

XCSLib: The XCS Classifier System Library

Pier Luca Lanzi and Daniele Loiacono
IlliGAL Report No. 2009005
March, 2009

Illinois Genetic Algorithms Laboratory
University of Illinois at Urbana-Champaign
117 Transportation Building
104 S. Mathews Avenue Urbana, IL 61801
Office: (217) 333-2346
Fax: (217) 244-5705

XCSLib: The XCS Classifier System Library

Pier Luca Lanzi*[†] and Daniele Loiacono[†]

*Illinois Genetic Algorithms Laboratory (IlliGAL)

University of Illinois at Urbana-Champaign, Urbana, Illinois, USA

[†]Politecnico di Milano — Dipartimento di Elettronica e Informazione

piazza Leonardo da Vinci 32, 20133 Milano, Italy

lanzi@elet.polimi.it, loiacono@elet.polimi.it

March 2, 2009

Abstract

The XCS Library (XCSLib) is an open source C++ library for genetics-based machine learning and learning classifier systems. It provides (i) several reusable components that can be employed to design new learning paradigms inspired to the learning classifier system principles; and (ii) the implementation of two well-known and widely used models of learning classifier systems.

1 Introduction

Learning classifier systems [8] are methods of genetics-based machine learning that combine temporal difference or supervised learning with a genetic algorithm to solve classification and reinforcement learning problems. They are characterized by a rule-based representation of solutions, a genetics-based exploration of the solution space, and an evaluation of solutions based on methods of supervised or reinforcement learning. In learning classifier systems, the current solution is represented by a set of condition-action-prediction rules, or a *population* of *classifiers*, in which each rule (each classifier) represents a portion of the overall solution. The classifier's condition identifies an area of the problem domain; the classifier's action represents a decision on the subproblem identified by its condition; the classifier's prediction estimates the value of the action in terms of problem solution.

Learning classifier systems learn to solve a problem by interacting with it through an on-line adaptation process that is very much similar to the trial-and-error interaction typical of reinforcement learning [22]. At discrete time steps, a classifier system perceives the current state of the problem through its *detectors* and based on its current knowledge (on the classifiers in the population) it selects an action to perform on the problem through its *effectors*. Depending on the effect that the performed action has on the problem, the system receives a numerical feedback which can be either interpreted as the fuel of a motivational reservoir [8] or as the incoming reward signal [7, 24, 27].

In this report, we briefly overview the XCS Library (XCSLib), an open source C++ library which provides several reusable components that can be employed to design new learning paradigms inspired by learning classifier system principles. XCSLib also provides the implementation of two learning classifier system models, namely XCS [24] and XCSF [27], for supervised and reinforcement learning problems.

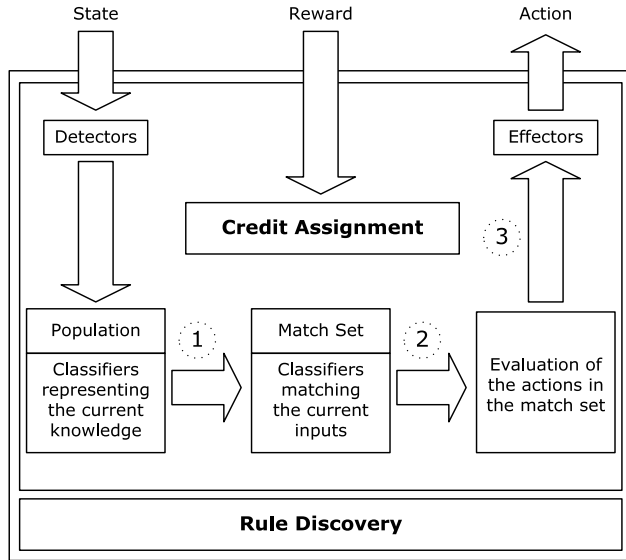


Figure 1: Simplified architecture of a learning classifier system:

2 Learning Classifier Systems, XCS, and XCSF

Everything in a learning classifier system revolves around the population of classifiers which identifies what the system knows about the problem solution. Each classifier represents a nugget of such a knowledge. In XCS and XCSF, a classifier consists of (i) a condition, which identifies an area of the problem domain; (ii) an action, which represents a decision on the subproblem identified by its condition; (iii) the prediction, which estimates the return that the system expects when the classifier is used; (iv) the error, which estimates the average error of the prediction; and (v) the fitness, which estimates the relative accuracy of the prediction and it is a function of the error.

