GENETIC AND NON GENETIC OPERATORS IN ALECSYS *

Marco Dorigo+

TR-92-075 - Revised Version
November 1992

Abstract

It is well known that standard learning classifier systems, when applied to many different domains, exhibit a number of problems: payoff oscillation, difficult to regulate interplay between the reward system and the background genetic algorithm (GA), rule chains instability, default hierarchies instability, are only a few. ALECSYS is a parallel version of a standard learning classifier system (CS), and as such suffers of these same problems. In this paper we propose some innovative solutions to some of these problems. We introduce the following original features. *Mutespec*, a new genetic operator used to specialize potentially useful classifiers. *Energy*, a quantity introduced to measure global convergence in order to apply the genetic algorithm only when the system is close to a steady state. *Dynamical adjustment* of the classifiers set cardinality, in order to speed up the performance phase of the algorithm. We present simulation results of experiments run in a simulated two-dimensional world in which a simple agent learns to follow a light source.

---

* To appear on the Evolutionary Computation Journal, 1993. This work has been partly supported by the Italian National Research Council, under the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo", subproject 2 "Processori dedicati", and under the "Progetto Finalizzato Robotica", subproject 2 "Tema: ALPI".

+ International Computer Science Institute, Berkeley, CA 94704, and Progetto di Intelligenza Artificiale e Robotica, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Piazza Leonardo da Vinci, 32, 20133 Milano, Italy (e-mail: dorigo@icsi.berkeley.edu).

1. Introduction

The goal of this research is two-fold. We want to gain a deeper understanding of some features of the classical learning classifier system (CS, see Holland, 1980, 1986), and we want to improve the classical CS by introducing new genetic and non-genetic operators. Results of this research have been used to improve the performance of ALECSYS (Dorigo, 1992a; Dorigo & Sirtori, 1991), a learning system based on a coarse grain parallel version of the standard CS. The main goal of the ALECSYS project is to learn to control an autonomous robot moving in a real environment (Dorigo & Schnepf, 1993; Dorigo, 1992b; Colombetti & Dorigo, 1992a, 1992b). In this paper the focus is on some technical, from a classifier systems point of view, aspects of ALECSYS. We discuss the following features of our system.

- *Mutespec*, a new genetic operator which is used to reduce reward variance in default classifiers (overly general classifiers can fire in conflicting situations, causing actions which in some situations are useful and in others are not: in these cases more specific classifiers perform better).
- The concept of *energy* is introduced to allow the call of background genetics at steady state (i.e., through *energy* we measure the achievement of a steady state; once in that state, we call the genetic algorithm). We also experimentally investigate the optimal number of classifiers to be introduced by an application of the genetic algorithm (GA).
- The number of activatable classifiers is dynamically changed at run-time (i.e., the cardinality of the set of classifiers is not fixed; instead, it shrinks as the bucket brigade finds out that some classifiers are useless or even dangerous).

All the experiments presented in this paper have been run on the sequential version of ALECSYS. No substantial differences have been found running, later on, the same experiments using the distributed version.

2. The task

Our long term goal is to apply CSs to the control of autonomous robots. So that we can run many experiments in a reasonable length of time, we use a computer simulation of a simplified version of the task our real robot will be required to perform. We have an agent living on the screen of a computer (two dimensional world). The agent can move in eight directions (say the eight cardinal directions), and can choose the step-size between three values: zero, one or two pixels. The agent has four sensors which allow him to perceive the presence of a light source, the *lamp*. Each sensor is a binary device that monitors a semispace; it sets to the ON value whenever the lamp is in the monitored semispace. The agent therefore can discriminate between four lamp positions: North-West, North-East, South-West, South-East. The agent can approach the lamp along three directions (see Figure 1). The reward for the

approaching behavior was set to three different values: +18 for the best move, +6 for the other two possibilities.

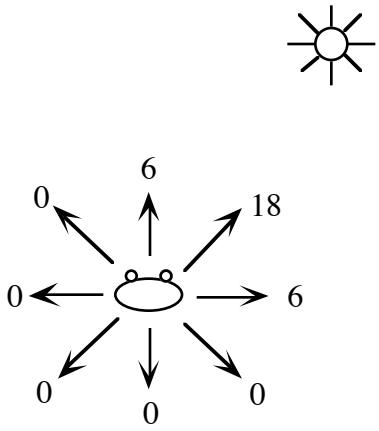

Figure 1. Different rewards for an *approaching the lamp* move.

In ALECSYS classifiers have two conditions and one action. Conditions and actions are strings of length k. Symbols in conditions belong to {0,1,#}, and symbols in actions belong to {0,1}.

Messages were set to be five bits long; four bits code the sensory information, one bit is a tag saying whether a message is a sensors reading or an internal message (i.e., a message sent by other classifiers). A classifier is therefore 15 bits long.

In all experiments, if not otherwise stated, we used a population of n=240 classifiers and the message list length was set to 10. In addition to the new operators, we used classic context operators (cover detector and cover effector, see for example Robertson & Riolo, 1988) and classic genetic algorithm operators (sometimes referred to as *background genetics*, i.e., crossover and mutation). Every classifier C with both conditions matched competes in a auction to gain the right to append a message to the message list. It makes therefore a bid, which in our experiments is given by the formula $Bid_c(t) = constant \cdot spec_c \cdot S_c(t)$, where the constant was set to 0.1, $spec_c$ is the specificity of classifier C (given by the ratio (number of 0 or 1 symbols)/(length of the classifier)), and $S_c(t)$ is the strength of classifier C at time t. Winning classifiers (i.e., those which append their message to the message list) are chosen among bidding classifiers with a probability proportional to their bid. Among the messages on the message list which have an "effector" tag, one is probabilistically chosen (with probability proportional to the strength of the corresponding posting classifier), and sent to the agent's actuators; the classifier which posted this message receives the external reward.

