Extending the Representation of Classifier Conditions Part I: From Binary to Messy Coding

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

Abstract

We present the first part of a study on the alternative representations of classifier conditions. We introduce an extension of the XCS classifier system in which variable-length messy chromosomes replace the original bitstring representation. With a series of experiments we show that to reach optimal performance covering, matching, and mutation must be adequately defined in order to avoid overgeneralization due to underspecification of classifier conditions.

1 INTRODUCTION

Classifier conditions are usually represented by bitstrings of fixed length on the ternary alphabet {0,1,#}. This simple representation, that was initially introduced by Holland [5] for sake of simplicity, has been used to tackle many different types of problems: autonomous robotics [2], classification [6], or personal agents [15].

Although there had been a significant amount of work in the field of *Fuzzy Classifier Systems* [1], the use of bitstring for coding classifier conditions has never been perceived as a limit of this learning paradigm. This is mainly because in the past years the research in learning classifier systems focused on the learning capabilities of these types of systems rather than on their generalization capabilities. Recently, Wilson [12] introduced a new model of learning classifier system, XCS, which has moved the focus of a branch of learning classifier system research from the performance issue to the generalization issue.

XCS represents a major advance in learning classifier systems research because it has an accurate generalization mechanism, and a learning mechanism which is based on a well-known reinforcement learning technique, namely Q-learning [11]. Wilson [13] presented experimental results showing that XCS can learn *optimal* behaviors which are represented by a small number of accurate and maximally general classifiers.

Since in XCS optimal performance is achieved by the use of a powerful reinforcement learning technique, the focus of the research for XCS has moved to the generalization issue [7, 9, 13]. From a more general viewpoint, XCS' capability of producing compact representations of the learned tasks makes it an interesting alternative to others and maybe most common reinforcement algorithms like tabular Q-learning. In this perspective the ability of XCS to generalize properly implicitly supports the use of learning classifier systems instead of other reinforcement learning techniques that come from the Machine Learning community that do not employ any sort of generalization.

As generalization becomes the main focus of the research, the representation of classifier conditions turns out to be probably the most interesting topic. The generalization capabilities of adaptive agents in fact rely on their ability to represent the knowledge they acquire in a compact form. Unfortunately, when the 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. Wilson [13] suggests 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 [12].

Following Wilson's ideas, we studied different ways to extend the representation of classifier conditions. To begin we replaced the usual bitstring coding of classifier conditions with a *messy coding* in which sensory inputs are still translated into bitstring but the bits in the classifier's conditions are not bound to the position of sensory input bits anymore. We analyzed the behavior of this *messy* version of XCS in grid environments to test how the change of condition representation influences the system performance. Although this extension to classifier conditions may appear simple, the analysis of the results with the *messy* version of XCS highlight many interesting phenomena that we also find when we pass to more complex representations like s-expressions. As a second step (not discussed here) we extended the condition representation to the more general syntax in which s-expressions are used to represent general conditions on sensory inputs (results are discussed in [10]).

In this paper we present the results of the first part of our research which involves the use of a *messy coding* to represent classifier conditions. We begin by introducing XCS in Section 2 and the design of experiments we use in this paper in Section 3. In Section 4 we discuss the reasons that took us to choose the messy representation; in Section 5 we briefly overview the principal ideas introduced with messy genetic algorithms [3]. In Section 6 we introduce the messy version of XCS, XCSm, and apply it to two environments showing that its performance is only near optimal. We discuss these results in Section 7 and present a series of modification to the original system that is able to reach optimal performance. In Section 8 we present an experiment which shows how messy classifiers make the reuse of previous learned behavior easy. Finally, in Section 9 we draw some conclusions and directions for future works. XCSm can evolve optimal solutions and we discuss the different problem that can be encountered when this type of representation is used in learning classifier systems. Finally, in Section 9 we draw some conclusions.

2 DESCRIPTION OF XCS

The major difference between Wilson's XCS classifier system [12] and Holland's classifier system [5] is the definition of fitness. In XCS the fitness of a classifiers is evaluated as the accuracy of the classifier prediction, while in Holland's learning classifier systems the fitness is evaluated as the classifier prediction itself. Accordingly, in XCS the original *strength* parameter is replaced by three parameters: (i) the prediction p, which evaluates the payoff that the agent is expected to gain; (ii) the prediction error ε, which evaluates the error of the prediction p; finally, (iii) the fitness F, which evaluates the accuracy of the prediction p.