The representation of the evolved solution depends both on the syntax used to express classifier conditions and on the approximator used to compute the classifier prediction. Classifier conditions can be represented using (i) simple Boolean predicates [24], when facing problems defined over binary domains, (ii) interval-based predicates [26, 20, 6], hyper-ellipsoid [5], and convex regions [18], when facing problems defined over real-valued domains; or even (iii) general-purpose symbolic LISP-like expressions or programs, which can be basically used in any domain [17]. Classifier prediction can be encoded as a real-valued parameter p , as in XCS, so that the population is an ensemble of piece-wise constant approximators estimating a target payoff landscape. Alternatively, as in XCSF, the classifier prediction is *computed* as a function $p(s, w)$ of the current input state s and a parameter vector w that is associated to each classifier. Thus, in XCSF, the population is an ensemble of approximators chosen from a wide spectrum of possibility such as, simple linear approximators [27], polynomial approximators [12], neural networks [10], support vector machines [19], tile-coding [13], etc. Alternatively, population may contain classifiers using several types of approximators at once [16].

The simplified architecture of a learning classifier system is shown in Figure 1. At each time step, the classifier system perceives the current state of the problem through its detectors; it builds a *match set* containing all the classifiers in the population whose condition matches the current sensory input. The match set typically contains classifiers which advocate contrasting actions; accordingly, the classifier system evaluates each action in the match set and selects an action to be performed balancing exploration and exploitation. The classifiers that advocate the selected action are put in the *action set* and the selected action is sent to the effectors to be executed.

Depending on the effect that the action has in terms of problem solution, the system receives a scalar reward. The incoming reward is distributed to the classifiers that are accountable for it through the *credit assignment* component. In the early classifier system models, this step was implemented by Holland's bucket brigade algorithm [9], a kind of temporal difference learning [21]. In XCS and XCSF [24, 27], credit assignment is implemented by a modification of Q-learning [22]. On regular basis, the rule discovery mechanism is activated with the goal of improving the current solution, that is finding better classifiers, by recombining and mutating the classifiers in the population. For this purpose, a genetic algorithm is applied to the classifiers in the action set. Two classifiers are selected, recombined, and mutated. The resulting offspring classifiers are inserted in the population while other two are deleted to keep the number of classifiers in the population constant.

3 Installation

To install `xcslib` first unpack the source code by typing:

```
tar zxvf xcslib-1.0.tar.gz
```

the command will create the directory `xcslib-1.0` containing the main sources and three subdirectories:

`./utility/` contains the scripts and the utilities to produce data for plotting;

`./docs/` contains the code documentation produced by `doxygen`;

`./examples/` contains example problems for XCS taken from [24] and [25] and for XCSF [15].

By default all the executables are copied in the directory `$(HOME)/bin`. This directory can be created by typing:

```
mkdir $(HOME)/bin
```

the directory should also be added to the `PATH` environment variable. If you are using `bash` just type:

```
export PATH=$PATH:$HOME/bin
```

To list all the possible options type `make` that will produce,

```
make
```

```
print this help
```

```
make all
```

```
build all the available executables
```

```
make mp
```

```
build the version of XCS to learn Boolean multiplexer
```

```
make fa
```

```
build the version of XCSF to approximate functions
```

```
make gw
    build the version of XCSF to solve the 2D gridworld
```

```
make utility
    build and install the utilities
```

To compile and install all the available versions of XCS and XCSF including the utilities, type “`make all`”, which will install everything in the directory `$HOME/bin/`. To specify another installation directory (e.g. `mydir`), type “`make all INSTALL=mydir`”. Alternatively, it is possible to compile each single version of XCS. For instance, the command “`make mp`” will compile the version of XCS that can be used to learn Boolean multiplexer (`xcs-mp`).

4 Running the Examples

After the executables have been generated and installed, it is possible to run the examples that are included in the distribution.

4.1 Boolean Multiplexer

As a very first example, we will apply XCS to learn the Boolean multiplexer of 6 bits (or 6-multiplexer) with the same parameter settings used in in [24]. First, move to the directory `xcslib-1.0/examples/6-multiplexer` which contains the configuration file for the experiment, i.e., “`confsys.mp6`”. The experiment consists of ten runs each one consisting of 10000 learning problems and 10000 testing problems (as in [24], learning problems and testing problems alternate). To check whether the executable of XCS for the Boolean multiplexer has been installed correctly type “`which xcs-mp`”. The command should return the directory where the file “`xcs-mp`” has been installed; if the default settings have used, the command should return `$(HOME)/bin/xcs-mp`, where “`$(HOME)`” is the home directory of the current user. At this point, the experiment can be started by typing:

```
xcs-mp -f mp6
```

which will print the parameters setting read from the configuration file and it will trace the execution of each run, as follows,

```
1/10 saving the experiment state... ok
1/10 saving the population state... ok
```

```
...
```

```
10/10 saving the experiment state... ok
10/10 saving the population state... ok
```

To avoid such printing, the experiment can be run in background by typing, `xcs-mp -f mp6 >& /dev/null &`. While the experiment is running, it is possible to monitor the statistics that the system keeps about the current run by typing “`tail -f statistics.mp6-?`”.

