt to discover new information and simply performs the action with the best predicted payoff. The exploit mode is also used to test learning performance of the agent in the application.

Various GP-based representations have been investigated to represent high level knowledge in LCS in an attempt to obtain compact classifier rules [6], to reach the optimal performance faster [11], to develop useful feature extractors [1], and to extract building blocks of knowledge [8,7]. In the work presented here, GP-based representation is used for the first time to compute continuous action in LCS.

Real Valued XCS (XCSR). The changes to XCS for real inputs were as follows [17,18]. The classifier condition was changed from a string from $\{0, 1, \#\}$

² If the classifier population size grows larger than the specified limit, then one of the classifier rules has to be deleted so that the new rule can be inserted.

to a concatenation of interval predicates, $int_i = (l_i, u_i)$, where l_i (“lower”) and u_i (“upper”) are real values. A classifier matches an input message x if each element x_i belongs to the corresponding interval predicate, i.e. $l_i \leq x_i \leq u_i$. When a new covered classifier is created, each interval predicate $int_i = (l_i, u_i)$ is generated as $l_i = x_i - rand(r_0)$ and $u_i = x_i + rand(r_0)$, where $rand(r_0)$ is a value uniform randomly from $[0, r_0]$ and r_0 is a real constant.

The GA works as in XCS. Crossover (with probability χ) permutes alleles of two parents between two crossover points. Since an allele is a real value, a new mutation operator is introduced. Mutation (with probability μ) modifies an allele by adding an amount $\pm rand(m_0)$ where m_0 is a real constant. A classifier *clfr1* can subsume another classifier *clfr2* if every interval predicate in *clfr1*’s condition is more general than the corresponding predicate in *clfr2*’s condition.

Covering and mutation operators can generate intervals out of the condition range. If it happens, l_i and u_i of an interval predicate int_i are brought back to the extremes of the condition range and they are possibly permuted in order to respect the predicate constraint $l_i \leq u_i$.

XCSR can handle real valued inputs using interval based conditions, but the possible actions still need to be determined in advance. Yet domains such as robot control require numerical actions, so that neither XCS nor XCSR with their discrete actions can yield high performance. In the work presented here, the discrete action in XCSR is replaced by a continuous action represented as a genetic programming-tree like code fragment. We have also implemented a discrete code fragmented action in XCS (while using the ternary alphabet based conditions in the classifier rules), to successfully evolve compact and optimal populations for multiplexer domain problems [6].

3 XCSR with Code Fragmented Action (XSRCFA)

In the work presented here, the discrete action in XCSR is replaced by a continuous action represented as a code fragment while using the interval based conditions in the classifier rules. Each code fragment is a binary tree of depth up to two, which was set to limit the tree size. A binary tree of depth two can have maximum seven nodes. The function set for the tree is problem dependent such as $\{+, -, *, / \dots\}$ for linear regression problems, and $\{AND, OR, NAND, NOR \dots\}$ for binary classification problems as is the simple standard set in GP domain [10]. The terminal set is $\{ERC, D0, D1, D2, \dots, Dn - 1\}$ where *ERC* is an ephemeral random constant and n is the length of condition in a classifier rule.

The action value of a classifier is determined by evaluating the action code tree. The action code tree is evaluated by replacing the terminal symbols with the corresponding environmental instances’ values. Thus the action is continuous with respect to the input state in XSRCFA. For example, consider the code fragmented action shown in Figure 2. The internal nodes of the tree are functions and leaves are the terminals. In this action tree, 1.0 is an *ERC* and *D0* is the environmental input. If the input value is 0.4 then the action value will be 0.6 for this code fragmented action.

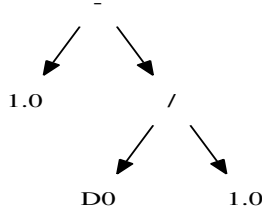


Fig. 2. A sample code fragmented action. Here 1.0 is an *ERC* and *D0* is the environmental input. If the input value is 0.4 then the action value will be 0.6 for this code fragmented action.

A classifier will become a member of the match set $[M]$ if its condition part matches the input as usual and its computed action belongs to the allowed action range. A pure random exploration is usually used for exploring actions of the environment, but in XCSRCFA it is replaced by a deterministic strategy, i.e. the action with the highest fitness-weighted average prediction value is selected. The classifiers advocating the best action will survive while GP-based crossover and mutation operations will allow the system to explore actions.

