Extending the Representation of Classifier Conditions Part II: From Messy Coding to S-Expressions

Pier Luca Lanzi and Alessandro Perrucci
Politecnico di Milano Artificial Intelligence and Robotics Project
Dipartimento di Elettronica e Informazione
Politecnico di Milano
`lanzi@elet.polimi.it`

Abstract

In this paper we present the results of the second part of our research which is aimed at the study of alternative representations of classifier conditions. In particular we introduce an extension of the XCS classifier system in which bitstring representation is replaced by s-expressions. We show that XCS with LISP s-expressions, XCSL, can reach optimal performance in different types of applications with different complexity. The results we present also suggest that great care must be taken in choosing the representation language; in particular we show that in certain cases the use of "or" clauses may lead to an unstable performance. Overall, our initial results show that this is a promising approach for a future development of a general purpose representation of classifier conditions and that there are still many issues which need to be investigated.

1 INTRODUCTION

The learning capabilities of an adaptive agent rely, in one way or another, on its ability to generalize over the many situations it experiences. An agent can generalize properly only if it can represent in some way the regularities of the environment it is learning in. In learning classifier systems generalization is achieved by the evolutions of *general* condition-action rules (i.e. the classifiers) which represents the task the agent is learning. Accordingly, the generalization capability of a learning classifier system implicitly depends on the representation it employs for classifier conditions. These are usually represented by bitstrings of fixed-length on the ternary alphabet {0,1,#} that test a condition over a set of binary sensory inputs. When using a learning classifier system, multivariate sensory inputs must be encoded into binary strings. Unfortunately, when sensory inputs are coded as bitstrings part of the structure of the environment may be lost. As a consequence the agent may be unable to generalize properly since the regularities of the environment have been lost through binary encoding.[1] To exploit the regularities of the environment different representations of classifier conditions are necessary. Wilson [11] suggested that an interesting extension to the classifier conditions syntax would consist of using conjunctions of interval predicates over *continuous* sensory inputs, or as more general s-expressions [10].

In this paper we present the results of the second part of our research aimed at the study of alternative representation of classifier conditions. In the first part [7] we showed how we can use a variable-length messy coding to represent classifier conditions instead of the usual fixed-length bitstrings. In this second part, we follow Wilson's idea and introduce an extension of Wilson's XCS classifier system, we call it XCSL, in which the usual bitstring conditions are replaced by general s-expressions. We use XCS because it has an accurate generalization mechanism, and a learning mechanism which is based on a well-known reinforcement learning technique, i.e., Q-learning [8]. In fact, XCS has been shown to evolve *optimal* solutions which are represented with near-minimal populations of classifiers that are accurate and maximally general [5, 11].

The remainder of this paper is organized as follows. In Section 2 we show how XCS is extended by introducing s-expressions to represent classifier conditions. In Section 3 we discuss the design of experiments we employ in the rest of this paper. The new system, XCSL, is applied to the problem of learning boolean functions

---

[1] The limitations of bitstring encoding in learning classifier systems have been discussed in the literature (see for example [4]). Recently, Wilson [12] has analyzed such limitations w.r.t. to the generalization issue.

in Section 4 while in Section 5 we apply XCSL to two types of grid environments. The first (Section 5.1) is the `Woods1` we already used to test messy classifier systems; the second (Section 5.2) is an extension of `Woods1` in which the agent can also perceive light. The paper ends in Section 6 where we draw some conclusions and some directions for future works.

2 EXTENDING XCS WITH LISP S-EXPRESSIONS

Similarly to what we have previously done with messy classifier systems [7] we now extend XCS by introducing s-expressions to represent classifier conditions. We do not describe XCS here but we refer the interested reader to the first part of this paper ([7]) for a brief overview or to [10] for a detailed description.

The new system, XCSL, works basically like XCS while it differs from it (i) in covering, (ii) in how conditions are matched, and (iii) in the genetic operators it employs. In the following we discuss each of these differences in detail.