The command will print seven scrolling columns like:

```
...
3 13745 1 1000 48 43.6233 Testing
3 13746 1 1000 48 1.94525 Learning
3 13747 1 1000 48 2.77569 Testing
3 13748 1 1000 48 24.9166 Learning
3 13749 1 1000 48 73.961 Testing
...
```

The first column reports the current run; the second column reports the current problem, the third column reports the number of steps required to solve the problem (note that, since the 6-multiplexer is a one-step problem, the third column is always one in this experiment); the fourth column reports the sum of the rewards the system obtained during the problem; the fifth column reports the number of macroclassifiers in the population; the sixth column reports the absolute difference between the reward predicted by the system and the actual reward received from the environment; finally, the seventh column indicates whether XCS has solved the problem in *learning* mode or in *testing* mode. The columns will continue to scroll until the run ends or the `tail` command is interrupted by a `control-C`. When the execution is completed, the directory will contain three types of files. Experiment files (e.g., `experiment.mp6-0.gz`) contain the state of the each experiment and they can be used to restart an experiment from the point it was stopped. Statistics files (e.g., `statistics.mp6-5.gz`) contain the most relevant data collected during each experiment such as, the number of steps, the reward obtained, the population size, and the prediction error. Population files (e.g., `population.mp6-3.gz`) contain the final population evolved during for each experiment. For instance, we can inspect the final population produced in the fourth run by typing:

```
zcat population_report.mp6-0 | more
```

which will produce an output like:

```
...
44385 01#0## : 0 1.000e+03 2.274e-13 9.996e-01 2.910e+01 1623 28
44373 10##1# : 0 1.171e-57 1.524e-55 1.000e-00 2.694e+01 585 27
44379 01#1## : 0 6.830e-57 2.989e-54 9.999e-01 2.896e+01 564 27
44376 10##0# : 1 2.540e-54 2.922e-52 9.997e-01 2.827e+01 558 26
44406 001### : 1 1.000e+03 2.274e-13 9.342e-01 2.876e+01 1305 26
44379 01#1## : 1 1.000e+03 2.274e-13 9.657e-01 2.744e+01 1313 26
44337 11###1 : 0 6.309e-45 6.046e-43 9.997e-01 2.508e+01 321 24
44391 11###0 : 0 1.000e+03 1.337e-160 9.991e-01 2.330e+01 1657 24
44370 000### : 0 1.000e+03 2.274e-13 9.995e-01 2.437e+01 1721 24
44340 11###1 : 1 1.000e+03 1.137e-13 1.000e-00 2.459e+01 1441 23
...
```

Where, column 1, is a unique identifier of the classifier; column 2, reports the classifier condition; column 3, reports the classifier action; column 4, reports the prediction; column 5, reports the prediction error; column 6, reports the fitness; column 7, reports the action set size; column 8, reports the experience (the number of updates); and column 9, reports the numerosity.

To analyze the results produced we will use the utility program we installed at the beginning. First to produce the plot for the performance just type:

```
prepare_rwd.sh -f mp6 -w 100 -i 100
```

this command asks for the results regarding the reward obtained by the system as a moving average over the last 100 problems (command “-w 100”) and to report only such data every 100 problems (command “-i 100”). The command will produce (i) ten files, from `perf.mp6-0` to `perf.mp6-9`, reporting the XCS performance in each run as the moving average of the reward received in the last 50 problems; (ii) the file `AVERAGE.perf.mp6-10` containing the average of the ten performance files. Likewise, we can produce the statistics regarding the number of classifiers in the population, by typing:

```
prepare_pop.sh -f mp6 -w 100 -i 100
```

The command will produce (i) ten files, from `popul.mp6-0` to `popul.mp6-9`, reporting the XCS performance in each run as the moving average of the reward received in the last 50 problems; (ii) the file `AVERAGE.popul.mp6-10` containing the average of the ten performance files. Finally in the directory there is the file `report.mp6-10` containing the following summary of the CPU time elapsed for the overall computation, for each experiment, and for each problem:

```
TOTAL ELAPSED TIME          9.4
DETAILS FOR EXPERIMENTS
```

EXP.	ELAPSED IN EXP.	AVG FOR PROB.
0	0.95sec	9e-05sec
1	0.95sec	8.6e-05sec
2	0.97sec	8.7e-05sec
3	0.91sec	8e-05sec
4	0.92sec	8.1e-05sec
5	0.93sec	8.9e-05sec
6	0.95sec	9.3e-05sec
7	0.87sec	8e-05sec
8	0.97sec	9.2e-05sec
9	0.99sec	9.3e-05sec