If the match set $[M]$ is empty then a new covered classifier is generated with a random code fragmented action that outputs an allowed action value against the current environmental input state s .

When the GA is applied in the action set $[A]$ to produce two offspring, action trees of the offspring are generated by crossing over the action trees of the parents. Then the action trees of the offspring are mutated with a given probability, to replace a subtree of the action with a randomly generated subtree of depth up to 1. The prediction of the offspring is set to the average of the parents' values whereas the prediction error and fitness of the offspring are set to the average of the parents' values reduced by constants *predictionErrorReduction* and *fitnessReduction* respectively, as in [3].

As the frog problem is one-dimensional, therefore the recombination of alleles of two parents does not generate new values of alleles. The system cannot explore values near the parents values, and thus appropriate conditions are not generated. To solve this problem, a mix crossover [14] is used.

It is to be noted that advantages of subsumption deletion are lost due to genotypic differences resulting in subsumption not occurring despite phenotypically similar behaviour. Subsumption deletion is made possible, albeit still problematic, by matching the action code on a character by character base.

4 The Problem Domain and Experimental Setup

4.1 The Frog Problem

In the frog problem, a system (frog) senses an object (fly) via a signal that monotonically decreases with the distance between them [20]. The frog should

learn to catch the fly in one jump. Let d ($0.0 \leq d \leq 1.0$) be the frog's distance from the fly. For simplicity, it is assumed that the frog's sensory input x , falls linearly with distance d , as given in Equation 1:

$$x(d) = 1 - d \quad (1)$$

On receiving the signal input x , the frog jumps a certain distance a as an action. A jump can be over or under the fly. Then the frog receives a payoff related to the remaining distance, as given in Equation 2:

$$P(x, a) = \begin{cases} x + a & \text{if } x + a \leq 1 \\ 2 - (x + a) & \text{otherwise} \end{cases} \quad (2)$$

It is to be noted that the payoff function is continuous and nonlinear – albeit composed of two linear planes. To solve the frog problem, a system must learn to choose, given x , the value of a corresponding to maximum payoff.

4.2 Experimental Setup

The system uses the following, commonly used in the literature, parameter values as suggested by Butz et al. in [3,5]: fitness fall-off rate $\alpha = 0.1$; prediction error threshold $\epsilon_0 = 0.01$; fitness exponent $\nu = 5$; learning rate $\beta = 0.2$; threshold for GA application in the action set $\theta_{GA} = 25$; experience threshold for classifier deletion $\theta_{del} = 20$; fraction of mean fitness for deletion $\delta = 0.1$; classifier experience threshold for subsumption $\theta_{sub} = 20$; crossover probability $\chi = 0.8$; mutation probability $\mu = 0.04$; $r_0 = 0.1$; $m_0 = 0.1$; initial prediction $p_I = 0.01$; initial prediction error $\epsilon_I = 0.0$; initial fitness $F_I = 0.01$; reduction of the prediction error *predictionErrorReduction* = 0.25; reduction of the fitness *fitnessReduction* = 0.1; and the selection method is tournament selection with tournament size ratio 0.4. The action of a newly created classifier by GA invocation is mutated with probability 0.2. Both GA-subsumption and action set subsumption are activated. The range of ERC is $[-2, 2]$ and the function set is $\{+, -, *, /\}$. The allowed action range is $[0, 1]$. The number of micro classifiers used is $N = 2000$. Explore and exploit problems are alternated with probability 0.5. All the experiments have been repeated 30 times with a known different seed in each run. One run is stopped after 100,000 explore problems.

5 Results

5.1 Performance

The results of the frog problem using GCS, XCSFCA, and the proposed approach XCSRCFA are shown in Figure 3³. The system error measures the difference between the prediction of the expected payoff and the payoff received. The payoff and system error curves are plotted by using a 50-point running average from