Representation. When considering s-expressions to represent classifier conditions we may think of using very general s-expressions capable of representing any functions over sensor values. However, since the problems we are considering here are quite simple we restrict classifier conditions to the set of possible boolean functions that can be generated by composition of logical operators (*and*, *or*, and *not*) with elementary conditions over sensory inputs. These atomic conditions depends on the problem: for boolean functions (Section 4) they are simple boolean variables; for woods environments (Section 5) they are predicates that test sensors values.

Matching. The matching with s-expressions is straightforward. Given a certain configuration of sensory inputs, the condition is evaluated as a LISP s-expression except for the *or* clause which is evaluated as follows. Suppose XCSL has to match the condition "`(or S1 S2)`", initially the system randomly selects the order in which the two s-expressions are evaluated (`S1-S2` or `S2-S1`), then it evaluates the *or* clause as usual. Elementary conditions which test single sensors values are evaluated as specified by the problem definition. As in messy classifier systems [7] a classifier condition can be *underspecified* (i.e. not all the sensors appear in the condition), or *overspecified* (i.e. some sensors are tested with different conditions). However, when using s-expressions the policy for dealing with underspecification and overspecification is implicitly defined by the evaluation mechanism of LISP s-expressions and has not to be defined at design time as it happens in messy classifier systems.

Covering. Covering creates a classifier with a random condition that matches the current sensors and a random action. The work with messy classifier system [7] has shown that to limit overgeneralization due to underspecification it is important to introduce most of the sensors in newly created classifiers. Accordingly, in XCSL covering creates a random condition which is a conjunction of three expressions each one matching the current sensory inputs. One of these is built as an *and* of elementary clauses of each current sensor input; for instance, when learning boolean functions this is the *minterm* which matches the current sensory configuration. The remaining two expressions are still *and* expressions which match a random number of current sensory inputs. This covering policy guarantees that the current sensor configuration is exactly matched by the first expression while introducing generalization by underspecifications in the remaining two expressions and by the two *or* clauses.[2]

Genetic Algorithm. As in XCS and in XCSm [7], in XCSL the genetic algorithm selects two classifiers from the action set with probability proportional to their fitnesses, copies them, with probability χ performs crossover, and with probability μ mutates them. In XCSL crossover and mutation works as in traditional Genetic Programming [6]. Crossover selects two cutting point in the trees that represent classifier conditions, then it exchange the two subtrees that are individuated by the cutting points. Mutation first selects a random mutation point in the classifier condition; then it deletes the subtree rooted by the mutation point; finally it inserts a randomly generated subtree at that point. Mutation on classifier action works like in XCS.

3 EXPERIMENTAL DESIGN

Each experiment consists of a number of problems that the system must solve. The system can solve a problem *exploring* possibly new solutions; otherwise, the system can solve a problem *exploiting* the knowledge it has acquired. In the former case we say that the system solves the problem *in exploration*, in the latter we say that the system solves the problem in *exploitation*. At the beginning of a new problem the agent decides with

---

[2] The number of *three* conjunctive expressions to be used in covering was found through a series of experiments in the different environments.

```
<cond> := "(" NOT <cond> ")" |
          "(" AND <cond> <cond> ")" |
          "(" OR  <cond> <cond> ")" |
          <var>

<var>  := "X0" | "X1" | "X2"
```

Figure 1: The BNF grammar that generates all the possible classifier conditions for the boolean functions with three variables. Non-terminal symbols are in square brackets. Terminal symbols are in quotation marks.