3. The *mutespec* operator

A problem caused by the presence of don't care (#) symbols in classifiers conditions is that the same classifier can receive high rewards when matched by some messages and low rewards (or even punishments if we use negative rewards) when matched by some other messages. We call these classifiers *oscillating classifiers*. Consider the example in Figure 2; the oscillating classifier has a don't care symbol in the third position of the first condition. Suppose that whenever the classifier is matched by a message with a 1 in the position corresponding to the # in the classifier condition the message 1 1 1 1 is useful while when the matching value is a 0 then the message 1 1 1 1 is harmful.

Oscillating classifier: 0 1 # 1 ; 0 1 1 0 -> 1 1 1 1
ML -> 0 1 1 1 implies that message 1 1 1 1 is very useful
ML -> 0 1 0 1 implies that message 1 1 1 1 is harmful

Figure 2. Example of oscillating classifier.

As a result the strength of that classifier cannot converge to a steady state value, but will oscillate between the steady state values that would be reached by the two more specific classifiers in the right side of Figure 3. The major problem with oscillating classifiers is that, on the average, they will be activated too often in situations in which they should not be used and, on the contrary, too seldom when they could be useful. This causes the performance of the system to be lower than it could be.

The *mutespec* operator tries to solve this problem using the oscillating classifier as the parent of two offspring classifiers: one of the offspring will have a 0 in place of the #, the other one a 1 (see Figure 3). This is the optimal solution in case there is a single #. In case the number of # symbols is greater than 1, the # symbol chosen could be the wrong one (i.e., not the one responsible for oscillations); in this case, as the *mutespec* operator did not solve the problem, with high probability the *mutespec* operator will be applied again. The *mutespec* operator can be likened to the mutation operator. The main differences are that it is applied selectively only to oscillating classifiers and that it always mutates # symbols to 0s and 1s. Also, the mutation operator mutates a classifier, while *mutespec* introduces two new, more specific, classifiers (the parent classifier remains in the population).

To decide which classifier should be mutated by the *mutespec* operator we monitor the variance of the rewards each classifier gets. This variance is computed according to the following formula:

$$VAR(R_c(t)) = \frac{1}{t}\sum_{j=1}^{t} R_c^2(j) - \left(\left(\sum_{j=1}^{t} R_c(j)\right) \Big/ t\right)^2$$, where $R_c(t)$ is the reward received by classifier C at time t.

A classifier C is an oscillating classifier if $VAR(R_c) \geq K \cdot AVG\_VAR$, where AVG_VAR is the average variance of the classifiers in the population and K is a user defined parameter (in our experiments we set K=1.25). At every cycle *mutespec* is applied to the oscillating classifier with the highest variance. (If no classifier is an oscillating classifier then *mutespec* is not applied.)

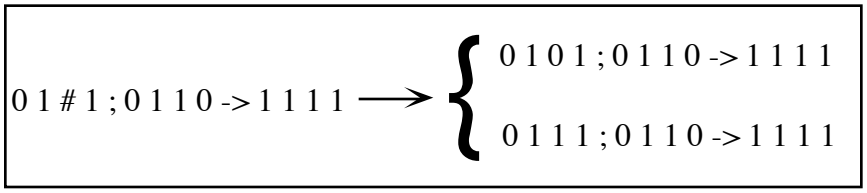

Figure 3. Example of application of *mutespec* operator.

Experiments have been run to verify the positive effects of *mutespec* in the simple environment described in Section 2. Four combinations of operators have been tested. In the first experiment we use only background genetics, in the second background genetics and context operators, in the third background genetics and *mutespec*, and finally in the last both background genetics, context operators and *mutespec*. We call standard CS the CS which uses only background genetics. It works as follows.

- Reproduction: offspring classifiers are uniformly sampled with repetitions from the population of parent classifiers; reproduction probability is proportional to classifiers strength. Every application of the background genetics introduces NC=24 new classifiers (i.e., 10% of the population is replaced) which replace the 10% lower strength classifiers.

- Crossover: we used one-point crossover; the crossing point was chosen with uniformly distributed probability; crossover probability was set to 0.6.

- Mutation: mutation rate was set to 0.04; this means that every bit position of every classifier is mutated with probability 0.04.

Results are presented in Figure 4. Performance is measured as the percentage of correct moves on total moves in the last 100 iterations. (A 100% performance is the maximum achievable by the system.) Background genetics was applied every 400 iterations of the CS. The starting generality of the classifiers (i.e., the percentage of # symbols in the classifier set) was set to 50%. In the graph of Figure 4 every point represents the performance level achieved between iterations 300 and 400, after background genetics was applied (the graph is averaged on 100 runs). This choice shows the behavior of the system when the noise introduced by new classifiers generated by the GA has faded away. The predicted improvement in performance can be observed both when using *mutespec* alone and when together with context operators. It is also interesting to note that *mutespec* alone was more effective than context

operators alone. The use of *mutespec* improves both the performance level achieved and the speed with which that performance is obtained.

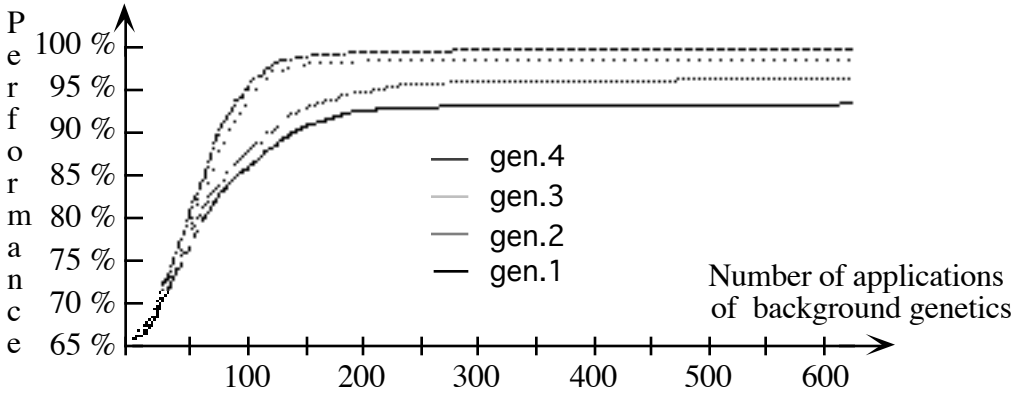

Figure 4. Performance of the CS using differing combinations of genetic and non-genetic operators:
- **gen.1 - background genetics only (i.e., crossover plus mutation),**
- **gen.2 - background genetics + cover detector and cover effector,**
- **gen.3 - background genetics + *mutespec*,**
- **gen.4 - background genetics + cover detector, cover effector and *mutespec*.**

