A Comparison between ATNoSFERES and XCSM

Samuel Landau[*] **Sébastien Picault**[*] **Olivier Sigaud**[*] **Pierre Gérard**[*,**]

[*]Laboratoire d'Informatique de Paris 6
8 rue du Capitaine Scott
75 015 Paris

[**]Dassault Aviation
DGT/DPR/ESA
78, Quai Marcel Dassault
92552 St-Cloud Cedex

Abstract

In this paper we present ATNoSFERES, a new framework based on an indirect encoding Genetic Algorithm which builds finite-state automata controllers able to deal with perceptual aliasing. We compare it with XCSM, a memory-based extension of the most studied Learning Classifier System, XCS, through a benchmark experiment. We then discuss the assets and drawbacks of ATNoSFERES in the context of that comparison.

Keywords

Evolutionary Algorithms, Learning Classifier Systems, perceptual aliazing, Augmented Transition Networks

1 Introduction

Most Learning Classifier Systems (LCS) (5) are used to tackle problems where situated and adaptive agents are involved in a sensori-motor loop with their environment. Such agents perceive situations through their sensors as vectors of several attributes, each representing a perceived feature. The task of the agents is to *learn* the optimal policy – *i.e.* which action to perform in every situation, in order to fulfill their goals the best way they can. Like in the general *Reinforcement Learning* (RL) framework (17), the goals of LCS are defined by scalar rewards provided by the environment. The policy is defined by a set of rules – or classifiers – specifying which action to choose according to *conditions* about the perceived situations.

In real world environments, it may happen that agents perceive the same situation in several different locations, some requiring different optimal actions, giving rise to *perceptual aliazing* problems. In such cases, the environment is said *non-Markov*, and agents cannot perform optimally if their decision at a given time step only depends on their perceptions at the same time step. Though they are more often used to solve Markov problems, there are several attempts to apply LCS to non-Markov problems, like (18, 10) for instance.

Within this framework, explicit internal states were added to the classical (condition, action) pair of the classifiers (11, 10, 20). These internal states provide additional information to choose the optimal action when the problem is non-Markov. The problem of properly setting the classifiers, and setting the internal states in particular, is devoted to *Genetic Algorithms* (GA).

In this paper, we will compare LCS to "ATNoSFERES", a new system that also uses GA to automatically design the behavior of agents facing problems in which they perceive situations as vectors of attributes, and have to select actions in order to fulfill their goals, in non-Markov environments. In ATNoSFERES, the goals are defined thanks to a *fitness* measure.

In the first section, we present the features and properties of the ATNoSFERES model (9, 15). It relies upon oriented, labeled graphs (§ 2.1) for describing the behavior and the action selection procedure. The specificity of the model consists in building this graph from a bitstring (§ 2.2) that can be handled exactly like any other bitstring of a Genetic Algorithm, with additional operators. Then we show that the graph-based representation is formally very similar to LCS representations, and, in particular, to XCSM (§ 3.2); thus we compare both approaches through classical experiments (§4). As a result of this comparison, we discuss the assets and drawbacks of both representations according to different criteria (§5). Finally, we conclude by stating what should be added to ATNoSFERES so as to improve it further where the comparison is not in its favor.

2 Description of ATNoSFERES

2.1 Graph-based expression of behaviors

The architecture provided by our model involves an "Augmented Transition Networks" (ATN)-like graph (21) which is basically an oriented, labeled graph with a Start (or initial) node and an End (or final) node (see figure 5). Nodes represent states and edges represent transitions of an automaton.

Such graphs have already been used for describing the behavior of agents (9). The labels on edges specify a set of conditions (*e.g.* `c1 c3 ?`) that have to be fulfilled to enable the edge, and in a sequence of actions (*e.g.* `a5 a2 a4!`) that are performed when the edge is chosen. We use those graphs as follows:

- At the beginning (when the agent is initialized), the agent is in the *Start* node (S).
- At each time step, the agent crosses an edge:
  1. It computes the set of eligible edges among those starting from the current node. An edge is eligible when either it has no condition label or all the conditions on its label are simultaneously true.
  2. If the set is empty, then an action is chosen randomly; else an edge is randomly chosen in the set.
  3. The edge occurs by performing the actions on the label of the current edge. When the action part of the label is empty, an action is chosen randomly.
  4. The new current node becomes the destination of the edge.
- The agent stops when it is in the *End* node (E).