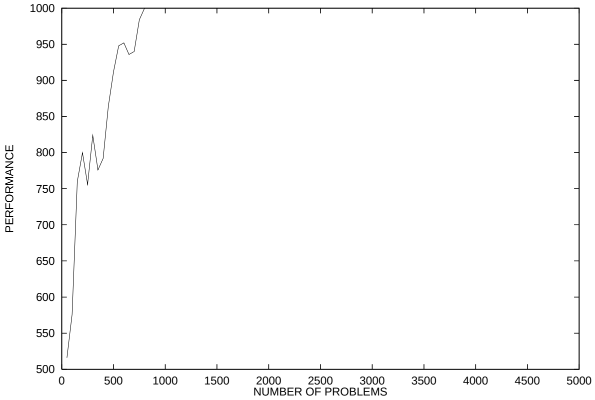

Figure 2: Performance of XCSL for learning the boolean function $EQ(.)$. Curve is an average over ten experiments.

probability 0.5 whether it will solve the problem in exploration or in exploitation. When in exploration XCSL selects actions randomly among the ones in the current match set (i.e. the action selection procedure is probabilistic). When in exploitation XCSL always selects the action that predicts the highest payoff (i.e. the action selection procedure is deterministic).

4 BOOLEAN FUNCTIONS

To study XCSL learning capabilities we start by applying XCSL to solve single steps problems in which the reward is not delayed. In particular we apply XCSL to the problem of learning boolean functions[3] which has been already tackled in Genetic Programming [6], in genetic algorithms [1, 3], as well as in learning classifier systems [5, 9, 11]. During each experiment input strings representing assignments of the function variables are randomly presented to XCSL. For example, if the function has two variables the input string "$((x_0$ $1)$ $(x_1$ $0))$" assigns 1 to variable x_0 and 0 to variable x_1. XCSL returns an action which is the predicted value of the boolean function. If XCSL' prediction is correct the system receives a constant reward equal to 1000, otherwise it receives 0. In exploration problems actions are selected randomly, the genetic algorithm operates, and classifier parameters are updated. In exploitation problems the action with the highest payoff is always selected, the genetic algorithm is turned off, and classifier parameters are not updated.

The classifier conditions that we use in these experiments are s-expressions representing boolean expressions generated by composing the logical *and*, *or*, and *not* functions, and the set of the possible input variables (x_0, x_1,...). Formally, admissible classifier conditions (with three input variables) are described by the BNF ([2]) depicted in Figure 1.

4.1 SIMPLE FUNCTIONS

Initially we apply XCSL to two simple boolean functions in which only three variables are used. The first function we use is $EQ(x_0, x_1, x_2)$ which returns 1 if $x_0 = x_2$, 0 otherwise. We use XCSL with 50 classifiers to learn this function. XCSL parameters are set as follows: [4] β=0.2, θ= 25, ε_0=.01, μ=0.01, ϕ=0.5. The performance of XCSL, calculated as the reward received in the last 50 exploitation problems, is reported in Figure 2. The curve is an average over ten experiments. As the results show XCSL can easily learn this simple function optimally, in fact, after the first 1000 problems XCSL prediction is always correct.

The second boolean function we use to test XCSL is the 3-multiplexer. Boolean multiplexer functions are defined for binary strings of length $l = k + 2^k$ bits: the first k bits represent an address that indexes into the remaining 2^k bits. The boolean multiplexer function returns as the result the indexed bit. For instance in the 3-multiplexer ($l = 3$ and $k = 1$) the value of the input string $((x_0\ 1)(x_1\ 0)(x_2\ 1))$ is 1: x_0 in fact address the second bit (x_2) which is 1. We apply XCSL to the 3-multiplexer with a population of 50 classifiers; XCSL parameters are set as in the previous experiment. XCSL performance depicted in Figure 3 shows that the system can easily reach the optimum also in this small problem.

---

[3] We remind the reader that a boolean function f with n variables, $x_0 \ldots x_n$, is defined as a function $f : \{0,1\}^n \rightarrow \{0,1\}$.

[4] Note that XCSL has the same parameter as XCS. Some of these parameters have not been presented but are reported here for sake of completeness. We refer the interested reader to Wilson's original paper [10] for a complete discussion of XCS parameters.