4. Classifier system *energy*

A problem with classifier system is how to decide when to apply the genetic algorithm (GA) to generate new classifiers. In principle the GA should be applied as soon as the system has reached a steady state, i.e. when the strength acquired by each classifier correctly reflects the classifier utility. In practice in existing implementations of CSs the choice has been to apply the GA every IBG (Interval Between Genetics) steps, with IBG experimentally determined. This policy requires some experimental work to find a good value for IBG; moreover, situations in which the optimal value for IBG changes in time cannot be considered. It is therefore advisable to try to introduce an automatic mechanism to detect the attainment of a steady state. A straightforward way to check whether the system is at a steady state could be to track the strength of each classifier. Unfortunately, as we saw in the preceding section, there can be oscillating classifiers, and to wait for the *mutespec* operator to substitute them all with non-oscillating ones can take too long. A way out is to use some kind of aggregate measure which indicates whether the CS as a whole is close to a steady state or not.

We propose to use a quantity we call *energy* to find out when the CS has reached a steady state and consequently apply the GA. We define the energy $E_{CS}(t)$ of a classifier system CS to be the sum of the strengths $S_c(t)$ of all the n classifiers in the classifiers set CS at time t:

$$E_{CS}(t) = \sum_{c=1}^{n} S_c(t)$$

$E_{CS}(t)$ oscillates less than the single classifiers because the various oscillating classifiers do not oscillate synchronously. Ideally, at a steady state the total energy of the system is constant and therefore its variance value is zero. In practice, as can be seen from Figure 5, this situation is never achieved. In Figure 5 we show two graphs. The upper one (and the most oscillating one) is given by the following sum

$$\sum_{c=1}^{n}\left(S_c^2(t) - \overline{S}_c^2(t)\right) \qquad [1]$$

where $\overline{S}_c(t) = \frac{\sum_{j=1}^{L} S_c(t-j)}{L}$). The lower one is given by

$$E_{CS}^2(t) \; - \overline{E}_{CS}^2(t) \qquad [2]$$

where $\overline{E}_{CS}(t) = \frac{\sum_{j=1}^{L} E_{CS}(t-j)}{L}$. (L is a parameter which was set to min {t, 20}.)

It can be seen that the behavior of the energy is much smoother than that of the sum [1].

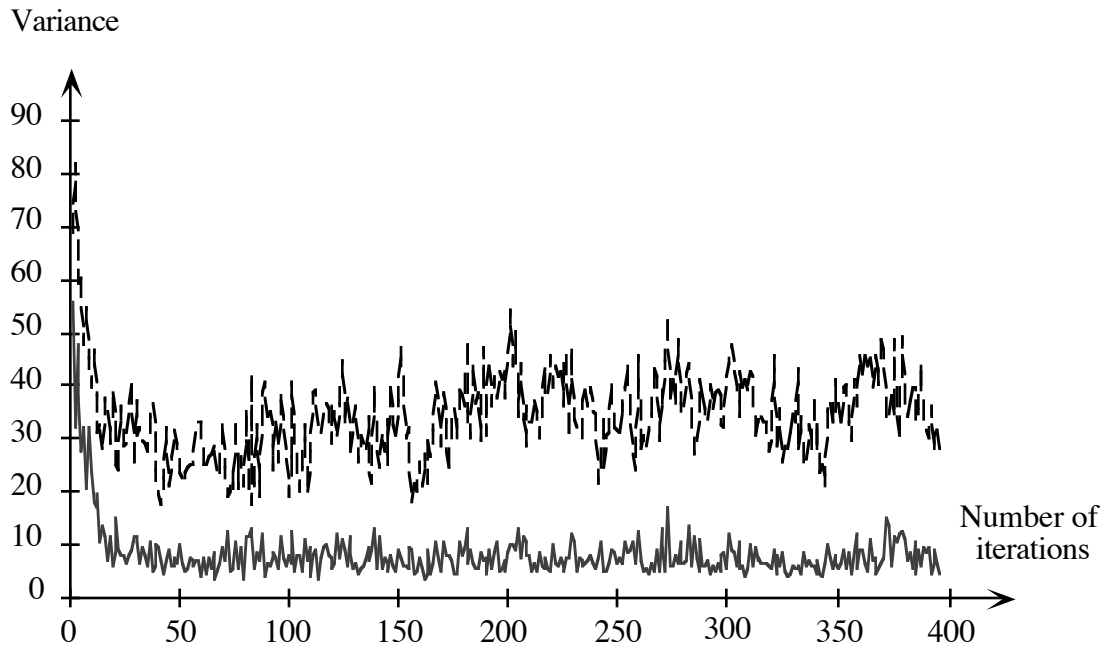

Figure 5. Comparison between the behavior of energy (lower graph, see formula [2]) and the behavior of the sum in formula [1] (upper graph); typical graph.

Operationally, we consider a system to be at a steady state when $E_{CS}(t) \in [E_{Min}, E_{Max}] \quad \forall t \in [t, t-k]$, where $E_{Min} = \min\{E_{CS}(t),\ t \in [t-k, t-2k]\}$, $E_{Max} = \max\{E_{CS}(t),\ t \in [t-k, t-2k]\}$, and k is a parameter. In this way we exclude both cases in which $E_{CS}(t)$ is increasing or decreasing, and situations in which $E_{CS}(t)$ is still oscillating too much. We experimentally found that the value of k is very robust; in our system it was set to 50, but no substantial differences were found for values of k in the range 20÷100. When the system is at a steady state the GA can be applied. We call steady state classifier system (SSCS) the CS that uses energy to decide when to apply the GA.