Note that most of behavioral structures involved in classical evolutionary approaches, *e.g.* program trees in Genetic Programming (7), are entirely interpreted at each time step to determine the actions to perform. It is not the case in our approach which relies on internal nodes. An example of the perception-action cycle performed during each time step is given further on figure 3.

2.2 The graph-building process

The behavioral graph is built from an hereditary substrate, by adding nodes and edges to a basic structure containing only the *Start* and *End* nodes.

There are many different evolutionary techniques to automatically design structures such as finite-state machines (2), neural networks (22) or program trees (7). Very roughly, we can sketch an opposition between, on the one hand, approaches that use the genotype as an encoding of a set of parameters (like Genetic Algorithms (5, 1, 3) or Evolution Stategies (16)) and, on the other hand, approaches that use the genotype as a structure producing the phenotype (such as Genetic Programming (7, 14), Evolutionary Programming (2), L-systems (12), developmental program trees (6, 4, 13)...).

In the ATNoSFERES model, we try to conciliate advantages from both kind of approaches: on the one hand, since the behavioral phenotype is produced by the interpretation of a graph, we want it to be of any complexity; on the other hand, we use a fine-grain genotype (a bitstring) to produce it, in order to allow a gradual exploration of the solution space through "blind" genetic operators.

Therefore, we follow a two-step process (see figure 1) [1]:

1. The bitstring (genotype) is translated into a sequence of tokens.
2. The tokens are interpreted as instructions of a robust programming language, dedicated to graph building.

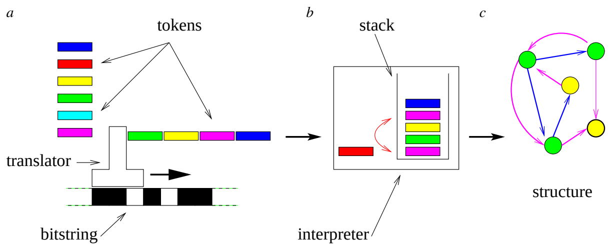

Figure 1: The principles of the genetic expression we use to produce the behavioral graph from the bitstring genotype. The string is first decoded into tokens (a), which are interpreted in a second step as instructions (b) to create nodes, edges, and labels (c).

2.2.1 Translation

Translation is a simple process that reads the bitstring genotype and decodes it into a sequence of *tokens* (symbols). It uses therefore a *genetic code*, i.e. a function $\mathcal{G} : \{0,1\}^n \longrightarrow \mathcal{T}$ $(|\mathcal{T}| \leq 2^n)$ where $\mathcal{T}$ is the set

[1] More details about those mechanisms and the nature of the tokens are provided in (9, 15)

of possible tokens (the different roles of which will be described in the next paragraph). Depending on the number of available tokens, the genetic code might be more or less redundant. Binary substrings of size n (decoded into a token each) are called "codons".

2.2.2 Interpretation

Tokens are instructions of the ATNoSFERES graph-building language. They operate on a stack in which data tokens or parts of the future graph are stored. All tokens fall into the following categories:

- condition or action tokens, which only put data in the stack, which will be used to label edges between nodes;
- node creation or node connection tokens (the latter use nodes and action/condition tokens already in the stack);
- stack manipulation tokens (swap, copy...) which have an action upon the stack containing nodes and action/condition tokens.

In order to cope with a "blind" evolutionary process (i.e. based on random mutations on a fine-grain genotype), the graph built has to be robust to mutations (15). For instance, the replacement of a token by another, or its deletion, should only have a *local impact*, rather than transforming the whole graph.

If an instruction cannot be executed successfully, it is simply ignored; for the same reasons, when all tokens have been interpreted, the graph is made consistent, *e.g.* by linking some nodes to Start/End nodes. Any sequence of tokens is meaningful, thus the graph-building language is robust to variations affecting the genotype (there is no specific syntactical nor semantical constraint on the genetic operators).

2.3 Integration into an evolutionary framework

In this paper, the ATNoSFERES model has been applied to produce agents behaviors within an evolutionary algorithm.

Therefore, each agent has a bitstring genotype from which it can produce a graph (the genetic code depends on the perception abilities of the agent and on the actions it can perform). The fitness of each agent is computed by evaluating its behavior in an environment. Then individuals are selected depending on their fitness and bred to produce offspring.