XCS works as follows. At each time step the system input is used to build the *match set* [M] containing the classifiers in the population whose condition matches the sensory inputs. If the match set is empty a new classifier that matches the input sensors is created through *covering*. For each possible action a_i the *system prediction* $P(a_i)$ is computed. $P(a_i)$ gives an evaluation of the expected payoff if action a_i is performed. Action selection can be *deterministic* (the action with the highest system prediction is chosen), or *probabilistic* (the action is chosen with a certain probability among the possible actions). Those classifiers in [M] that propose the selected action are put in the current *action set* [A]. The selected action is performed and an immediate reward is returned to the system together with a new input configuration. The reward received from the environment is used to update the parameters of the classifiers in the action set corresponding to the previous time step $[A]_{-1}$. Classifier parameters (p, ε, and F) are updated using a Q-learning-like technique [11, 12].

Covering. Covering is used when the match set [M] is empty or the system is stuck in a loop. In both cases, the covering operator creates a classifier that matches the sensory inputs and has a random action. This classifier is then inserted in the population and, if necessary, another classifier is deleted. The situation in which the system is stuck in a loop is detectable because the predictions of the classifiers involved start to diminish steadily. To detect this phenomenon when [M] is created the system checks whether the total prediction of [M] is less than ϕ times the average prediction of the classifiers in the population.

Genetic Algorithm. The genetic algorithm in XCS is applied to the action set. It selects two classifiers with probability proportional to their fitnesses, copies them, and with probability χ performs crossover on the copies while with probability μ mutates each allele.

3 EXPERIMENTAL DESIGN

All the experiments presented in this paper have been conducted in the woods environments. These are grid worlds 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 perceives the environment by eight sensors, one for each adjacent cell: food is perceived as "11"; an obstacle is perceived as "10"; finally, an empty cell is perceived as "00". The agent can move into 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.

Each experiment consists of a number of problems that the agent must solve. For each problem the agent is randomly placed in a blank cell of the environment; then it moves under the control of the system until it reaches a goal position receiving a constant reward, and the problem ends. The agent can solve a problem by *exploring* the environment trying to learn a better solution; otherwise, the agent can solve a problem *exploiting* the knowledge it has acquired. The agent takes this decision at the beginning of a new problem when it decides with probability 0.5 whether it will solve the problem *in exploration* or *in exploitation*. When solving a problem in exploration the system selects the action to be performed randomly (i.e. the action selection procedure is probabilistic). When solving a problem in exploitation the system selects the action that predicts the highest payoff (i.e. the action selection procedure is deterministic). The performance of XCS is computed as a running average of the number of steps to a goal position in the last 50 problems solved in exploitation. Every statistic presented in this paper is averaged over ten experiments.

4 WHY MESSY CODING?

Bistring representation of classifier conditions has two major limitations. The most recognized one concerns the use of binary encoding for sensors which, in general, can result in a loss of information about the environment structure. The second limitation concerns the fixed correspondence between the position of bits in the classifier condition and the position of sensors bits. Wilson's proposal [14] of using conjunctions of interval predicates over continuous inputs to represent classifier conditions would eliminate the first kind of limitation. The proposal of using general s-expression to represent classifier conditions [12] would eliminate both limitations at once.

Comparing the two types of limitations that current representation has, we observe that the use of binary coding for sensors can become a limitation only in some types of environments. Conversely, the positional bound between sensory inputs and classifier conditions is independent from the environment and from the types of sensors.

We believe that to start developing a general purpose representation for classifier conditions we should initially face general problems in a simple way. Accordingly, in this paper we start by introducing a very basic extension of the bitstring representations in which conditions have variable-length and there is not a fixed correspondence between the positions of sensory inputs and genes in classifier conditions.

The simplest solution for eliminating the fixed correspondence between the positions of condition bits and the positions of sensory bits consists of (i) naming the sensors with a *tag* so that the classifier system can separate the inputs of the different sensors, (ii) using the sensors tags to separate the different subparts of the condition string that are devoted to a specific sensor. This type of solution was firstly introduced in genetic algorithms by Goldberg [3] who introduced the idea of *Messy Genetic Algorithms*. This type of *messy* representation was also used by Hoffmann [4] with Fuzzy Classifier Systems.