4.2 Restarting an experiment

Suppose now, we wish to continue the previous experiment for other 10000 problems. To do this we have just to edit the `confsys.mp6` file and to replace the line:

```
first problem = 0
```

with the following line:

```
first problem = 10000
```

which specifies that additional 10000 learning problems (and 10000 testing problems) should be solved by starting from problem 10000. Now we can restart the experiment by typing:

```
xcs-mp -f mp6 >& /dev/null &
```

Note that as the new series of experiments begins the files about the experiments statistics will be uncompressed and the information about the new problems will be added.

```

.....
.QQF..QQF..OQF..QQG..OQG..OQF.
.OOO..QOO..OQO..OOQ..QQO..QQQ.
.OOQ..OQQ..OQQ..QQO..OOO..QQO.
.....
.QQF..QOG..QOF..OOF..OOG..QOG.
.QQO..QOO..OOO..OQO..QQO..QOO.
.QQQ..OOO..OQO..QQQ..QQQ..OQO.
.....
.QOG..QOF..OOG..OQF..OOG..OOF.
.OOQ..OQQ..QQO..OQQ..QQO..OQQ.
.QQO..OOO..OQO..OOQ..OQQ..QQQ.
.....

```

Figure 2: The Woods2 Environment.

4.3 Woods Environments

In the second example, we apply XCS to control an artificial animal (or animat [23]) that must learn how to survive in an artificial environment. `Woods2` (see Figure 2) is a grid with two types of obstacles (represented by “O” and “Q” symbols), food positions (represented by “F” and “G” symbols), and empty positions (represented by “.” symbols). `Woods2` is a torus: its left and right edges are connected as well as its top and bottom edges. The animat, controlled by XCS, can occupy any of the empty positions and it can move to any adjacent position that is empty. The animat has eight sensors, one for each adjacent position, that are encoded by three bits coding features of the object. Thus, the animats sensory input is a string of 24 bits (3 bits \times 8 positions). There are eight possible actions, one for each possible adjacent position. The animat must learn how to reach food positions starting from any empty position. When it reaches a food position (F or G) the problem ends, and the animat receives a constant reward equal to 1000; otherwise it receives zero.

To apply XCS to `Woods2`, we first have to move into the directory `xcslib-1.0/examples/woods2` which contains the configuration file for the experiment (`confsys.woods2`), the map of the environment (`woods2.map`), and a `gnuplot` file to generate the plots for performance and population size. The experiment consists of twenty runs of 5000 learning problems and 5000 testing problems each (as in [24, 25], learning problems and testing problems alternate). To check whether the executable of XCS for the woods environment has been installed correctly type “`which xcs-woods2`”. The command should return the directory where the file “`xcs-woods2`” has been installed; if the default settings have used, the command should return `$(HOME)/bin/xcs-woods`, where “`$(HOME)`” is the home directory of the current user. At this point, the experiment can be started by typing:

```
xcs-woods2 -f woods2
```

which will again print the parameters setting read from the configuration file and it will trace the execution of each run as follows,

```

1/20 saving the experiment state... ok
1/20 saving the population state... ok
...

```

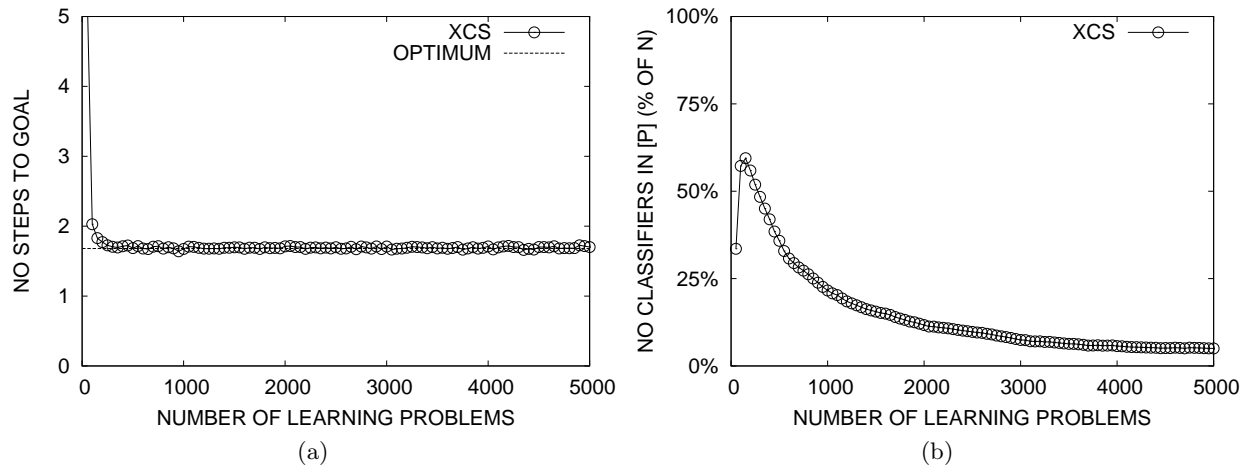


Figure 3: XCS applied to Woods2: (a) performance; (b) number of classifiers in the population.

```
20/20 saving the experiment state... ok
20/20 saving the population state... ok
```