5. How many classifiers should be substituted by the GA?

Figure 6 shows the typical behavior of energy in SSCS; every insertion of new classifiers in the CS causes a peak because new possibly better classifiers replace useless, and therefore low strength, classifiers. The magnitude of peaks, and their frequency[1], depends on the number NC of new classifiers generated by the GA. Peaks are due to the substitution of low strength classifiers with new classifiers generated by the GA. The strength of new generated classifiers is set to the average value of active classifiers.

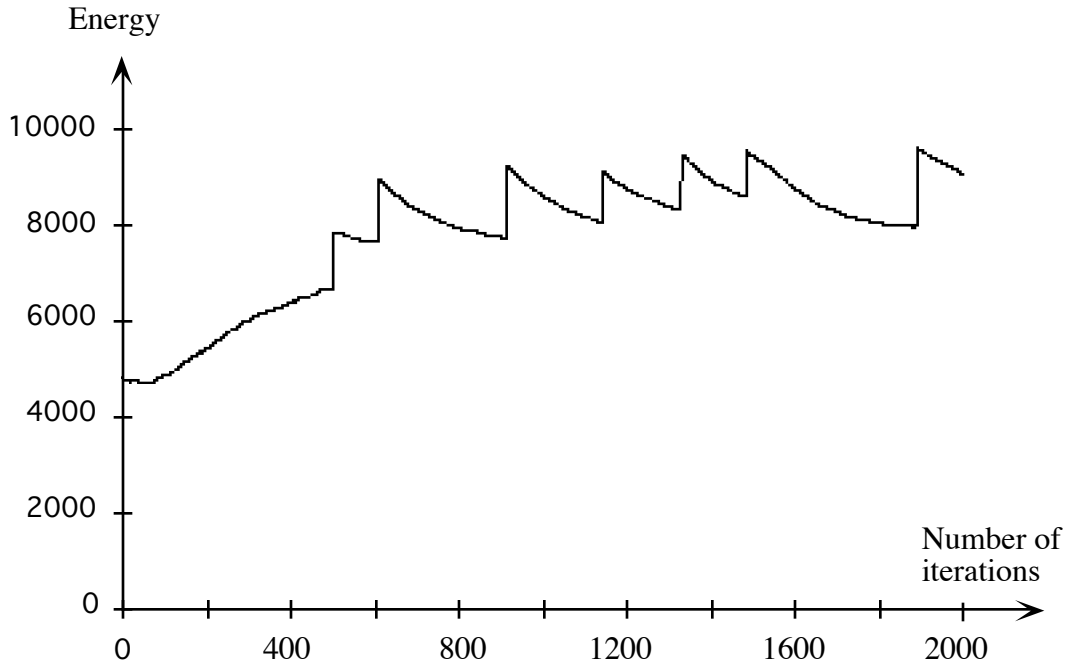

Figure 6. Behavior of the energy function in a typical run. Peaks in the graph are due to the substitution of low strength classifiers with new classifiers generated by the GA.

1 In fact, the higher the number of new classifiers generated by the GA, the higher the peak and the longer the time required to reach a new steady state.

It is interesting to study how the behavior of the system changes when changing NC. We have two opposite effects. With NC set to a high value, a high number of new classifiers will be tested between two calls of the GA, but, as it takes longer to achieve a steady state, the GA is called with a lower frequency. We ran some experiments and evaluated the behavior of SSCS for NC $\in$ {2, 4, 6, 12, 24, 48, 72}. Results are shown in Figures 7, 8, 9, and Table 1. Figures 7, 8, 9 are histograms; each vertical bar indicates the number of iterations of the SSCS before the GA was called. It can be observed that increasing the value of NC increases the average distance (measured by the number of iterations) between two calls of the GA. Still the total number of classifiers generated during the run was maximum for NC=72. Although for NC=72 exploration is favored (the number of new classifiers introduced is maximum, see Table 1), it is important to consider the effects that the introduction of many new classifiers at each GA call has on performance. Results shown in Table 1 show that the introduction of too many classifiers lowers the overall performance. Moreover, the on-line behavior of the system changes abruptly when many new classifiers are inserted at once, and this can be an undesirable property for real world applications. We also compared SSCS with the standard CS, for the same values of NC. The comparison was done both with respect to the total number of classifiers generated and the performance level achieved in a 20000 iterations experiment. Performance was measured as the sum of all the rewards received during the complete run. Results, averaged over 5 trials, are reported in Table 1. As Figure 10 shows (Figure 10 plots the last column of Table 1 for easier visualization of the results) SSCS always outperformed the standard CS. In two cases, for NC=48 and NC=72, SSCS explored a lower number of classifiers.

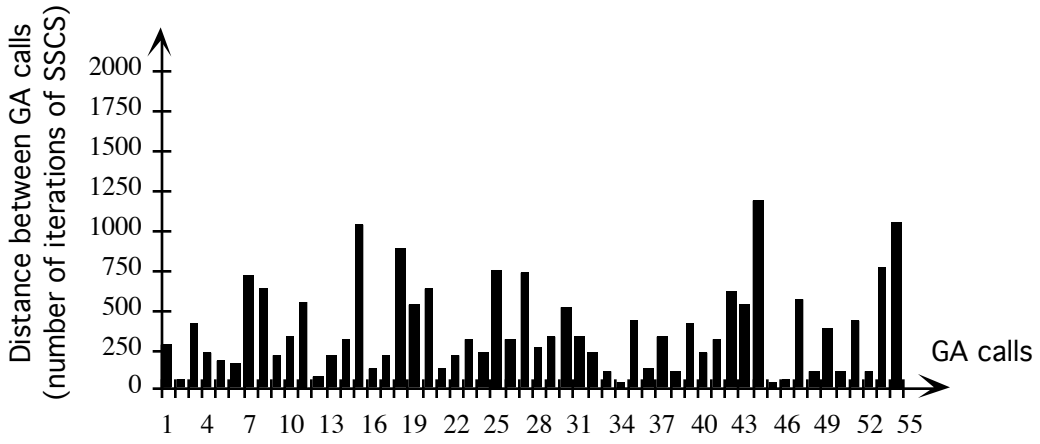

Figure 7. Distance between two GA applications in the case of NC=24. On the horizontal axis we report an index into GA calls in the considered run. In this case the GA was called 55 times.