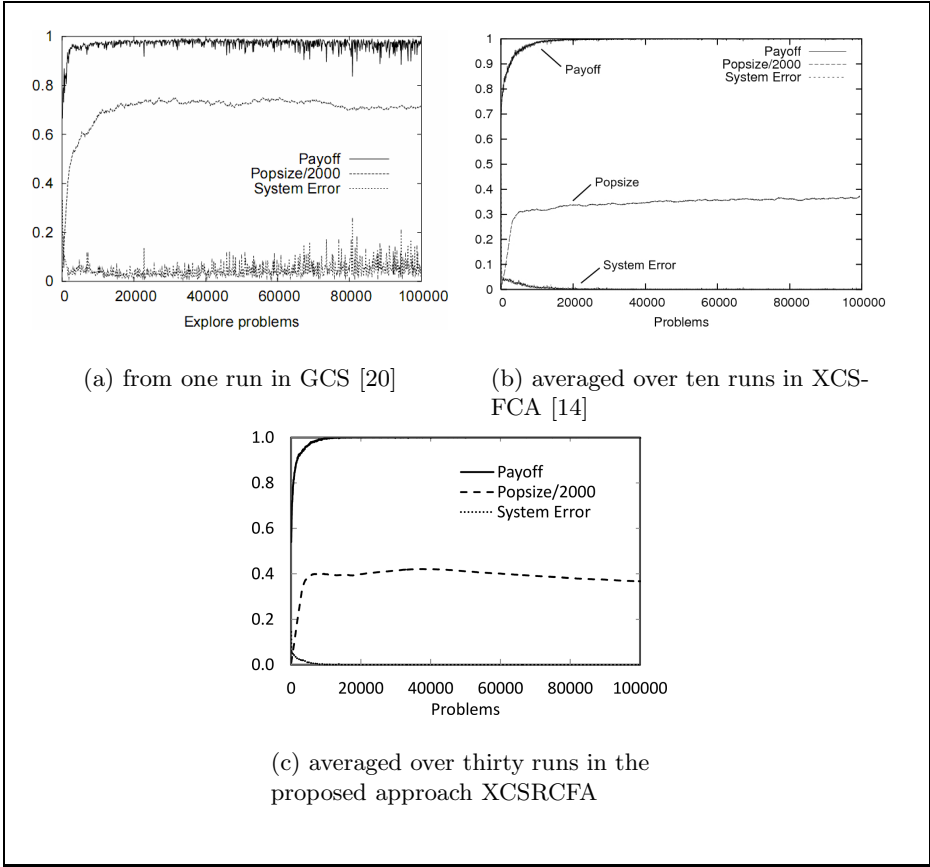


Fig. 3. Results of the frog problem in terms of payoff, system error, and population size using GCS, XCSFCA, and the proposed approach XCSRCA (curve order same as in legend). (a) one run in GCS [20], (b) averaged over ten runs in XCSFCA [14], and (c) averaged over thirty runs in XCSRCA.

exploit problems. In each problem of an experiment, the fly was placed at a random distance d ($0.0 \leq d \leq 1.0$) from the frog.

The payoff curve in Figure 3(a) shows that the performance of GCS was volatile. Although the performance did rise quickly to receive payoff greater than 0.95, it did not reach optimal performance (payoff 1.0). The system error curve approximately complements the payoff curve, so the system error for GCS is approximately 0.05 throughout the learning process. The population size curve for GCS rises to about 70% of N and then declines very gradually. The payoff curve in Figure 3(b) shows that the performance of XCSFCA is greater than 99% after an averaged number of 30,000 problems, but could not achieve stabilized

³ The results of GCS and XCSFCA were taken from [20] and [14] respectively, with permission from authors and copyright holders.

100% performance. The system error drops to smaller than 1%. The population size of classifiers is about 37% of N . The payoff curve in Figure 3(c) shows that the performance of the proposed approach XSRCFA reached 100% after an averaged number of 18,000 problems and did not decline after that. The population size of classifiers is about 37% of N .

Preen et al. [13] conducted experimentation on the frog problem using fuzzy dynamical genetic programming in XCSF and reported greater than 99% performance after an averaged number of 4,000 problems, but could not achieve stabilized 100% performance. The reported population size of classifiers is around 0.08% of N .

5.2 Best Action

If the frog wants to catch the fly at d , it must jump a distance $a = d = 1 - x$. Since d is generated randomly from $[0, 1]$, the sensory input x received by Equation 1 gives the value from $[0, 1]$. As a direct indication of the system's ability to choose the best action a^* and to gauge a^* 's continuity with respect to x , at the end of each run x was scanned from 0 to 1 with increment 0.001 and the resulting a^* plotted in Figure 4⁴. The plot for GCS, Figure 4(a), lies close to the diagonal but has discontinuities. The plot for XCSFCA, Figure 4(b), is nearly coincident with the diagonal ' $1 - x$ ', but is *slightly broken* at some inputs x . The plot for the proposed approach XSRCFA, Figure 4(c), is exactly the diagonal with *no broken points*.