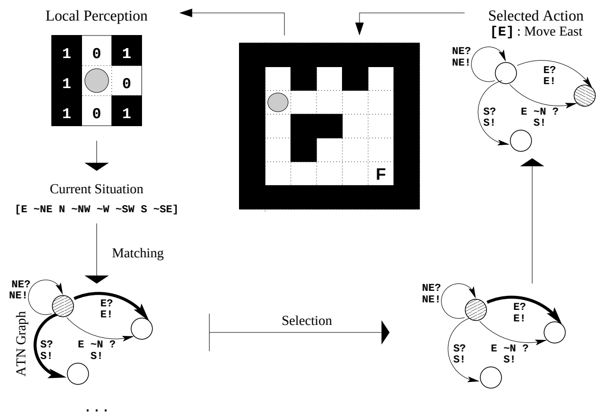

Figure 2: In this example, the agent, located in a cell of the maze, perceives the presence/absence of blocks in each of the eight surrounding cells. It has to decide whether to try to move towards one of the eight adjacent cells. From its current location, the agent perceives `[E ¬NE N ¬NW ¬W ¬SW S ¬SE]` (token `E` is true when the east cell is empty). From the current state (node) of its graph, two edges (in bold) are eligible, since the condition part of their label match the perceptions. One is randomly selected, then its action part (move East) is performed and the current state is updated.

The genotype of the offspring is produced by a classical crossover operation between the genotypes of the parents. Additionally, we use two different mutation strategies to introduce variations into the genotype of new individuals: classical bit-flipping mutations, and random insertions or deletions of one codon. This modifies the sequence of tokens that will be produced by translation, so that the complexity of the graph itself may change. Nodes or edges can in fact be added or removed by the evolutionary process, as can condition/action labels.

3 Learning Classifier Systems

As explained in the introduction, the problems tackled by LCS are characterized by the fact that situations are defined by several attributes representing perceivable properties of the environment. A LCS has to learn classifiers, which define the behavior of the system as shown in figure 3. Within the LCS framework, the use of *don't care* symbols "#" in the condition parts of the classifiers results in generalization, since *don't care* symbols make it possible to use a single description to describe several situations. Indeed, a *don't care* symbol *matches* any particular value of the considered

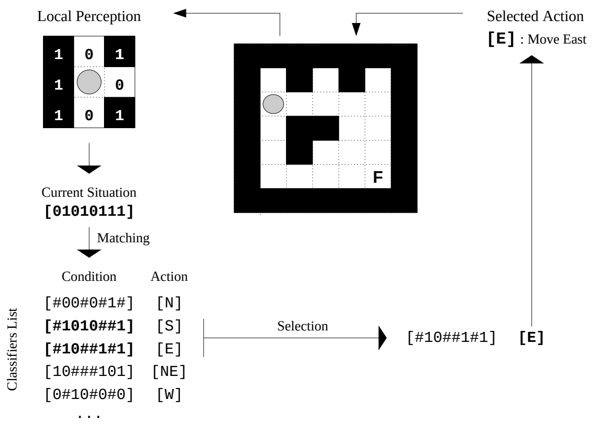

Figure 3: The agent perceives the presence/absence (resp. 1/0) of blocks in each of the eight surrounding cells (considered clockwise, starting with the north cell). Thus from its current location, the agent perceives `[01010111]`. Within the list of classifiers characterizing it, the LCS first selects those matching the current situation. Then, it selects one of the matching classifiers and the corresponding action is performed.

attribute.

The main issue with generalization is to organize conditions and actions so that the *don't care* symbols are well placed. To do so, LCS usually call upon a GA.

In the *Pittsburg* style, the GA evolves a population of LCS with their whole lists of classifiers. The lists of classifiers are combined thanks to crossover operators and modified with mutations. The LCS are evaluated according to a fitness measure and the more efficient ones – with respect to the fitness – are kept. Thus, like in the ATNoSFERES model, a Pittsburg style LCS evolves a population of controllers.

On the contrary, in the *Michigan* style, the GA evolves a population of classifiers within the list of classifiers of a single agent. Here, this is the classifiers which are combined and modified. A fitness is associated to each classifier and the best ones are kept. Thus Michigan style LCS use GA to perform online learning: the classifiers are improved during the life time of the agent. Usually, such LCS rely on utility functions that depend on scalar rewards given by the environment, as defined in the RL framework (17).