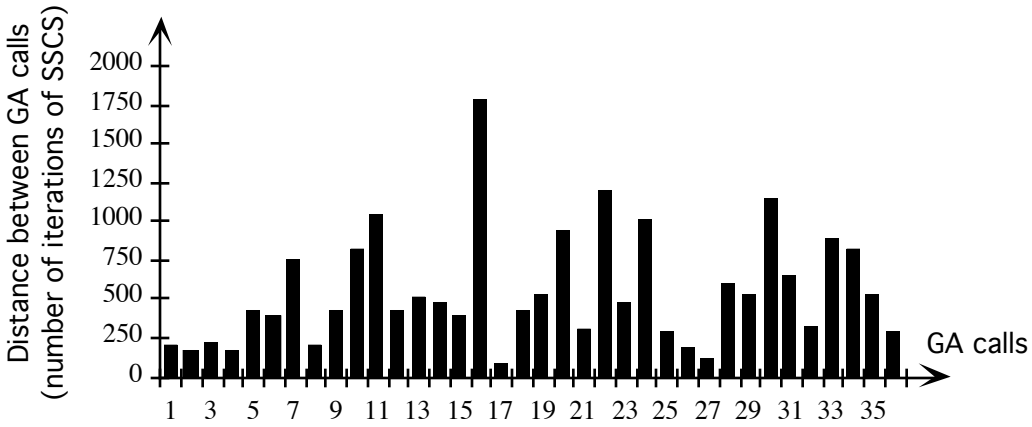

Figure 8. Distance between two GA applications in the case of NC=48. On the horizontal axis we report an index into GA calls in the considered run. In this case the GA was called 36 times.

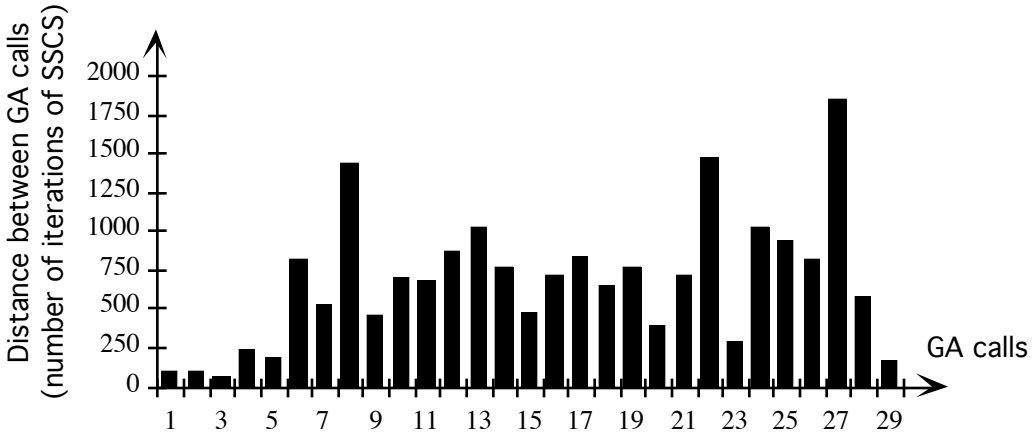

Figure 9. Distance between two GA applications in the case of NC=72. On the horizontal axis we report an index into GA calls in the considered run. In this case the GA was called 29 times.

Table 1. Comparison of SSCS and standard CS for different values of NC.

Type of CS used	NC (number of classifiers introduced by an application of the GA)	Number of iterations between two applications of the GA[2]	Total number of classifiers tested in 20000 iterations	Average performance over 10 trials (best performance was set to 100)
Standard CS	2	500	80	83.7
SSCS	2	201	199	90.5
Standard CS	4	500	160	86.1
SSCS	4	251	319	92.9
Standard CS	6	500	240	83.5
SSCS	6	292	411	94.3
Standard CS	12	500	480	91.4
SSCS	12	293	819	93.0
Standard CS	24	500	960	91.9
SSCS	24	370	1297	96.6
Standard CS	48	500	1920	75.5
SSCS	48	556	1727	100.0
Standard CS	72	500	2880	82.9
SSCS	72	690	2087	91.7

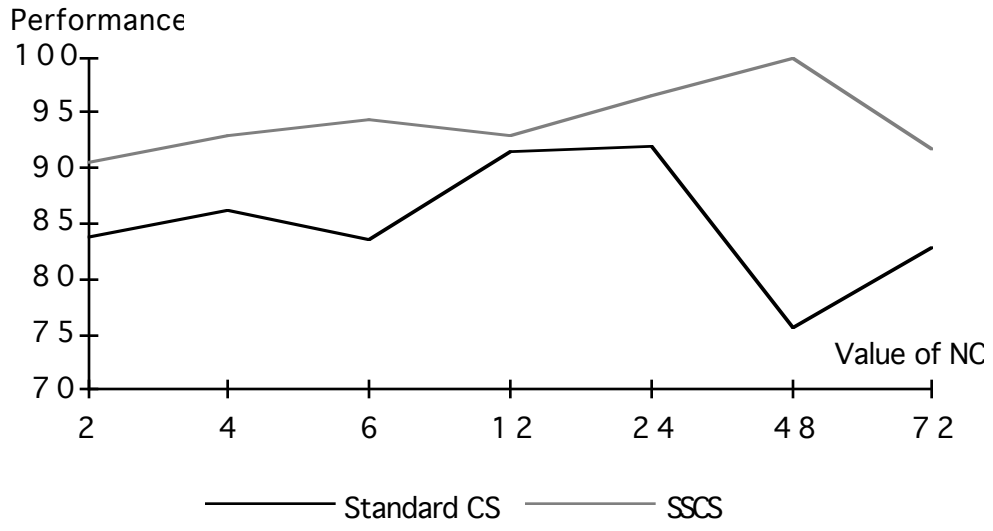

Figure 10. A plot of the last column of Table 1. For all the tested values of NC the SSCS outperformed the standard CS.