5 MESSY GENETIC ALGORITHMS

Messy genetic algorithms were introduced in [3] as an enhancement of standard genetic algorithms with fixed-length chromosomes. In the following, we briefly overview some of the characteristics of messy genetic algorithms that are particularly interesting for learning classifier systems. We do not discuss the general properties of messy genetic algorithms, and we refer the interested reader to [3] for a complete overview.

Representation. In messy genetic algorithms genes are represented as pairs: (*Gene Number*, *Allele Value*). A messy chromosomes is a sequence of messy genes; for example $((1,1)(5,0)(1,2)(0,3))$ is a chromosomes with four genes. A messy chromosomes can be *underspecified*, i.e., the chromosome does not have an allele value for all the possible genes (in the example genes number 2 and 4 are not represented). To evaluate the fitness of a chromosome that is underspecified a *partial evaluation* of the fitness function is introduced. A messy chromosome may also be *overspecified* in that it may specify more values for the same gene (in the example above two values are given for gene number 1). Overspecification is solved by using positional precedence: the value of a gene is specified by the first messy gene found in a left-to-right scan of the chromosome.

Genetic Operators. In messy genetic algorithms crossover is replaced by two operators named *cut* and *splice*. The cut operator given a bitwise cut probability p_k and an overall cut probability $p_c = p_k(\lambda - 1)$ (where λ is the number of messy genes in the chromosome), cuts the chromosome into two parts. The splice operator concatenates with probability p_s two messy chromosomes. Recombination of two chromosome is performed as follows. First, cut is applied to the two individuals selected for recombination. Then splice is applied with a certain probability on the possible couples that have been produced by the cut operator. Mutation is applied with a certain probability to the *value* part of messy genes.

6 A MESSY VERSION OF XCS

We now introduce the messy version of XCS, XCSm, in which classifiers conditions are represented by variable length messy chromosomes. Then we apply XCSm to two environments to test whether XCSm as defined here can learn an optimal policy in these environments.

6.1 DESCRIPTION OF XCSm

It is quite straightforward to extend XCS adding messy representation to classifier conditions. The messy version of XCS, XCSm, works basically as XCS, but it differs from it in three main parts: covering, matching, and in the genetic operators. In the following we introduce the messy representation of classifier condition and show how covering, matching, and the genetic operators works in XCSm.

Representation. In XCSm, the original fixed length binary representation is replaced by a *messy* representation of classifier conditions. A messy classifier condition is a sequence of messy genes which represent elementary conditions on a specific sensor. Messy genes consist of a *tag* that specifies which sensor the messy gene tests, and a fixed length bitstring that represents a condition on binary sensors. Specifically, in woods environments the tag specifies one of the eight possible agent sensors (N, E, S, W, NE, SE, SW, and NW). For instance the messy gene (N,1#) matches current sensory input if the position at the north of the agent (symbol N) contains a goal (sensed as "11") or an obstacle (sensed as "10");

Matching. To define how messy classifiers are matched against sensory inputs we need to decide a policy for dealing with the possible underspecification and overspecification of a classifier condition. Although in messy genetic algorithms underspecification is difficult to handle, in messy classifier systems dealing with underspecification is straightforward. Genes that are not specified in the conditions are treated as though their bitstrings would contain only don't cares. Note that this approach introduces two forms of generalization: A classifier can be general because its condition tests many sensors (i.e., it contains many different tags) each one against a very general condition. On the other hand a classifier can be general because it tests only few sensors. To deal with overspecification (i.e., more genes in the condition test the same sensor), we employed the same approach, based on positional precedence, that is used in messy genetic algorithms. When a condition is matched, messy genes are checked in left-to-right order: the first messy gene that tests a sensor is used for matching.

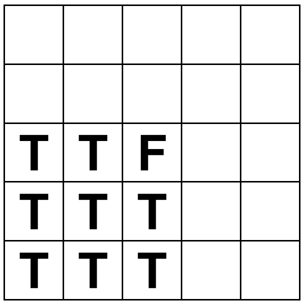

Figure 1: The `Woods1` environment.