5.3 Analysis of Evolved Rules

The proposed approach evolved maximally general and accurate classifiers by exploiting the generalization ability of LCS (to create a maximally general condition) and the rich GP-based representation (to create the corresponding accurate action in the classifier). The experienced (i.e. $exp \geq 1/\beta$) and accurate (i.e. $\epsilon \leq \epsilon_0$) classifier rules from the final population of a run are shown in Table 1. The code fragmented action is shown in postfix form, where $D0$ represents the sensory input x . Here F , n , p , ϵ , and exp denote fitness, numerosity, prediction, prediction error, and experience of a classifier respectively, and as denotes the average size of the action sets this classifier has belonged to.

The proposed XSRCFA has successfully solved the frog problem evolving maximally general and accurate classifier rules like $9 - 16$ and $19 - 23$, shown in Table 1, that have the optimal payoff value of 1.0. Each of these maximally general classifier covers the whole input space ranging from 0.0 to 1.0 and has the accurate action that is equivalent to the diagonal ' $1 - x$ ', therefore each of them is able to solve the frog problem individually.

Subsumption deletion was made possible by comparing the code fragmented action on a character by character base, but due to the multiple genotypes to a

⁴ The results of GCS and XCSFCA were taken from [20] and [14] respectively, with permission from authors and copyright holders.

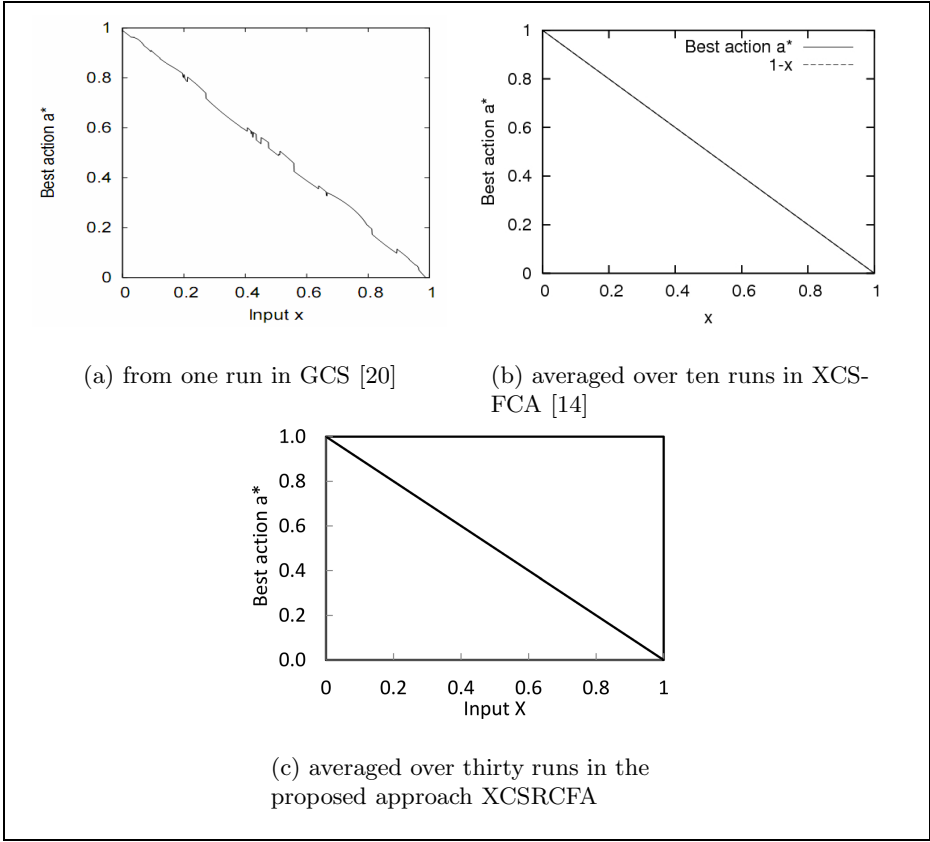


Fig. 4. Best action a^* of the frog problem is plotted by scanning the values of x from 0 to 1 increased by 0.001 using GCS, XCSFCA, and the proposed approach XCSRCFA. (a) The plot for GCS [20] has discontinuities. (b) The plot for XCSFCA [14] is nearly coincident with the diagonal ' $1 - x$ ', but is *slightly broken* at some inputs x . (c) The plot for the proposed approach is coincident with the diagonal and have *no broken points*.

single phenotype issue code fragmented XCSR could not combine the phenotypically similar classifiers, like 9 – 16 and 19 – 23, into a single macro-classifier while evolving the population. So, the final population contains redundant classifier rules.