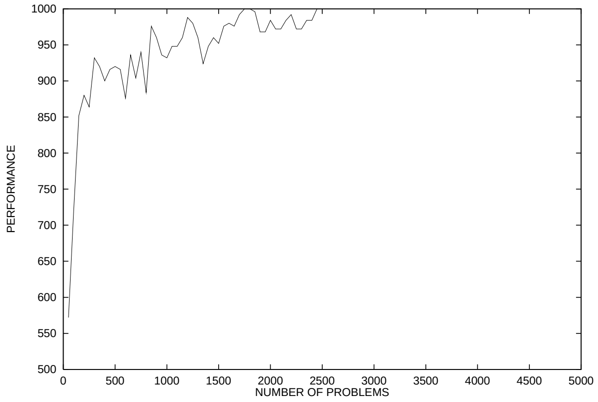

Figure 3: Performance of XCSL for learning the 3-multiplexer. Curve is an average over ten experiments.

4.2 THE 6-MULTIPLEXER FUNCTION

We now extend the results for boolean functions by applying XCSL to the 6-multiplexer problem a common testbed for learning classifier systems [5, 9, 10]. In the 6-multiplexer there are two address bits (k=2), thus the function has six variables. We use XCSL to learn the 6-multiplexer function with a population of 800 classifiers; parameters are set as in the previous experiments. Figure 4 reports XCSL best performance over ten experiments (solid line) and a typical XCSL performance for this problem. The best XCSL performance for the 6-multiplexer (solid line in Figure 4) show that the system can reach the optimum for small periods. However, the typical performance is quite unstable as the plot shows (dashed line). In particular, we note that XCSL performance widely oscillates near to the optimum but it never reaches it in a stable way. This behavior is similar to what we observed with messy classifier systems [7]. In that case the oscillating behavior was due to the presence of overgeneral genes that remained "hidden" in classifier conditions because they were not checked during the matching phase. This phenomenon in XCSL may be produced by the *or* function. In fact, when matching an *or* clause XCSL selects randomly in which order the two conditions shall be tested. Therefore, it may happen that some part of a classifier condition is not matched for a certain period; like in messy classifier systems [7] that portion of condition is "hidden" until it is evaluated because it is selected for matching. It is worth noting that this phenomenon is likely to increase as the complexity of the condition grows because of possibly increasing number of nested *or* functions.

To test whether our intuition is correct, we apply two versions of XCSL to the the 6-multiplexer problem. The former does not insert any *or* in classifier conditions during covering but let mutation introduce *or* clauses. The latter does not employ *or* functions in classifier conditions. Figure 5 compares three typical performances of the three versions of XCSL we discussed so far. As can be noticed XCSL performance improves as we move from the first version of XCSL, in which *or* are used both in covering and mutation (lower dashed line in Figure 5), to the basic XCSL which does not employ *or* (solid line). This results supports our intuitions and also what we noticed with messy classifier conditions.

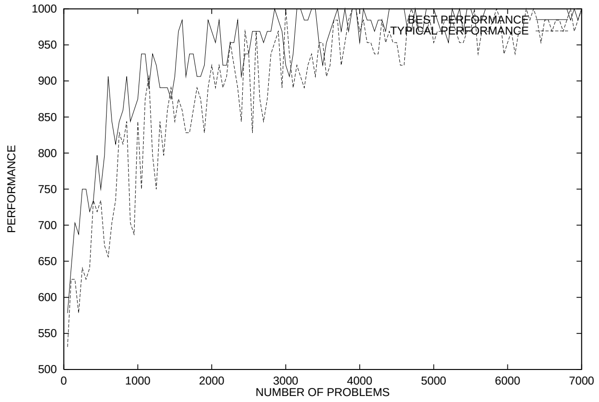

Figure 4: Best performance of XCSL over ten experiments for learning the 6-multiplexer boolean function (solid line) and a typical performance of an XCSL run for the same problem (dashed line).