Covering. Covering creates a classifier with a random messy condition that matches the current sensors and a random action. The messy condition is generated as a random sequence of messy genes. Covering in XCSm is a combination of two random factors: the number of sensors explicitly covered by messy genes (which in a certain sense represents the degree of underspecification of the classifier), and the creation of the bitstring that covers the sensors bits. For each sensor the system generates with probability P_s a messy genes with the tag corresponding to that sensor. Then the bitstring of the messy gene which matches that sensor is generated like in XCS. To avoid trivial classifiers, conditions generated through covering must have at least one gene.

Genetic Algorithm. In XCSm the genetic algorithm selects two classifiers from the action set with probability proportional to their fitnesses, copies them, and with probability χ recombines them through cut and splice. Probability χ is not fixed, like in XCS, but is computed as $p_k(\lambda - 1)$ where p_k is the probability of cutting a single messy gene and λ is the length of the shorter condition of selected classifiers. Splice combines the four segments of the two classifiers conditions according to one the four policies discussed in [3] which is selected with uniform probability. Mutation is applied with probability μ on the bitstrings of each messy gene in the messy condition and on the action part of the classifier.

T	T	T	T	T	T	T	T
T			T			F	T
T	T			T			T
T	T		T			T	T
T							T
T	T		T				T
T					T		T
T	T	T	T	T	T	T	T

Figure 2: The `Maze4` environment.

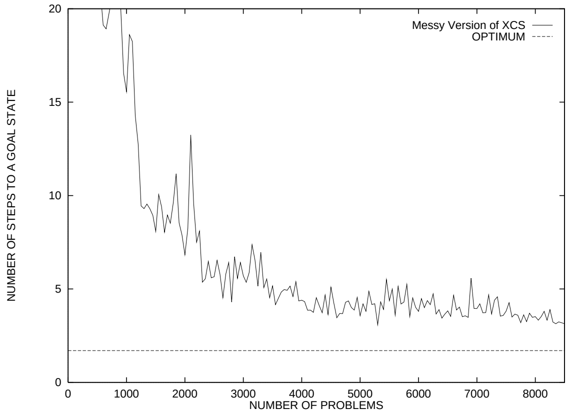

Figure 3: Performance of the first version of XCSm in `Woods1`. Population size is 800 classifiers. The curve is an average over ten runs. Horizontal line represents optimal performance.

6.2 EXPERIMENTAL RESULTS

We now apply the version of XCSm we introduced in the previous section in two environments, `Woods1` (Figure 1) and `Maze4` (Figure 2), to test whether XCSm can learn an optimal policy in these environments. In the first experiments we apply XCSm in `Woods1`. Population size is set to 800 classifiers; general XCS parameters are set as follows: β=0.2, γ=0.71, θ= 25, ε_0=.01, $P_{\#}$=.3 μ=0.01, ϕ=0.5;[1] XCSm specific parameters are set as follows: $P_s = 0.5, p_k = 0.1$. The results reported in Figure 3 show that XCSm can converge to a solution that is quite near to the optimum but it never reaches it. In particular neither can a single run reach the optimum. We have the same type of result when we apply XCSm to `Maze4` with a population size of 1600 classifiers and the same parameter setting we used in the previous experiment. The results depicted in Figure 4 show that XCSm cannot reach the optimum in `Maze4`.

[1] Some of these parameters have not been presented in the overview in Section 2 but are reported here for sake of completeness. We refer the interested reader to Wilson's original paper [12] for a complete discussion of XCS parameters.

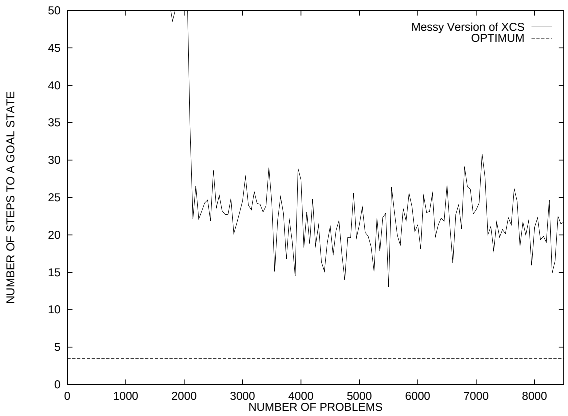

Figure 4: Performance of the first version of XCSm in `Maze4`. Population size is 1600 classifiers. The curve is an average over ten runs. Horizontal line represents optimal performance.

7 HOW TO IMPROVE XCSm

We now analyze the results presented in the previous section and show what are the causes that prevent XCSm from reaching optimal performance.