The execution will produce several files containing information about the state of each experiment (whose name starts with `experiment.woods2`), the final populations (whose name starts with `population.woods2`), and the statistics collected for each problem (whose name starts with `statistics.woods2`).

Woods environments are sequential (or multistep) problems and to analyze the performance of XCS we need to generate the data of the average number of steps that the animat needs to reach goal position. This can be achieved by using the scripts provided in the distribution and typing,

```
prepare_steps.sh -f woods2 -w 50 -i 50
```

which generates the results regarding the number of steps required to reach goal by the system as a moving average over the last 50 test problems (command “`-w 50`”) and to produce only one data point every 50 problems (command “`-i 50`”). The command will produce (i) twenty files, from `steps.woods-0` to `steps.woods2-19`, reporting the XCS performance in each run as the moving average of the reward received in the last 50 problems; (ii) the file `AVERAGE.perf.woods2-20` containing the average of the twenty performance files. Similarly, we can produce the statistics regarding the number of classifiers in the population, by typing:

```
prepare_pop.sh -f woods2 -w 100 -i 100
```

The two average files can be used to produce the typical performance and population plots [24, 25] as the ones showed in Figure 3 by executing `gnuplot` on the configuration file `plot_woods2.gpi` that is provided in this distribution.

4.4 XCSF for Function Approximation

In the third example, we will apply XCS with computed prediction (briefly XCSF) to approximate a simple real-valued function, i.e., a sinusoidal function with period 1 and defined between 0 and 1. The experiment is similar to the one discussed in [27] and it uses using piece-wise linear

approximators updated using Normalized Least Mean. The configuration file for this experiment, `confsys.sin`, can be found in the directory `xcslib-1.0/examples/functions` where we have move to begin the experiment. To run XCSF on this simple function approximation problem just type “`xcsf-fa -f sin`”; as before, the parameter settings read from the configuration file will be printed and then the execution of ten runs will be traced as follows,

```
1/10 saving the experiment state... ok
1/10 saving the population state... ok
...
10/10 saving the experiment state... ok
10/10 saving the population state... ok
```

As in the previous examples, the execution will generate the usual files of the final populations, the files of the experiment states which can be used to restart the experiments, and the files of the statistics. When applied to function approximation problems, the performance of XCSF is measured as the approximation error obtained [27] and the learning process is analyzed by plotting how the approximation error changes as the learning proceeds. While generalization is again evaluated by tracing how the number of classifiers changes during evolution. To generate the data regarding the evolution of the approximation error type,

```
prepare_se.sh -f sin -w 100 -i 100
```

This command generates the data file of the approximation error as a moving average over a window of 100 test problems (command “`-w 100`”) and it reports only one data point every 100 test problems (command “`-i 100`”). As usual, the data of the population size can be obtained as in the previous examples by typing,

```
prepare_pop.sh -f sin -w 100 -i 100
```

Figure 4 reports the plots of the approximation error and population size for this example that have been generated using the `gnuplot` configuration file provided in the distribution. To generate the plots just type “`gnuplot plot_sin.gpi.`” As can be noted in Figure 4a the approximation error rapidly goes below the error threshold *epsilon zero* (or ε_0) that has been specified in the configuration file (`confsys.sin`). The same directory also provides other configuration files to perform experiments on other well-known functions [12, 14, 15].

4.5 XCSF for the Two-Dimensional Gridworld

In the fourth example, we will apply XCSF to the 2D gridworld (firstly introduced in [3]) in which the current state is defined by a pair of real coordinates $\langle x, y \rangle$ in $[0, 1]^2$, the goal is in position $\langle 1, 1 \rangle$, and there are four possible actions (left, right, up, and down); each action corresponds to a step of size s in the corresponding direction; in this experiment we set s as 0.05; 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, otherwise it receives -0.5.