4.3 *OR* CLAUSES IN CLASSIFIER CONDITIONS

Before we proceed any further we wish to discuss some general issues concerning the use of logical *or* in classifier conditions. A classifier represents a rule like:

```
if C then action A predicts P
```

which states that if condition `C` is satisfied by current sensory inputs, the action `A` promises a payoff `P`. A classifier is accurate and maximally general if its condition, `C`, matches many situations and if its payoff prediction, `P`, is correct in each situation the classifier matches. In learning classifier systems the disjunction of two conditions, `C1` and `C2`, can be represented by two distinct classifiers:

```
(a) if C1 then action A predicts P
(b) if C2 then action A predicts P
```

When we introduce the *or* operator in classifiers condition we try to represent these two classifiers, (a) and

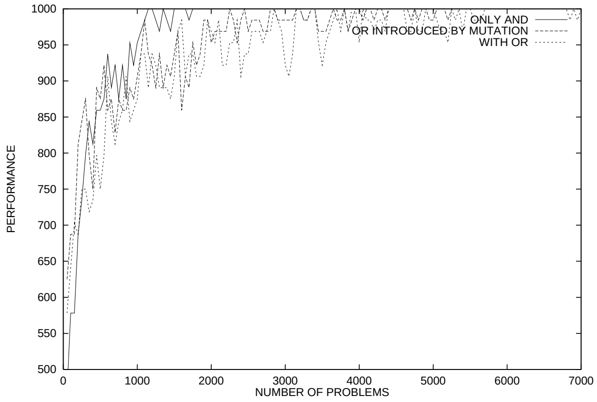

Figure 5: Three performances of XCSL for the 6-multiplexer problem: XCSL performance when only *and* and *not* operators are used (solid line); XCSL performance when *or* are not inserted during covering but only by mutation (dashed line); best XCSL performance when *or* is used both in covering and mutation.

(b), as one classifier:

```
(c) if C1 or C2 then action A predicts P
```

This classifier applies in the same situations as (a) and (b), predicting in each situation the same payoff predicted by (a) and (b). Thus we can say that classifier (c) is equivalent to the twin classifiers (a) and (b). The *or* clause increases the system capabilities of representing general conditions because it makes possible the merging of two classifiers, (a) and (b), into a single classifier (c) that is more general than the previous ones. However, there is a subtle difference between the form of generalization introduced by the *or* function in classifier (c) and the pair of classifiers (a) and (b). Let us illustrate this difference with an example.

Suppose that XCSL is learning a certain task and that for a relatively long period of time its sensory inputs match condition `C1` but not condition `C2`;[5] suppose also that XCSL uses classifier (c) for deciding how to act. Note that, as long as only condition `C1` is matched XCSL performance does not change even if condition `C2` is corrupted by the genetic algorithm (we can suppose that `C2` becomes overgeneral). Classifier (c) can be corrupted (for example by crossover or mutation over condition `C2`) but XCSL cannot be aware of what is happening because not all the condition is used during matching. We can say that condition `C2` is "*hidden*" by the *or* clause.

This does not happen when we use the twin classifiers (a) and (b) in place of (c) because when matching condition `C1` (but not `C2`) the genetic algorithm only applies to classifier (a). We can summarize this as follows:

> When we use the **or** clause in classifier condition we can change part of the condition without having any evidence whether this change improves or corrupt the classifiers until we can check **all** the elements of the condition.

5 MULTISTEP PROBLEMS

The problem of learning boolean function we used in the previous experiments are single step problems since each system action can be rewarded. We now extend previous results applying XCSL to more challenging *sequential* problems in which reward is received only after a sequence of actions is performed.