7.1 ANALYSIS OF THE RESULTS

Analyzing the final populations evolved during the previous experiments we observe that most of the classifiers which represent the final solutions are overgeneral. In particular we noted that only few sensors were used by most classifiers while the rest of them were left unspecified in the majority of the conditions. This may suggest that the cut-and-splice policy XCSm employs to recombine classifiers cause a proliferation of messy genes that match a limited set of sensors. However, different cut-and-splice policies that we evaluated did not cause any significant modification in the overall system performance in the two environments.

The bet of covering. Since from our experiments it seems that optimal performance is not reached because of sensor underspecification, in the first instance, we can force the covering operator to cover almost every sensor and let evolution decide which sensors are important. Accordingly, we repeated the above experiments for different values of probability P_s (we remind the reader that P_s is the probability of covering a single sensor). The experimental results (not reported) show an improvement in both environments for a value of P_s greater than 0.8. However, the performance of XCSm in the two environments is still not optimal. This result can be easily explained. When XCSm covers a certain sensor configuration it makes a bet on what may be important in the future. In XCS, the system bets on single sensory bits: a condition bit can be set to # betting that it will unimportant. During covering XCS takes only few risks because although it would cover all the bits of an important sensors with #s, as the evolution proceeds, genetic operators can recover this situation through crossover, mutation, or specification [9]. On the other hand when XCSm covers it bets: (i) on which sensors will be important in the future by including a messy genes for a certain sensor with probability P_s; and (ii) on which bits of the sensor will be important by means of #s. Note that in XCSm overgeneralization due to #s symbol can still be corrected through evolution like in XCS. However, the case when a classifier is overgeneral because it is underspecified, can be only recovered by the cut-and-splice operator because it is the only operator that can introduce new messy genes in classifier conditions. Briefly, XCSm can easily deal with overgeneralization on sensors, but has difficulty with overgeneralization due to underspecification.

Extended mutation. We have seen that XCSm does not reach optimal performance because in general it cannot deal with overgeneralization due to underspecification. This happens because XCSm can recover an underspecified condition only through cut-and-splice. To improve XCSm' capabilities of dealing with underspecification, we extend the mutation operator so that: (i) tags can be mutated with the same probability μ as value bits; (ii) a messy gene, that matches one of current sensory inputs, can be added to classifier condition with probability μ; (iii) a messy gene can be eliminated with probability μ. This extended version of mutation can deal with underspecification by modifying the tags of existing genes and by adding new messy genes. If we apply XCSm with a high covering probability ($P_s \geq .8$) and extended mutation to the previous environments (results not reported) we find that XCSm can learn an optimal solution in `Woods1`, but not in `Maze4` (results not reported). In particular XCSm performance in `Maze4` suddenly decreases suggesting that, in some way, the population is corrupted.

Dealing with overspecification. If we analyze intermediate populations produced during the experiments with this latest version of XCSm in `Maze4`, we note that when a classifiers is overspecified (i.e. it has multiple messy genes testing one sensor) the genes that are used for matching tend to represent a correct condition for the positions the classifier applies to. On the contrary messy genes that are not tested during matching because they follow other genes that have the same tags, usually test overgeneral conditions over the sensors or otherwise do not apply to the situations the classifier matches. It would seem that the use of the positional precedence policy for matching a classifier condition results in a proliferation of overgeneral or non admissible messy genes. This observation suggests that the unstable performance we observed with XCSm and extended mutation in `Maze4` may be explained by the presence of such overgeneral or non admissible messy genes that for long periods are not checked because of overspecification and suddenly, through cut-and-splice, become active parts in the classifier matching process. To limit this phenomenon we modified the matching procedure so that when a condition is matched against sensory inputs all messy genes are tested on the corresponding sensors. Therefore a condition matches current inputs if all its messy genes match the sensors they test.

7.2 EXPERIMENTS WITH AN EXTENDED XCSm

We now apply the extended version of XCSm we developed so far (which consists of a high covering probability, extended mutation, and new matching procedure) in `Woods1` and `Maze4` to show that this XCSm can learn an optimal solution in both environments; parameters are set like in the original experiments except for covering probability, P_s, that is 0.8. The XCSm performance in `Woods1` depicted in Figure 5 shows that the new version of XCSm can learn an optimal policy for this environment. Optimal performance is also reached in `Maze4` as the results in Figure 6 show.