[2] For the SSCS we report the average number of cycles. For the standard CS we call the GA every IBG=500 iterations (this value was experimentally found to be optimal).

6. Dynamically changing the number of activatable classifiers

The computational complexity of the matching phase in the performance algorithm of CSs is, on a sequential computer, a linear function of a number of parameters: the number of classifiers in the classifier population, the number of conditions in each classifier, the number of bits in each condition and the message list (ML) length[3]. Unfortunately, the relationship between the values of these parameters and the achievable results (expressed as performance level) is poorly understood. Some hints have been given about the optimal number of classifiers: unfortunately results from Goldberg (1989) and from Robertson & Riolo (1988) are antithetical. We chose in our work a different approach. We propose a method that allows a dynamical change, at runtime, of the number of classifiers used by the CS.

The main idea is to inhibit the matching phase for classifiers with a strength lower than $h \cdot \overline{S}(t)$, where $\overline{S}(t)$ is the average strength of classifiers in the population at time t (this is different from $\overline{S}_c$ of Section 4), and h is a user defined parameter ($0.25 \leq h \leq 0.35$ was experimentally found to be a good range for h). In this way there are two savings. First, the system does not lose time in trying to match conditions of classifiers that would, with very high probability, produce bad actions[4]; therefore, computation time per iteration is reduced, giving faster real time performance. Second, steady state is reached faster, giving more applications of the genetic algorithm in the same number of iterations. What we do therefore is to trade short-term accuracy for computational speed. This computational speed allows us to test a greater number of classifiers.

The utility of this approach is also suggested by examination of the typical distribution of classifiers strengths in early phases of computation compared to those obtained at a steady state (see Figure 11). At the beginning all classifiers have the same strength. As computation goes on classifiers gain or lose strength and classifiers strength in the population is no longer uniform. Figures 11 shows that after 4000 iterations there is already a marked difference between strong and weak classifiers. After 8000 iterations the gap is still bigger[5]. Figure 12 shows the algorithm used to dynamically change the number of classifiers used. It should be noted that this algorithm eliminates only a few classifiers in the early phases of the computation, but many more as computation goes on. Figure 13 shows an example of a run in which h was set to 0.3. It can be observed that after about 1000 iterations the CS dimension dramatically shrinks: it moves from an average dimension of 200 classifiers to just 35 classifiers.

---

[3] To be sure, it is not exactly linear in the number of bits in a message if you match a word at a time; but it is on some average linear in the number of bits of the message.

[4] The system also saves the time required for the auction phase.

[5] The GA was never called in this run.

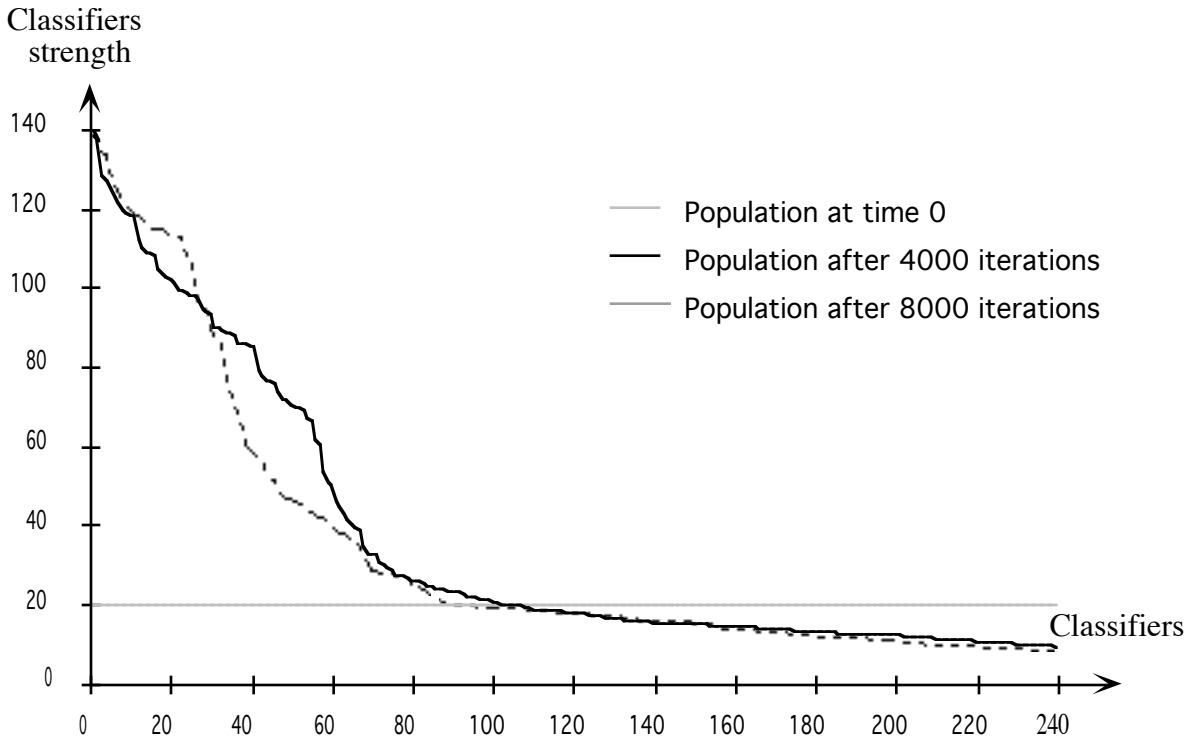

Figure 11. Typical distribution of classifiers strengths at different stages of computation. Classifiers are sorted by strength. On the horizontal axis we report an index into the set of classifiers ordered by strength.

Let **CS** be the classifiers set, CS_i a classifier $\in$ **CS**, and h a user defined parameter.