5.1 WOODS ENVIRONMENTS

In this second set of experiments we apply XCSL in *woods* environments. These are grids in which each cell can contain an obstacle (a "`T`" symbol), a goal (an "`F`" symbol), otherwise it can be empty. An agent placed in the environment must learn to reach goal positions. The agent has eight sensors, one for each adjacent cell, and can move in any of the adjacent cells. If the destination cell contains an obstacle the move does not take place; if the destination cell is blank then the move takes place; finally, if the cell contains a goal the agent moves receiving a constant reward, and the problem ends. For each problem the agent is randomly placed in an empty cell of the environment. Then the agent moves under the control of XCSL until it reaches a goal position receiving a constant reward, and the problem ends. System performance is computed as the running average of the number of steps to a goal position in the last 50 testing problems. Every statistic is averaged over ten experiments. We employ the usual exploration/exploitation strategy (see Section 3). When solving a problem in exploration the agent selects actions randomly, the genetic algorithm operates, and the classifier parameters are updated. When solving a problem in exploitation, the agent always selects the action which predict the highest payoff, the genetic algorithm is *not* operating, and classifier parameters are updated.

Classifier conditions for this environment are s-expressions representing the set all the possible boolean functions generated composing elementary

---

[5] This example works also if both conditions match the sensory inputs but only `C1` is used for matching.

```
<cond> := "(" NOT <cond> ")" |
          "(" AND <cond> <cond> ")" |
          "(" OR  <cond> <cond> ")" |
          <sensor>

<sensor> := "(" <tag> "," <value> ")"
<tag> := "N" | "W" | "S" | "E" |
         "NW" | "SW" | "SE" | "NE"
<value> := "." | "F" | "T"
```

Figure 6: The BNF grammar that generates all the possible classifier conditions for woods environments. Non-terminal symbols are in square brackets. Terminal symbols are in quotation marks.

T	**T**	**F**		
T	**T**	**T**		
T	**T**	**T**		

Figure 7: The `Woods1` environment.

conditions that test sensory inputs by the usual logical operators (*and*, *or*, and *not*). The notation for atomic conditions is similar to the one we used for messy genes in [7]. An elementary condition consists of a *tag* which represents one sensor of the possible eight agent's sensors (N, W, S, E, NW, SW, SE, and NE) and a value that represents the sensor reading ("T", "F", or "."). For instance the condition "`(N,.)`" tests whether in the position at north is empty. Admissible classifier conditions for woods environments are represented by the BNF in Figure 6.

We apply XCSL in `Woods1` environment (Figure 7) with a population of 800 classifiers. XCSL parameters are set as follows: β=0.2, γ=0.71, θ= 25, ε_0=.01, χ=0.8, μ=0.01, ϕ=0.5. The experimental results depicted in Figure 8 show that XCSL learns an optimal policy for `Woods1`.

This result apparently contradicts what we observed in the experiments with boolean functions. In fact, in `Woods1` the use of the *or* clause does not introduce any instability in XCSL performance. However, we observe that in `Woods1` sensory inputs change frequently because it is very easy for the agent to change position. Accordingly, the phenomenon we discussed previously is limited. Moreover, in the boolean function representation it is very easy to produce overgeneral condition because all the search space is considered. On the contrary, in `Woods1` only a limited number of input configurations are presented to the agent.

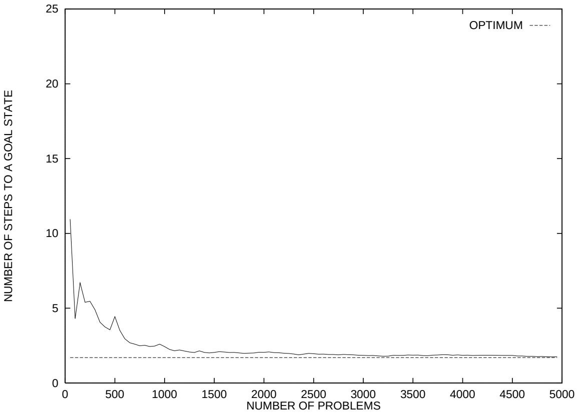

Figure 8: Performance of XCSL in `Woods1`. Population size is 800 classifiers. The curve is an average over ten runs. Horizontal line represents optimal performance.