In most of the early LCS (5), the fitness was defined directly according to the utility associated to the classifier. After having defined a very simple LCS called ZCS in (19), Wilson found much more efficient to define the fitness according to the accuracy of the utility prediction. Its system, XCS (20), is now the most widely used LCS to solve Markov problems.

3.1 XCSM

Dealing with simple `Condition-Action` classifiers does not endow an agent with the ability to behave optimally in perceptually aliazed problems. In this kind of problems, it may happen that the current perception does not provide enough information to always choose the optimal action: as soon as the agent perceives the same situation in different states, it will choose the same action though this action may be inappropriate in some of these states (see figure 4).

For such problems, it is necessary to introduce internal states in the LCS. Tomlinson and Bull (18) proposed a way to probalistically link classifiers in order to bridge aliazed situations. Lanzi (10) proposed XCSM, where M stands for Memory, as an extension of XCS with explicit internal states. XCSM manages an internal memory register composed of several bits that explicitely represent the internal state of the LCS. Therefore, a classifier contains four parts (*cf.* table 1) an external condition about the situation, an internal condition about the internal state, an external action to perform in the environment and an internal action that may modify the internal state.

The internal condition and the internal action contain as many attributes as there are bits in the memory register. In order to be selected by the LCS, a classifier has to match with both external and internal conditions. When it is selected, the LCS performs the corresponding action in the environment and modifies the internal state if the internal action is not composed only of *don't change* symbols "`#`". When a classifier is fired, a *don't care* symbol in the internal action results in letting the corresponding bit in the memory register at its value before applying the classifier. As XCS, XCSM draws benefits from generalization in the external condition, but also in the internal condition and the internal action.

The memory register provides XCSM with more than just the environmental perceptions. It permits to deal with perceptual aliazing by adding information from the past experience of the agent.

3.2 Formal relations between ATNoSFERES and Learning Classifier Systems

An ATN such as those evolved by ATNoSFERES can be translated into a list of classifiers, whether they have been obtained through a Michigan or a Pittsburgh style process. The nodes of the ATN play the role of internal states and permit ATNoSFERES to deal with perceptual aliazing. The edges of the ATN carry several informations which can be translated in a rule-based formalism: the source and destination nodes of the edge can be respectively represented by an internal condition and an internal action; the conditions associated to the edges correspond to the external conditions of the classifiers; the actions associated to the edges correspond to the external actions of the classifiers.

It is clear in our example that an important difference between both formalisms is due to the possibility to perform *a sequence of actions* (such as `a3`–`a5`) as a consequence of matching conditions. We restricted this feature to a single action in the experiments described below (§ 4.3).

There are two other differences, that have been kept in our experiments:

- When the action part of the edge label is empty (represented by a `#` on the graphs), an action is randomly chosen among possible ones. We represent it by a classifier containing only `#` in the LCS-like formalism. The consequences of that feature will be discussed in §5.
- In XCSM, the "internal state" is regarded as an extension, while it is an inherent feature of the graph-based approach. Hence XCSM may have general rules that match in any situation (whatever the internal state can be, i.e. `#`).

4 Experiments

4.1 The perceptual aliazing problem

In some environments (like `Maze10` on figure 4), some states may induce identical perceptions by the agent, though different actions must be performed. This defines the "perceptual aliazing" issue that is frequently encountered in real-world environments.

We have compared the nature of the results that have been obtained through Evolution to those produced by a LCS like XCSM (10) in the `Maze10` environment.

4.2 Experimental setup

We tried to reproduce an experimental setup close to that used in Lanzi (10) with the `Maze10` environment, with regards to the specificities of our model.

The agents used for the experiments are able to perceive the presence/absence of blocks in the eight adjacent cells of the grid. They can move in those adjacent

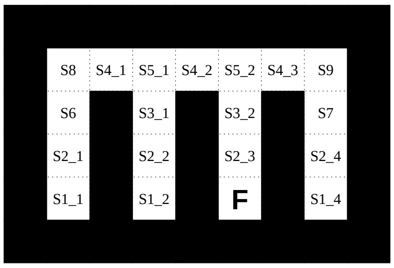

Figure 4: The Maze10 environment. **F** represents the goal to reach (food) from any cell of the maze; a few cells are unambiguous (S_i) but in the other ones the same perceptual situations may require either similar actions or different ones (*e.g.* go north in $S_{2\_\{1,2,4\}}$ but go south in S_{2_3})

cells (the move will be effective when the cell is empty or contains food). Thus the genetic code includes 16 condition and 8 action tokens. In order to encode 24 condition-action tokens together with 7 stack manipulation and 4 node creation/connection tokens, we need at least 6 bits to define a token ($2^6 = 64$ tokens, which means that some tokens are encoded twice).