CS_Usable $\leftarrow$ |**CS**|; /* CS_Usable is the set of classifier actually used*/
AVG_CS $\leftarrow$ AVG(CS_Usable); /*the average strength of classifiers in CS_Usable is computed*/
CS_Usable $\leftarrow$ $\{CS_i : CS_i \in$ CS_Usable and Strength(CS_i) $\geq h \cdot$ AVG_CS$\}$;

Figure 12. The algorithm used to dynamically change the number of classifiers used.

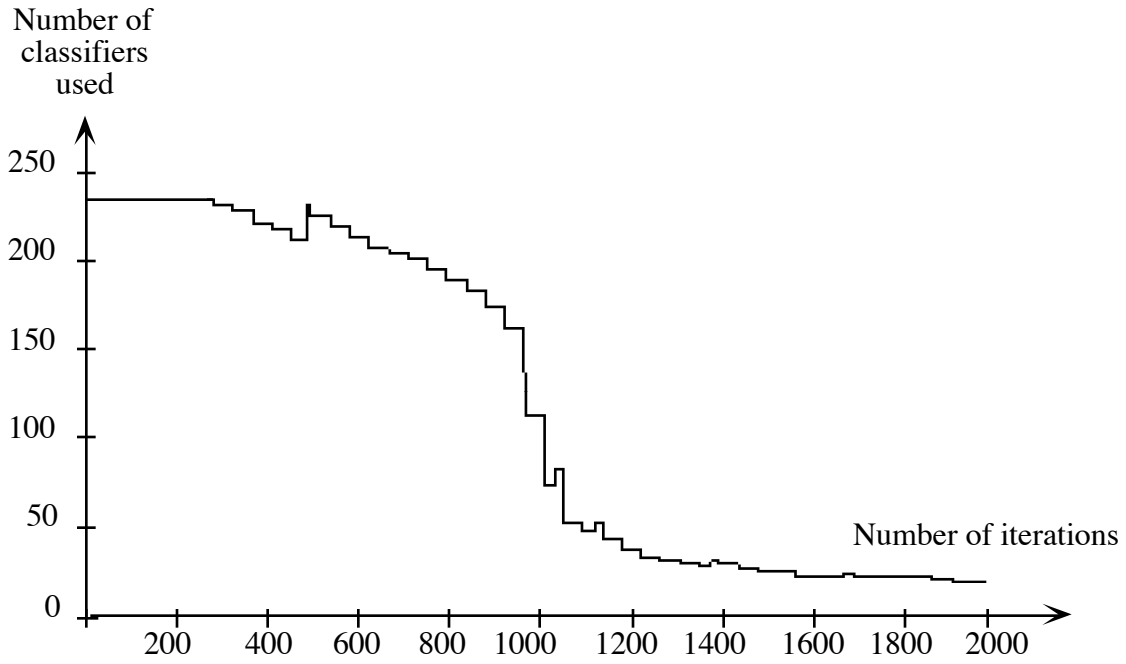

Figure 13. Number of used classifiers in a typical run.

In Table 2 we compare a system in which all classifiers are always used with a system in which the number of classifiers is reduced dynamically; for all the observed performance indexes the second system was the best one.

Table 2. Comparison between a standard CS and a CS with dynamically changing number of classifiers (8000 iterations per run).

	Maximum performance achieved (100% is the maximum performance achievable)	Number of iterations required to achieve 67 % performance	Time required to run 8000 iterations (seconds)	Average time of an iteration (seconds)
Standard CS	78 %	3200	6000	0.75
CS with dynamically changing number of classifiers	92 %	1800	3120	0.39

7. Related work and discussion

Classifier systems have, up to now, seen only a limited number of real world applications (examples of such applications are Frey & Slate's (1991) letter recognition and Colombetti & Dorigo's (1992a, 1992b) learning robot (Dorigo,

1992b). The main reasons for this shortage of complex applications is the slow rate of learning presented by standard implementations of CSs, which often makes working with real world problems unfeasible. For this reason, most of CSs application are in the realm of simulation (e.g., see Grefenstette, Ramsey & Schultz, 1990; Zhou, 1990; Booker, 1988), which can be run for days with a limited experimental effort. In our work we followed two directions to improve the applicability of CSs to real world problems. One direction, not considered in this paper (but see Colombetti & Dorigo, 1992a, 1992b; Dorigo, 1992a, 1992b) is that of using many interacting CSs running in parallel on different processors of a MIMD architecture. This is an engineering approach, in which design choices are based on both ethological plausibility and efficient engineering solutions.

The other direction, whose results have been presented in this paper, is that of improving the efficiency of a single learning classifier system. Some work in this same direction has been done by other researchers. For example Zhou (1990) introduced a mechanism to give a CS some memory and generalization capabilities, while Grefenstette, Ramsey & Schultz (1990) introduced a *specialize* operator[6]. In learning classifier systems research the problem of how to partition efficiently the state space has been there from the very beginning (see for example Holland 1986; Riolo, 1987; 1989). It has been suggested that the most efficient way to partition the state space is by default hierarchies (Holland, Holyoak, Nisbett & Thagard, 1986). A default hierarchy is a multi-level structure composed of rules (classifiers) with different degree of generality. General rules cover broad set of states, while particular rules can respond to very specific states in which the general rules are wrong. I.e., a general rule covers default situations, while a specific rule covers exceptions. To evolve and maintain default hierarchies has proven difficult (Wilson, 1988; Smith & Goldberg, 1991; Wilson & Goldberg, 1989). The author has therefore chosen a different approach, in which default hierarchies have a less prominent role. In this approach the search space is kept small by design. This means that difficult problems are solved by designing many different learning classifier systems, each one devoted to the solution of a single smaller problem which can either be a basic problem (and we call these CSs *basic classifiers*, like the light following CS we developed as example in this paper), or a higher level problem, like learning the coordination of basic behaviors (more on this in Dorigo & Schnepf (1993), in Dorigo (1992b), and in Colombetti & Dorigo (1992a, 1992b). In this perspective, default hierarchies are, although still potentially useful, much less important, and *mutespec* seems therefore to be the ideal operator; in fact, it speeds up convergence towards high performance rule sets. Although it certainly does not help default hierarchies formation, experiments have shown that the developed rule set is not an isomorphic model (i.e., a model where all rules are maximally specific), and that some general rules are still there.