5.2 ENVIRONMENTS WITH LIGHT

Up to now we have indeed used s-expressions to represent classifier conditions but the problem we tackled and the sensors models we used are more or less the same employed in previous experiments with XCS. Now, we extend the representation of classifier conditions by adding different types of sensors to the ones we used in the previous experiment. For this purpose we use a variation of the woods environments in which the goal (represented by the "`F`" symbol) glows. The agent has *sixteen* sensors, two for each adjacent cell: one is the usual "*proximity*" sensor which indicates the content of the cell; one is a light sensor which tells the agent how intense is the the light in the corresponding cell. Light intensity is represented by a positive integers. Light is intense near the goal position and decreases by one for each cell that separates the agent from the goal. In our example, the goal is perceived as a light intensity equal to 3; a light intensity of 1 means that the agent is at two positions from the goal; obstacles ("`T`" symbols) are perceived as a light intensity of 0.

Elementary conditions consists of the proximity conditions we employed in the previous experiment and of light conditions which are used to test the presence of light. For instance the condition "`(GT S 1)`" tests whether the sensor at south senses a light with an intensity greater than 1; the condition "`(GT N NE)`" compares the light perceived by the sensor at north with the light perceived by the sensor at north-east.

```
<cond> := "(" NOT <cond> ")" |
          "(" AND <cond> <cond> ")" |
          "(" OR  <cond> <cond> ")" |
          <proximity> |
          <light>

<proximity>
        := "(" <tag> "," <type> ")"

<light> := "(" <cmp> <tag> <value> ")" |
           "(" <cmp> <tag> <tag> ")"

<tag>   := "N"  | "E"  | "S"  | "W" |
           "NE" | "SE" | "SW" | "NW"

<type>  := "T" | "F" | "."

<cmp>   := "LT" | "EQ" | "GT"

<value> := "0" | "1" | "2" | "3"
```

Figure 9: The BNF grammar that generates all the possible classifier conditions for woods environment with light. Non-terminal symbols are in square brackets. Terminal symbols are in quotation marks.

The set of the admissible classifier conditions are described by the BNF in Figure 9.

We apply XCSL to this extended version of the `Woods1` environment with a population of 800 classifiers and the same parameter settings employed in the previous experiments. Figure 10 shows XCSL performance in `Woods1` when light sensors are used (solid line). As can be noticed the system converges rapidly to an optimal performance. If we analyze each single experiment, we note that sometimes the system rapidly converges to the optimum (lower dashed line in Figure 10). Other times XCSL converges very slowly to the optimal performance (upper dashed line).

6 CONCLUSIONS

We have presented an extension of the XCS classifier system, called XCSL, in which s-expressions are used to represent classifier conditions in place of bitstrings. In XCSL, classifier conditions have been defined as any boolean expression that can be generated by composing logical operators (*and*, *or*, and *not*) with elementary predicates over sensors values. XCSL was used to learn booleans functions of increasing complexity. The results of these experiments showed that in general XCSL can reach optimal performance, but also that the use of *or* clauses can cause instability in XCSL performance. We analyze this phenomenon and developed an explanation of it. Briefly, we suggest that when using *or* clauses the system may not be aware that evolution has corrupted part of the condition (for example making it overgeneral) since that part is not tested during the matching procedure because of an *or* clause. Finally, we applied XCSL to two sequential problems involving proximity sensors and light sensors. The results we present show that in both cases XCSL can evolve an optimal policy for the two problems.

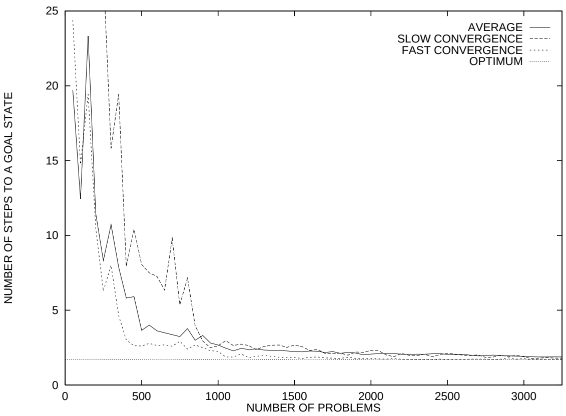

Figure 10: Performances of XCSL in `Woods1` when the goal glows: average XCSL performance over ten experiments (solid line); XCSL performance when convergence is fast (lower dashed line); XCSL performance when convergence is slow. Population size is 800 classifiers. Horizontal line represents optimal performance.