To perform this experiment, first we have to move to `xcslib-1.0/examples/2d-gridworlds` where the configuration file (`confsys.2dgrid`) is found. Then, to start the experiment, we must type,

```
xcsf-gw -f 2dgrid
```

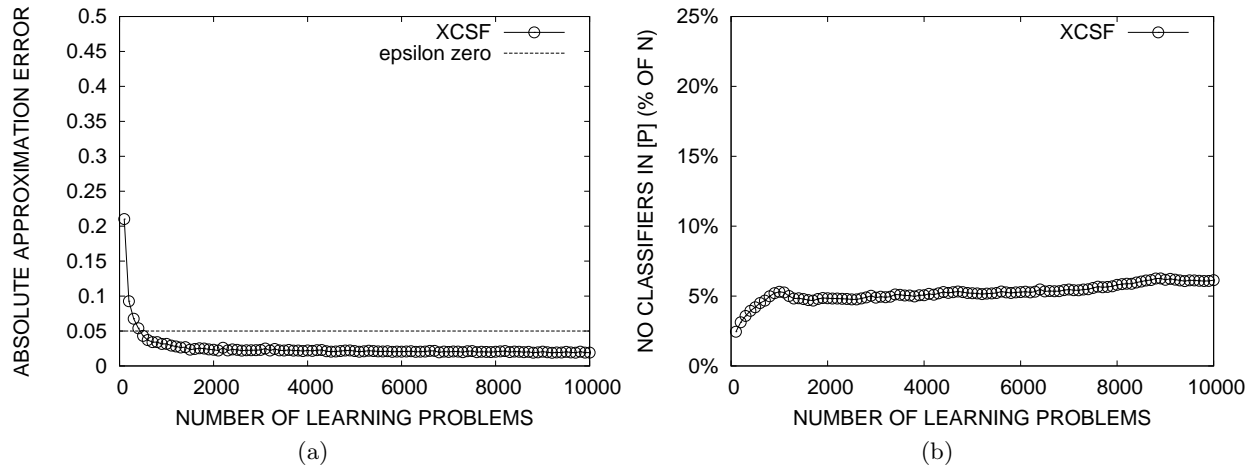


Figure 4: XCSF applied to approximate a sinusoidal function with period 1 and defined between 0 and 1: (a) approximation error; (b) number of classifiers in the population.

the parameter settings read from the configuration file will be printed and then the execution of ten runs will be traced as follows,

```
1/10 saving the experiment state... ok
1/10 saving the population state... ok
...
10/10 saving the experiment state... ok
10/10 saving the population state... ok
```

When the execution stops, we can generate the data files for the performance (computed as the average number of steps to the goal position) and the number of classifiers in the population by typing,

```
prepare_steps.sh -f 2dgrid -w 100 -i 100
```

and,

```
prepare_pop.sh -f 2dgrid -w 100 -i 100
```

The two data files generated by these commands (`AVERAGE.steps.2dgrid-10` and `AVERAGE.popul.2dgrid-10`) can be used to generate the plots for the performance and population size reported in Figure 5 using the gnuplot script (`plot_2dgrid.gpi`) that has been included in the directory.

4.6 Other Examples

The distribution include other examples involving several problems that has been used in the literature including woods1 [24], the hidden parity [11], the count ones [4], and the finite state world environment [1, 2]. All the executables to run these examples can be compiled using the file `makefile.others`. To check all the other versions of XCS available in the distribution just type `make -f makefile.others`. For instance, the version of XCS that solves the finite state world problem can be generated by typing `make -f makefile.others fsm`. The configuration files are available in the subdirectories of the the directory `examples`.

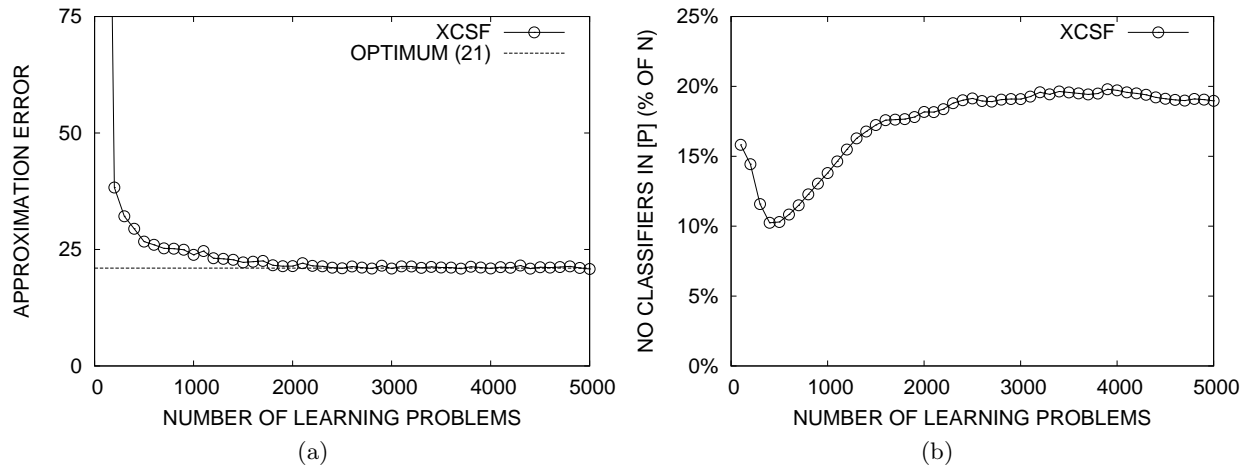


Figure 5: XCSF applied to the two-dimensional gridworld [3]: (a) performance; (b) number of classifiers in the population.