8 KNOWLEDGE REUSE

One possible advantage of messy representation in learning classifier systems is that it does not bind classifier syntax to a specific sensory configuration. Conditions are defined in terms of which sensors the specific condition matches, not in term of which position of the sensory inputs matches. Accordingly, classifiers evolved to solve a certain problem can be reused in another application assuming that the tags of messy genes are still valid in the new application. We illustrate this idea with an example. Consider the `Maze4` environment that we used in the previous section. First we apply an agent with four sensors, one for each cardinal direction, in `Maze4`. We apply XCSm

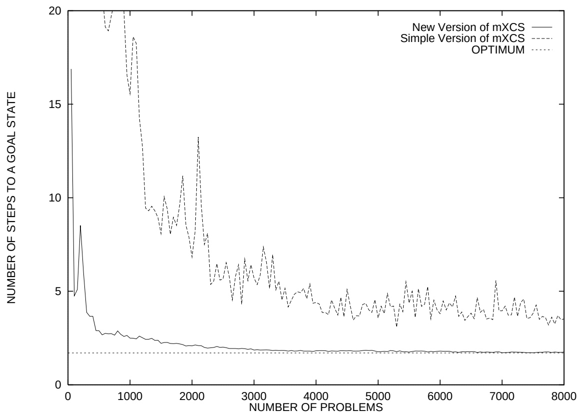

Figure 5: Performance of the enhanced version of XCSm in `Woods1` (solid line) and of the original XCSm (dashed line). Population size is 800 classifiers. The curve is an average over ten runs. Horizontal line represents optimal performance.

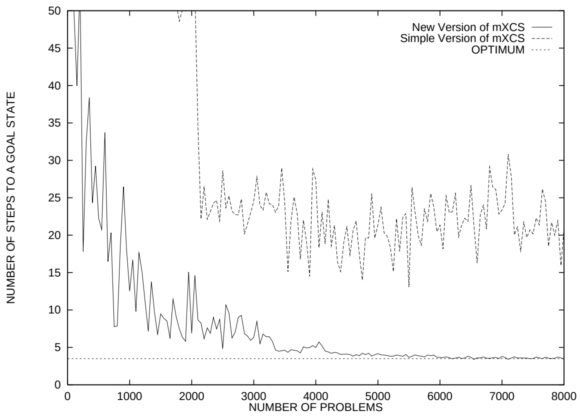

Figure 6: Performance of the enhanced version of XCSm in `Maze4` (solid line) and of the original XCSm (dashed line). Population size is 1600 classifiers. The curve is an average over ten runs. Horizontal line represents optimal performance.

in `Maze4` when only the tags corresponding to the four cardinal sensors are used for 10000 problems.[2] Population size is set to 1600 classifiers. The results depicted in Figure 7 (first 5000 problems) show that the agent cannot learn a good solution for `Maze4` when only four sensors are available.

Suppose now that we "buy" an improved agent that has all the eight possible sensors (one for each adjacent cell). We apply this agent in `Maze4` with the starting population consisting of the previous solution. Note that we *do not* explicitly introduce messy genes that test new sensors in classifiers conditions, but we let mutation use all the new admissible tags. We apply XCSm with eight sensors to `Maze4` for 10000 problems. Results in (last 10000 problems) show that starting from the previous solution, and using *only* mutation, XCSm can evolve an optimal solution for `Maze4`. This result suggests that extending classifier with representations that are independent from the position of the sensors can improve the portability of the behaviors learned between different agents.

9 CONCLUSIONS

We have presented an initial study concerning the possible extensions to the current representation of classifier conditions. In particular, we extended the XCS classifier system by replacing the current fixed-length bitstring representation with a variable-length messy representation that was borrowed from messy genetic algorithms. We applied an initial version of XCS with messy conditions, XCSm, to two environments and we showed that XCSm can reach only near optimal performance. The analysis of these results pointed out that the possible underspecification and overspecification of classifier conditions is a important factor in learning with a variable-length representation. Accordingly, great care must be taken when designing the covering operator, the matching procedure, and the mutation operator. We repeated previous experiments with an enhanced version of XCSm in which covering, matching, and mutation were designed in order to cope with underspecification and overspecification of classifier conditions. Experimental results we reported show that this new version of XCSm can reach the optimum in both environments. Finally, we presented an experiment in which a behavior initially learned by an agent with a small sensory equipment was used by another agent, with an improved sensory equipment, as the basis for learning an improved behavior by exploiting newly available sensors.