---

6 *Specialize* is different from *mutespec* because it is applied to general rules which fired during a successful episode. Moreover, *specialize* is applied to high-level symbolic rules.

To the author knowledge, no previous work has been done on dynamically changing the number of rules in the classifiers set and on the automatic detection of a steady state. It has been sustained that low strength individuals are a pool from which the system can pick up rules when no high strength rule is available. We have found that it is more efficient to use cover operators to solve such situations. Again, this result must be evaluated in the context of our approach; we let a CS solve a simple (i.e., small search space) problem. This result might not transfer to more difficult tasks.

8. Conclusions

A number of problems are still open regarding learning classifier systems. In this paper we reported about research aimed at shedding some light on some of them. We investigated a dynamical mechanism to shrink the classifier set by disabling the matching phase for classifiers judged useless by the bucket brigade. The use of this mechanism improved the efficiency of the algorithm and permitted the testing of a greater amount of classifiers, with the final result of quicker convergence to higher values of performance.

A new genetic operator was introduced, *mutespec*, to reduce the annoying presence of *oscillating* classifiers (i.e., classifiers whose reward is highly context sensitive). Although mutespec was shown to be useful for the robotic task used as example in this paper, it is not clear whether it is useful in the general case.

In fact, its use tends to drive the system away from using default hierarchies. We introduced mutespec mainly because we assumed we were going to use ALECSYS as a tool to build distributed learning systems composed of many cooperating CSs whose tasks were chosen to be easy.

We also introduced *energy*, a quantity used to ease the evaluation of attainment of a steady state. Using energy we are able to call the genetic algorithm at the right moment, increasing in this way its efficacy.

As we said, the improvements to the standard CS introduced in this paper are part of the distributed version of ALECSYS, which is currently being investigated on more complex real and simulated robotic tasks.

Acknowledgments

I would like to thank Rick Riolo and the three referees for their valuable comments.

References

Booker L., 1988. Classifier Systems that Learn Internal World Models. *Machine Learning*, **3**, 3, 161–192.

Colombetti M., & M. Dorigo, 1992a. Robot Shaping: Developing Situated Agents through Learning. Technical Report 92–040, International Computer Science Institute, Berkeley, CA.

Colombetti M., & M. Dorigo, 1992b. Learning to Control an Autonomous Robot by Distributed Genetic Algorithms. *Proceedings of the Second International Conference on Simulation of Adaptive Behavior (From Animals To Animats–SAB92)*, Honolulu, December 7–11, 1992.

Dorigo M., 1992a. Using Transputers to Increase Speed and Flexibility of Genetics-based Machine Learning Systems. *Microprocessing and Microprogramming Journal*, 34, 147–152.

Dorigo M., 1992b. ALECSYS and the AutonoMouse: Learning to control a real robot by distributed classifier systems. Technical Report 92–011, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milano, Italy.

Dorigo M. & U. Schnepf, 1993. Genetics-based machine learning and behavior-based robotics: A New Synthesis. *IEEE Transactions on Systems, Man, and Cybernetics*, **SMC–23**, 1.

Dorigo M. & E. Sirtori, 1991. ALECSYS: A parallel laboratory for learning Classifier systems, *Proceedings of Fourth International Conference on Genetic Algorithms*, R.K. Belew & L.B. Booker (Eds), Morgan Kaufmann, San Diego, California.

Frey P.W. & D.J. Slate, 1991. Letter recognition using Holland-style adaptive classifiers. *Machine Learning*, **6**, 2, 161–182.

Goldberg D.E., 1989. Sizing populations for serial and parallel Genetic Algorithms. *Proceedings of the Third International Conference on Genetic Algorithms*, J.D. Schaffer (Ed.), Morgan Kaufmann.

Grefenstette J.J., C.L. Ramsey & A.C. Schultz, 1990. Learning sequential decision rules using simulation models and competition. *Machine Learning*, **5**, 4, 355–382.

Holland J.H., 1980. Adaptive algorithms for discovering and using general patterns in growing knowledge-bases. *International Journal of Policy Analysis and Information Systems*, **4**, 217–240.

Holland J.H., 1986. Escaping Brittleness: The possibilities of general-purpouse learning algorithms applied to parallel rule-based systems. In *Machine learning: An artificial intelligence approach*, Vol.II, R.S. Michalski, J.G. Carbonell, & T.M. Mitchell (Eds), Morgan Kaufmann.

Holland J.H., K.J. Holyoak, R.E. Nisbett & P.R. Thagard, 1986. *Induction: Processes of inference, learning and discovery*. MIT Press.

Riolo R.L., 1987. Bucket Brigade performance: II. Default Hierarchies. *Proceedings of the Second International Conference on Genetic Algorithms*, J.J. Grefenstette (Ed.), Lawrence Erlbaum.

Riolo R.L., 1989. The emergence of default hierarchies in learning classifier systems. *Proceedings of the Third International Conference on Genetic Algorithms*, J.D. Schaffer (Ed.), Morgan Kaufmann.

Robertson G.G., & R. L. Riolo, 1988. A tale of two classifier systems. *Machine Learning*, **3**, 2-3, 139–160.

Smith R.E. & D.E. Goldberg, 1991. Variable default hierarchy separation in a classifier system. In *Foundations of Genetic Algorithms* (G.J.E. Rawlins, Ed.), Morgan Kaufmann, 148–167.

Wilson, S.W. 1988. Bid competition and specificity reconsidered. *Complex Systems*, **2**, 6, 705–723.

Wilson & Goldberg, 1989. A critical review of classifier systems. *Proceedings of the Third International Conference on Genetic Algorithms*, J.D. Schaffer (Ed.), Morgan Kaufmann, 244–255.

Zhou H.H., 1990. CSM: A computational model of cumulative learning. *Machine Learning*, **5**, 4, 383–406.