5 Using XCSLib as a Library

XCSLib not only implements XCS [24, 25] and XCSF [27] but it also can be used to build programs based on these models using the classes provided in the distribution. For instance, we can use the classes provided in the distribution to write a program that takes as input a population evolved by XCS (using binary inputs, binary actions and ternary conditions), a sensory input (possibly also an action), and it prints all the classifiers in the population that match the current input and advocate the specified action (if provided). The source code of the program is reported as Listing 1.

To read a classifier population, the program must first include all the basic classes (binary inputs, binary actions, ternary conditions and classifiers) and the class `xcs_config_mgr2` that is used to read configuration files (lines 1-3 in Listing 1)). Then, the parameters specified by the users are read from the command line (lines 11-25), the configuration file specified by the user is read (line 34) and the classes for binary actions and ternary conditions are initialized using the configuration parameters read from the file (lines 36-40). After the classes are initialized, the population file is opened (line 49) and the classifiers can be read from the population file one by one (line 51); classifiers are matched against the input provided by the user (line 53) and (if specified) also against an action (lines 55 and 57); and only the matching ones are printed (line 59). To compile, move into the directory containing the source files and type,

```
make -f make/xcs.make MATCH VERSION=tb
```

which will produce the executable `match-tb` (i.e., match for ternary conditions and binary actions). The compiled program can be applied to print all the classifiers in a population that match a certain input. For instance, the command,

```
match mp6 population.mp6-0 100001
```

reads the configuration file `confsys.mp6` and it prints all the classifiers in the population file `population.mp6-0` that match the input 100001.

Listing 1: Match Utility

```

1 #include "xcs_definitions.hpp"
2 #include "xcs_classifier.hpp"
3 #include "xcs_config_mgr2.hpp"
4
5 int
6 main(int argc, char *argv[])
7 {
8     unsigned long act;
9     bool    select_action = false;
10
11     if ( (argc!=4) && (argc!=5) )
12     {
13         cout << endl << endl;
14         cout << argv[0] << "<prefix>_<population>_<input>_<action>_" << endl;
15         cout << "\t<suffix>_specify_the_configuration_file" << endl;
16         cout << "\t<population>_file_containing_classifier_population" << endl;
17         cout << "\t<input>_sensory_input_" << endl;
18         cout << "\t<action>_classifier_action_to_be_selected" << endl;
19         cout << endl << endl;
20         exit(-1);
21     }
22
23     string    suffix = string(argv[1]);
24     string    population = string(argv[2]);
25     string    inputs = string(argv[3]);
26
27     if (argc==5)
28     {
29         act = atoi(argv[4]);
30         select_action = true;
31     }
32
33     xcs_config_mgr2 xcs_config(suffix);    /// init the configuration manager
34
35     t_action    dummy_action(xcs_config); /// init the action class
36
37     t_condition    dummy_condition(xcs_config); /// init the condition class
38
39     string    string_condition;
40     t_condition    condition;
41     xcs_classifier    classifier;
42
43     unsigned long no_actions = dummy_action.size();
44
45     vector<xcs_classifier> set;    /// population is a vector of classifiers
46     ifstream    INPUT(population.c_str());
47
48     while (INPUT>>classifier)
49     {
50         if (classifier.match(t_state(inputs)))
51         {
52             if (select_action)
53             {
54                 if (classifier.action==t_action(act))
55                 {
56                     cout << classifier << endl;
57                 }
58             }
59             else
60                 cout << classifier << endl;
61         }
62     }
63 }