Overall we believe that this work contributes in drawing some directions for subsequent explorations of the alternative representations that might be employed in classifier systems. In particular, our results with messy classifier systems will help us in the second part of our research [10] in analyzing and understanding the behavior of the next representation we are going to introduce, which is based on s-expressions.

[2] Note that when the agent has only the four cardinal sensors, there are many different positions in the environment that the agent perceives as identical; briefly we say that with four sensors `Maze4` is partially observable with respect to agent sensors [8]. Accordingly, XCSm cannot evolve an optimal solution in `Maze4` with four sensors.

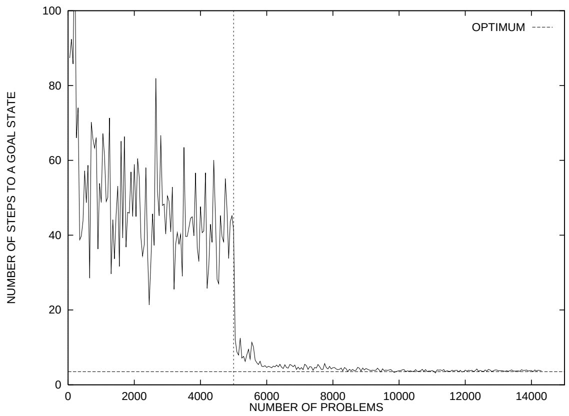

Figure 7: Performance of XCSm in `Maze4`. First 5000 problems are solved by an agent with four sensors. Subsequent 10000 problems are solved by an agent with eight sensors that started to learn from the solution evolved by the former agent.

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.

I wish to thank Andrea Bonarini and Pino Contini who helped me in making this submission possible after that fire devastated part of my department. Many thanks go also Stewart W. Wilson for his invaluable advices and to Tim Kovacs for his comments and corrections.

References

[1] Andrea Bonarini. *Genetic Algorithms and Soft Computing*, chapter Delayed Reinforcement, Fuzzy Q-learning and Fuzzy Logic Controllers, pages 447–465. Physica-Verlag, A Springer-Verlag Company, 1996.

[2] Marco Dorigo and Marco Colombetti. *Robot Shaping: An Experiment in Behavior Engineering.* MIT Press, 1997.

[3] David E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms. motivation, analysis and first results. *Complex Systems*, 3:493–530, 1989.

[4] Frank Hoffmann and G. Pfister. *Genetic Algorithms and Soft Computing*, chapter Learning a Fuzzy Control Rule Base Using Messy Genetic Algorithms, pages 279–305. Physica-Verlag, A Springer-Verlag Company, 1996.

[5] John H. Holland. *Machine learning, an artificial intelligence approach. Volume II*, chapter Escaping Brittleness: The possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based Systems, pages 593–623. Morgan Kaufmann, 1986.

[6] John H. Holmes. Hierarchical exemplar based credit allocation for genetic classifier systems. In J. Koza et al, editor, *Proceedings of the Third Annual Genetic Programming Conference*, pages 622–628, Madison (WI), 1998. Morgan Kaufmann San Francisco (CA).

[7] 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.

[8] Pier Luca Lanzi. An Analysis of the Memory Mechanism of XCS. In J. Koza et al, editor, *Proceedings of the Third Annual Genetic Programming Conference*, pages 643–651, Madison (WI), 1998. Morgan Kaufmann San Francisco (CA).

[9] Pier Luca Lanzi. An analysis of generalization in the xcs classifier system. *Evolutionary Computation Journal*, 1998. To be published.

[10] Pier Luca Lanzi and Alessandro Perrucci. Extending the representation of classifier conditions, part ii: From messy coding to s-expressions. 1999.

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

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

[13] 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).

[14] Stewart W. Wilson. Structure and function of the XCS classifier systems. Available at `http://prediction-dynamics`, 1998.

[15] Stan Franklin Zhaohua Zhang and Dipankar Dasgupta. Metacognition in software agents using classifier systems. In AAAI, editor, *AAAI-98 Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 83–88, Madison (WI), 1998. AAAI-Press and MIT Press.