The work we present in this paper is just at the beginning. There are a number of topics that we have not discussed here which we are currently studying while writing. Maybe the most interesting concerns the concepts of "general classifier" that the use of s-expressions introduces. In bitstring representation is quite simple to state that a classifier is more general than another one (conditions can be compared bit by bit) and the cost of this operation is linear in the number of condition bits. Unfortunately, when using s-expressions testing whether a condition is more general than another one is far more complex (at least NP). As a consequence the subsumption operator (which has proved effective in increasing XCS generalization capability) cannot be employed anymore and different heuristics must be developed. In particular our initial results suggest that the use of *condensation* techniques [11, 5] are particularly effective in reducing the population size.

Another problem we have noticed in these initial experiments concerns the complexity of classifier conditions which tends to grow as learning proceeds. Also in this case exact methods based on the simplification of boolean conditions are infeasible because they would require too much computation time. Accordingly some heuristic methods should be developed in order to limit condition complexity and to increase XCSL tendency to evolve short conditions preferentially.

Acknowledgments

This work was partially supported by the Politecnico di Milano Research Grant "Development of Autonomous Agents Through Machine Learning," and by the project "CERTAMEN" co-funded by the Italian Ministry of University and Scientific Research.

Pier Luca wish to thank Andrea Bonarini and Pino Contini who helped him in making this submission possible after that fire devastated part of his department. Many thanks go also Stewart W. Wilson for his invaluable advices and to Tim Kovacs for his comments and corrections. Alessandro wish to thank his parents for their invaluable support.

References

[1] Lashon B. Booker. Triggered rule discovery in classifier systems. In *Proceedings Third International Conference on Genetic Algorithms*, pages 265–274. Morgan Kaufmann, 1989.

[2] Mehdi Jazayeri Carlo Ghezzi. *Programming Language Concepts 3rd ed.* John Wiley and Sons, 1997.

[3] D. E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[4] David E. Goldberg, Jeffrey Horn, and Kalyanmoy Deb. What makes a problem hard for a classifier system? Technical Report IlliGAL Report No 92007, University of Illinois, 1992.

[5] Tim Kovacs. XCS Classifier System Reliably Evolves Accurate, Complete, and Minimal Representations for Boolean Functions. In Chawdhry Roy and Pant, editors, Soft Computing in Enginerring Design and Manufacturing, pages 59–68. Springer-Verlag London, 1997.

[6] John Koza. *Genetic Programming.* MIT Press, 1992.

[7] Pier Luca Lanzi. Extending the representation of classifier conditions, part I: From binary to messy coding. In W Banzhaf et al., editor, *GECCO-99: Proc. of the Gen. and Evol. Comp. Conf.* Morgan Kaufmann, 1999.

[8] C.J.C.H. Watkins. Learning from delayed reward. PhD Thesis, Cambridge University, Cambridge, England, 1989.

[9] Stewart W. Wilson. Classifier systems and the animat problem. *Machine Learning*, 2(3):199–228, November 1987.

[10] Stewart W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

[11] Stewart W. Wilson. Generalization in the XCS classifier system. In J. Koza et al, editor, *Proceedings of the Third Annual Genetic Programming Conference*, pages 665–674, Madison (WI), 1998. Morgan Kaufmann San Francisco (CA).

[12] Stewart W. Wilson. State of XCS classifier system research. Technical Report 99.1.1, 1999. Available at `http://prediction-dynamics`.