The average size of the action sets as as classifiers 1 – 8 and 17 – 18 have belonged to is very small as compared with that of classifiers 9 – 16 and 19 – 23 due to the corresponding range of input space covered by them. As as is smaller for classifiers 1 – 8 and 17 – 18 so higher proportionate fitness F , but low numerosity n and experience exp highlight lack of worth.

Table 1. The experienced (i.e. $exp \geq 1/\beta$) and accurate (i.e. $\epsilon \leq \epsilon_0$) classifier rules, obtained in a typical run for the frog problem, using XCSR with code fragmented actions. The code fragmented action is shown in postfix form, where $D0$ represents the sensory input x . Here F , n , p , ϵ , and exp denote fitness, numerosity, prediction, prediction error, and experience of a classifier respectively, and as denotes the average size of the action sets this classifier has belonged to.

Sr. No.	Condition	Action	F	n	p	ϵ	exp	as
1	0.465356, 0.595225	D0 D0 *	0.907365	10	0.921854	0.006570	52	63.282196
2	0.000000, 0.064200	D0 2.0 /	0.893442	4	0.022754	0.005511	10	3.148032
3	0.133880, 0.279742	D0 D0 /	0.614114	10	0.834261	0.005719	72	41.985043
4	0.610654, 0.820274	D0 D0 *	0.612922	12	0.987231	0.009383	43	35.619358
5	0.673753, 0.881177	D0 2.0 / 0.0 -2.0 * +	0.498389	6	0.977880	0.008186	288	78.365341
6	0.891880, 0.943779	D0 D0 0.0 - -	0.497211	3	0.907607	0.003826	128	51.026615
7	0.891880, 0.943779	0.0	0.390534	3	0.907607	0.003826	128	51.026615
8	0.374393, 0.380699	D0 0.0 + 2.0 D0 * /	0.245285	1	0.877544	0.007733	5	22.799999
9	0.000000, 1.000000	D0 D0 / D0 -	0.153247	59	1.0	0.0	62585	384.999939
10	0.000000, 1.000000	0.0 -1.0 - D0 -	0.135065	52	1.0	0.0	58435	384.999939
11	0.000000, 1.000000	1.0 1.0 D0 * -	0.122078	47	1.0	0.0	50335	384.999939
12	0.000000, 1.000000	-2.0 -2.0 / D0 -	0.119481	46	1.0	0.0	67660	384.999939
13	0.000000, 1.000000	1.0 D0 0.0 + -	0.116883	45	1.0	0.0	49710	384.999939
14	0.000000, 1.000000	1.0 D0 -	0.111689	43	1.0	0.0	71558	384.999939
15	0.000000, 1.000000	-2.0 -2.0 / 1.0 D0 * -	0.100853	39	1.0	0.0	62710	384.999939
16	0.000000, 1.000000	-2.0 -2.0 / D0 0.0 + -	0.096548	37	1.0	0.0	53760	384.999939
17	0.680369, 0.895523	0.0 -2.0 * 0.0 D0 / +	0.052886	1	0.884182	0.006870	20	43.390942
18	0.641876, 0.825135	D0 2.0 / 0.0 D0 / +	0.037919	1	0.985085	0.006275	5	64.599998
19	0.000000, 1.000000	2.0 1.0 - D0 1.0 / -	0.033766	13	1.0	0.0	28460	384.999939
20	0.000000, 1.000000	-2.0 -2.0 / D0 1.0 / -	0.002597	1	1.0	0.0	736	384.999939
21	0.000000, 1.000000	1.0 0.0 - D0 0.0 + -	0.002597	1	1.0	0.0	5161	384.999939
22	0.000000, 1.000000	1.0 0.0 - D0 -	0.002597	1	1.0	0.0	12761	384.999939
23	0.000000, 1.000000	-2.0 -2.0 / 0.0 D0 + -	0.002597	1	1.0	0.0	34235	384.999939

6 Conclusions

This paper presented an extension to XCSR in which the discrete action has been replaced by a code fragmented action that is continuous with respect to the input state. The proposed XSRCFA has successfully solved the frog problem, consistently producing the optimum rules with the maximum predicted payoff.

The investigation of code fragments shows that the multiple genotypes to a single phenotype issue in XSRCFA disables the subsumption deletion function. Due to this loss of subsumption deletion, the final population contains optimal, but redundant classifier rules.