Each experiment involves the following steps:

1. Initialize the population with $N = 300$ agents with random bitstrings.
2. For each generation, build the graph of each agent and evaluate it in the environment.
3. Select the individuals with higher fitness (namely, 20 % of the population) and produce new ones by crossing over the parents. The system performs probabilistic mutations and insertions or deletions of codons on the bitstring of the offspring.
4. Iterate the process with the new generation.

In order to evaluate the individuals, they are put into the environment, starting on any blank cell in the grid, and they have to find the food within a limited amount of time (20 time steps). The agent can perform only one action per time step; when this action is incompatible with the environment (*e.g.* go towards a wall), it is simply discarded (the agent loses one time step). Its fitness for each run is: $F = D - K + B + 2 * R$ (F: fitness for the run; D: number of blank cells that have been discovered during the run; K: time steps spent on already known cells; B: bonus when the food is found (30 points); R: remaining time if the food has been found within the time limit ($R < 19$)). It was designed to advantage exploring agents (see D and K)

that reach quicker the food (thanks to the R coefficient, remaining time steps are more rewarding than the discovering of any more new cells). Since there is no reinforcement learning during the run, the fitness *has* to provide delayed information to measure the quality of the behavior. Each agent is evaluated 4 times starting on each empty cell, then its total fitness is the sum of the fitnesses computed for each run. In the optimal case, the fitness is 4500.

The experiments reported here were carried out on various initial genotype sizes, from 300 to 540 bits. The original population genotype sizes change during evolution. Each experiment has been bounded by 10,000 generations, which in most cases is sufficient to reach high enough fitness values.

4.3 Results

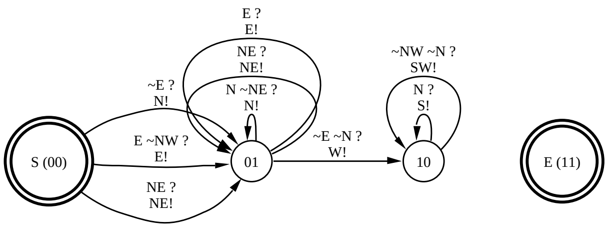

Figure 5: Graph of the best individual in a representative experiment

A representative example is reported on figure 6, which shows the best and average fitness values.

Table 1: A LCS-like representation of the graph on figure 5. EC: external conditions, IC: internal conditions, EA: external actions, IA: internal actions

EC								IC	EA	IA
E	NE	N	NW	W	SW	S	SE			
1	#	#	#	#	#	#	#	00	N	01
0	#	#	1	#	#	#	#	00	E	01
#	0	#	#	#	#	#	#	00	NE	01
#	1	0	#	#	#	#	#	01	N	##
#	0	#	#	#	#	#	#	01	NE	##
0	#	#	#	#	#	#	#	01	E	##
1	#	1	#	#	#	#	#	01	W	10
#	0	#	#	#	#	#	#	10	S	##
#	#	1	1	#	#	#	#	10	SW	##

Figure 5 presents a behavioral graph obtained by the best individual in a representative experiment. It has also been represented in a LCS-like formalism (table 1).

The agent whose graph is described in figure 5 has the following behavior: from any vertical corridor, it first reaches horizontal corridor, then the NE corner, and finally goes straight to the food. This is a nearly optimal solution. The graph presented in figure 5 shows that a nearly optimal behavior can be obtained. Especially, there are clear distinctions between the bottom of vertical corridors (`N ¬NE` identifies cells $S_{\{1,2\}\_n}$), the top of vertical corridors (`NE` $\rightarrow S_6, S_7, S_{3\_n}$), the horizontal corridor (`E` $\rightarrow S_8, S_{\{4,5\}\_n}$) and the crucial NE corner (S_9 is identified by `¬E ¬N ¬NW`).