```

References

- [1] Alwyn M. Barry. *XCS Performance and Population Structure within Multiple-Step Environments*. PhD thesis, 2000.
- [2] Alwyn M. Barry. The stability of long action chains in xcs. *Journal of Soft Computing*, 6(3–4):183–199, 2002.
- [3] Justin A. Boyan and Andrew 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.
- [4] Martin Butz, Kumara Sastry, and David E. Goldberg. Tournament selection: Stable fitness pressure in XCS. In Erick Cantú-Paz, James A. Foster, Kalyanmoy Deb, Lawrence Davis, Rajkumar Roy, Una-May O’Reilly, Hans-Georg Beyer, Russell K. Standish, Graham Kendall, Stewart W. Wilson, Mark Harman, Joachim Wegener, Dipankar Dasgupta, Mitchell A. Potter, Alan C. Schultz, Kathryn A. Dowsland, Natasa Jonoska, and Julian F. Miller, editors, *GECCO*, volume 2724 of *Lecture Notes in Computer Science*, pages 1857–1869. Springer, 2003.
- [5] Martin V. Butz, Pier Luca Lanzi, and Stewart W. Wilson. Function approximation with xcs: Hyperellipsoidal conditions, recursive least squares, and compaction. *IEEE Transactions on Evolutionary Computation*, 12(3):355–376, 2008.
- [6] Hai H. Dam, Hussein A. Abbass, and Chris Lokan. Be real! xcs with continuous-valued inputs. In Franz Rothlauf, editor, *GECCO Workshops*, pages 85–87. ACM, 2005.
- [7] J.H. Holland. Escaping brittleness: The possibilities of general purpose learning algorithms applied to parallel rule-based systems. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning, An Artificial Intelligence Approach*, volume 2, chapter 20, pages 593–623. Morgan Kaufmann, Los Altos, CA, 1986.
- [8] John H. Holland. *Adaptation in Natural and Artificial Systems*. Universtiy of Michigan Press, Ann Arbor, MI, 1975. second edition 1992.
- [9] John H. Holland. Escaping Brittleness: The possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based Systems. In Mitchell, Michalski, and Carbonell, editors, *Machine learning, an artificial intelligence approach. Volume II*, chapter 20, pages 593–623. Morgan Kaufmann, 1986.
- [10] David Howard, Larry Bull, and Pier Luca Lanzi. Self-adaptive constructivism in neural xcs and xcsf. In *GECCO ’08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*. ACM Press, 12-16 July 2008.
- [11] Tim Kovacs and Manfred Kerber. What makes a problem hard for XCS? In Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *Advances in Learning Classifier Systems*, volume 1996 of *LNAI*, pages 80–99. Springer-Verlag, Berlin, 2001.
- [12] Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. Extending XCSF beyond linear approximation. In *Genetic and Evolutionary Computation – GECCO-2005*, pages 1859–1866, Washington DC, USA, 2005. ACM Press.

- [13] Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. Classifier prediction based on tile coding. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1497–1504, New York, NY, USA, 2006. ACM Press.
- [14] Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. Prediction update algorithms for xcsf: Rls, kalman filter, and gain adaptation. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1505–1512, New York, NY, USA, 2006. ACM Press.
- [15] Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. Generalization in the xcsf classifier system: Analysis, improvement, and extension. *Evolutionary Computation Journal*, 15:133–168, 2007.
- [16] Pier Luca Lanzi, Daniele Loiacono, and Matteo Zanini. Evolving classifier ensembles with heterogeneous predictors. In *A Compilation of two exciting workshop years - IW LCS 2006 / 2007. Current advances and future outlooks*. Springer-Verlag, 2008. in print.
- [17] Pier Luca Lanzi and Alessandro Perrucci. Extending the Representation of Classifier Conditions Part II: From Messy Coding to S-Expressions. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 99)*, pages 345–352, Orlando (FL), July 1999. Morgan Kaufmann.
- [18] Pier Luca Lanzi and Stewart W. Wilson. Using convex hulls to represent classifier conditions. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1481–1488, New York, NY, USA, 2006. ACM Press.
- [19] Daniele Loiacono, Andrea Marelli, and Pier Luca Lanzi. Support vector machines for computing action mappings in learning classifier systems. In *Proceedings of the 2007 Congress on Evolutionary Computation (CEC2007)*, Singapore, September 2007. IEEE.
- [20] Christopher Stone and Larry Bull. For real! XCS with continuous-valued inputs. *Evolutionary Computation*, 11(3):298–336, 2003.
- [21] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.
- [22] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning – An Introduction*. MIT Press, 1998.
- [23] Stewart W. Wilson. Classifier Systems and the Animat Problem. *Machine Learning*, 2(3):199–228, 1987.
- [24] Stewart W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.
- [25] Stewart W. Wilson. Generalization in the XCS classifier system. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deband Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldbergand Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 665–674. Morgan Kaufmann, 1998. <http://prediction-dynamics.com/>.

- [26] Stewart W. Wilson. Mining oblique data with xcs. In Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *IWLCS*, volume 1996 of *Lecture Notes in Computer Science*, pages 158–176. Springer, 2001.
- [27] Stewart W. Wilson. Classifiers that approximate functions. *Natural Computing*, 1(2-3):211–234, 2002.