The next stage is to introduce a mechanism for treating two phenotypically similar code fragments as a single code fragment during the learning process, by simplifying the code fragment trees using algebraic and numerical simplification methods [9], in an attempt to reduce the number of classifiers in the final population. The XSRCFA will be tested for more continuous state space problems from the reinforcement learning domain.

Acknowledgments. We would like to thank the authors and copyright holders of [20] and [14] for granting permission to re-use the figures.

References

1. Ahluwalia, M., Bull, L.: A Genetic Programming Based Classifier System. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 11–18 (1999)
2. Alfaro-Cid, E., Merelo, J.J., de Vega, F.F., Esparcia-Alcázar, A.I., Sharman, K.: Bloat Control Operators and Diversity in Genetic Programming: A Comparative Study. *Evolutionary Computation* 18(2), 305–332 (2010)
3. Butz, M.V.: XCSJava 1.0: An Implementation of the XCS Classifier System in Java. Technical Report 2000027, Illinois Genetic Algorithms Laboratory (2000)
4. Butz, M.V.: Rule-based Evolutionary Online Learning Systems: A Principal Approach to LCS Analysis and Design. Springer (2006)
5. Butz, M.V., Wilson, S.W.: An Algorithmic Description of XCS. *Soft Computing - A Fusion of Foundations, Methodologies and Applications* 6(3-4), 144–153 (2002)
6. Iqbal, M., Browne, W.N., Zhang, M.: Evolving Optimum Populations with XCS Classifier Systems. *Soft Computing* (2012), <http://dx.doi.org/10.1007/s00500-012-0922-5>, doi:10.1007/s00500-012-0922-5
7. Iqbal, M., Browne, W.N., Zhang, M.: Extracting and Using Building Blocks of Knowledge in Learning Classifier Systems. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 863–870 (2012)
8. Iqbal, M., Zhang, M., Browne, W.N.: Automatically Defined Functions for Learning Classifier Systems. In: Proceedings of the Genetic and Evolutionary Computation Conference (Companion), pp. 375–382 (2011)
9. Kinzett, D., Johnston, M., Zhang, M.: Numerical Simplification for Bloat Control and Analysis of Building Blocks in Genetic Programming. *Evolutionary Intelligence* 2(4), 151–168 (2009)
10. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press (1992)
11. Lanzi, P.L.: XCS with Stack-Based Genetic Programming. In: Proceedings of the Congress on Evolutionary Computation, pp. 1186–1191 (2003)
12. Luke, S., Panait, L.: A Comparison of Bloat Control Methods for Genetic Programming. *Evolutionary Computation* 14(3), 309–344 (2006)
13. Preen, R.J., Bull, L.: Fuzzy Dynamical Genetic Programming in XCSF. In: Proceedings of the Genetic and Evolutionary Computation Conference (Companion), pp. 167–168 (2011)
14. Tran, T.H., Sanza, C., Duthen, Y., Nguyen, D.T.: XCSF with Computed Continuous Action. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1861–1869 (2007)
15. Urbanowicz, R.J., Moore, J.H.: Learning Classifier Systems: A Complete Introduction, Review, and Roadmap. *Journal of Artificial Evolution and Applications* 2009(1), 1–25 (2009)
16. Wilson, S.W.: Classifier Fitness Based on Accuracy. *Evolutionary Computation* 3(2), 149–175 (1995)
17. Wilson, S.W.: Get Real! XCS with Continuous-Valued Inputs. In: Lanzi, P.L., Stolzmann, W., Wilson, S.W. (eds.) *IWLCS 1999*. LNCS (LNAI), vol. 1813, pp. 209–219. Springer, Heidelberg (2000)
18. Wilson, S.W.: Mining Oblique Data with XCS. In: Lanzi, P.L., Stolzmann, W., Wilson, S.W. (eds.) *IWLCS 2000*. LNCS (LNAI), vol. 1996, pp. 158–174. Springer, Heidelberg (2001)
19. Wilson, S.W.: Classifiers that Approximate Functions. *Natural Computing* 1, 211–233 (2002)
20. Wilson, S.W.: Three Architectures for Continuous Action. In: Kovacs, T., Llorà, X., Takadama, K., Lanzi, P.L., Stolzmann, W., Wilson, S.W. (eds.) *IWLCS 2003*. LNCS (LNAI), vol. 4399, pp. 239–257. Springer, Heidelberg (2007)