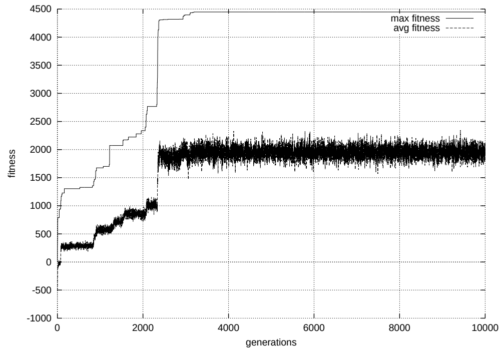

Figure 6: Best and average fitness obtained with 360-bit genotypes

5 Discussion

5.1 Readability and Minimality of Representation

One important advantage of ATNoSFERES with respect to XCSM is that the ATN resulting from the evolution is very easy to understand. But this feature is not only a question of graphical representation.

XCSM produces a constant size list of classifiers into which the size of the external conditions part and of the memory register must be chosen in advance. As a result, there are generally more classifiers and more internal states than necessary.

By contrast, ATNoSFERES builds a graph whose number of nodes, edges, and labels on the edges are not given in advance. Thus it can build a minimal controller to solve the given problem.

Another key difference is that, in XCSM, the sequence of internal states of the agent during one run is not explicitely stated and must be derived by hand through careful examination. On the contrary, this sequence is perfectly clear when one reads an ATN. Furthermore, the internal state is very stable in ATNoSFERES. But this advantage of ATNoSFERES has its counterpart that will be discussed in § 5.2: ATNoSFERES cannot represent `Condition-Action` rules that can be fired whatever the internal state is, as it is the case in XCSM with an internal condition composed of "#" only.

5.2 Generalization

An important difference between XCSM and ATNoSFERES formalisms call upon the elements on which generalization can take place. In the current implementation of ATNoSFERES, generalization is not possible with respect to the internal conditions and actions. This prevents ATNoSFERES from dealing with a default behavior, regardless of the internal state.

In XCSM, a `#` in the internal condition allows the classifier to be applied whatever the internal state represented by the memory register is. This mechanism permits to act regardless of the internal state.

Furthermore, in the current implementation of ATNoSFERES, there is no explicit selection pressure on the generality of the conditions on the labels, while the production of generalized classifiers is inherent to the LCS approach. Thus, we do not necessarily obtain general rules and the condition labels still contain redundant information, *e.g.* in the identification of the NE corner.

However, the conditions that are actually encountered in the graphs are quite general. In fact, once a good solution has been found, the population tends to become homogeneous and the size of genotypes stabilizes. Many different genotypes can lead to similar behaviors, but we assume that there is a bias towards compact solutions.

5.3 Reinforcement Learning and Classifier Selection

Another important difference between the ATN produced by ATNoSFERES and the list of classifiers produced by XCSM is that in the latter each classifier is endowed with a prediction representing its propensity to be fired, while in the former the edges get an equal probability to be selected if their condition token matches with the current situation.

Thus, in ATNoSFERES, if two edges can be selected simultaneously, the selection will not be deterministic. Since the optimal behavior is compatible with non-determinism only if both behaviors are strictly equivalent, the selection pressure in ATNoSFERES will prevent non-determinism in situations where it is detrimental. This provides a strong bias towards minimal controllers.

By contrast, in XCSM, several classifiers can match with the same situation, but only the strongest will be fired. Thus, it is not necessary that the other matching classifiers are deleted.

However, one important advantage of LCS with respect to ATNoSFERES is that the strength of classifiers are learned through a RL algorithm. Combining GA with RL is well known to help finding better individuals faster. In the Markov decision process (MDP) context, RL algorithms use more information about the experience of the agent than GA. While the GA only selects agents according to a global fitness function, RL algorithms distribute the reward obtained when the goal is reached only to the rules which have contributed to the behavior, taking into account the exact sequence of actions performed by the agent in the way the reward is back-propagated.

In order to remedy the fact that ATNoSFERES does not use RL, it has been necessary to include into the fitness function elements that carry some information about the actual behavior of the agent (see §4.2). But tuning such a fitness function is both difficult and crucial for the success of the experiment.

5.4 Optimality

The behaviors that have been obtained are still not completely optimal: when the agent starts from the west corridor, it should recognize the NW corner and then go directly in the third vertical corridor without checking the NE corner as it does. This is partly due to the fitness function we used: part of the time lost in exploring the NE corner is balanced by the exploration reward. Additionally, the structure for recognizing the NW corner would require at least two nodes and five edges and associated condition/action tokens. Thus it would constitute a major structural change in the graph with respect to the small selective advantage.

6 Conclusion and Future Work

From the perspective adopted in this paper, ATNoSFERES is similar to a *Pittsburgh* style LCS endowed with the ability to tackle non-Markov problems. By contrast with Michigan style LCS like XCSM, ATNoSFERES is deprived from any RL mechanism. We have shown that ATNoSFERES can produce controllers that are both very efficient in terms of the behavior they generate and very parsimonious in the way they specify that behavior. Thus we believe that ATNoSFERES is a good starting point to address more complex non-Markov problems than the benchmark experiment studied here.

The comparison with XCSM suggests two points in our agenda of research. First, it seems useful to investigate the possibility of adding a parameter equivalent to the classifier force, so as to combine RL with the GA already in use.

Second, it seems necessary to address the sub-optimality problem highlighted in §5.4. It seems that finding an optimal individual in the `Maze10` environment from the one presented in figure 5 requires a very expensive structural modification. As a result, it is unlikely that the GA will find this modification without further improvements in the representation or the mechanisms. In that respect, the ability of classifiers to deal with unspecified internal states seems a key advantage, and we should try to find a way to give that property to ATNoSFERES. Though this feature has not been implemented at this time in the model, it would only consist in copying the same edge on each existing node, by adding one special connection token to the genetic code.

References

[1] K. A. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, Dept. of Computer and Communication Sciences, University of Michigan, 1975.

[2] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution.* John Wiley & Sons, 1966.

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

[4] F. Gruau. *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm.* Ph.D. thesis, ENS Lyon – Université Lyon I, 1994.

[5] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* University of Michigan Press, Ann Arbor, MI, 1975.

[6] J. Kodjabachian and J.-A. Meyer. Evolution and Development of Neural Controllers for Locomotion, Gradient-Following, and Obstacle-Avoidance in Artificial Insects. *IEEE Transactions on Neural Networks*, 9:796–812, 1998.

[7] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, 1992.

[8] J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors. *Genetic Programming 1996: Proceedings of the First Annual Conference*, Stanford University, CA, 1996. MIT Press.

[9] S. Landau, S. Picault, and A. Drogoul. ATNoSFERES: a Model for Evolutive Agent Behaviors. In *Proceedings of the AISB'01 Symposium on Adaptive Agents and Multi-Agent Systems*, 2001.

[10] P. L. Lanzi. An Analysis of the Memory Mechanism of XCSM. In *Proceedings of the Third Genetic Programming Conference*, 1998.

[11] P. L. Lanzi and S. W. Wilson. Toward optimal classifier system performance in non-markov environments. *Evolutionary Computation*, 8(4):393–418, 2000.

[12] A. Lindenmayer. Mathematical Models for Cellular Interaction in Development, parts I and II. *Journal of theoretical biology*, 18, 1968.

[13] S. Luke and L. Spector. Evolving Graphs and Networks with Edge Encoding: Preliminary Report. In Koza et al. (8), pages 117–124.

[14] D. J. Montana. Strongly Typed Genetic Programming. In *Evolutionary Computation*, volume 3. 1995.

[15] S. Picault and S. Landau. Ethogenetics and the Evolutionary Design of Agent Behaviors. In N. Callaos, S. Esquivel, and J. Burge, editors, *Proceedings of the 5th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI'01)*, volume III, pages 528–533, 2001.

[16] H.-P. Schwefel. *Evolution and Optimum Seeking.* John Wiley and Sons, Inc., 1995.

[17] R. S. Sutton and A. G. Barto. *Reinforcement Learning, an introduction.* MIT Press, Cambridge, MA, 1998.

[18] A. Tomlinson and L. Bull. CXCS. In P. Lanzi, W. Stolzmann, and S. Wilson, editors, *Learning Classifier Systems: from Foundations to Applications*, pages 194–208. Springer Verlag, Heidelberg, 2000.

[19] S. W. Wilson. ZCS, a Zeroth level Classifier System. *Evolutionary Computation*, 2(1):1–18, 1994.

[20] S. W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

[21] W. A. Woods. Transition Networks Grammars for Natural Language Analysis. *Communications of the Association for the Computational Machinery*, 13(10):591–606, 1970.

[22] X. Yao. Evolving Artificial Neural Networks. *Proceedings of the IEEE*, 87, 1999.