Further Comparison between ATNoSFERES and XCSM

Samuel Landau[1], Sébastien Picault[2], Olivier Sigaud[1], and Pierre Gérard[1]

[1] Laboratoire d'Informatique de Paris 6
8, rue du Capitaine Scott
75 015 Paris France
{Samuel.Landau,Olivier.Sigaud,Pierre.Gerard}@lip6.fr
http://miriad.lip6.fr/~landau
http://animatlab.lip6.fr/~{sigaud,pgerard}
[2] Laboratoire d'Informatique Fondamentale de Lille
Cité Scientifique
59 655 Villeneuve d'Ascq Cedex, France
Sebastien.Picault@lifl.fr
http://www.lifl.fr/~picault

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 aliazing. In the context of our ongoing line of research, we compare it with XCSM, a memory-based extension of the most studied Learning Classifier System, XCS, through two benchmark experiments. We focus in particular on internal state generalization, and add special purpose features to ATNoSFERES to fulfill that comparison. We then discuss the role played by internal state generalization in the experiments studied.

Keywords Evolutionary Algorithms, Learning Classifier Systems, perceptual aliazing, internal state generalization, ATN[1]

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 of the environment. 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. As in the general *Reinforcement Learning* (RL) framework [20], the goal of a LCS-based agent is to maximize the scalar rewards it receives from its environment. The policy is defined by a set of rules – or classifiers – specifying the action to choose according to some *conditions* concerning the perceived situations.

[1] ATN stands for "Augmented Transition Networks"

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 a *perceptual aliazing* problem. In such a case, 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 ([21, 14] for instance).

Within this framework, explicit internal states were added to the classical (condition, action) pair of the classifiers, e.g. in XCSM [14, 23]. These internal states provide the additional information required to choose an action when the problem is non-Markov. The problem of properly setting the classifiers is generally devoted to *Genetic Algorithms* (GA).

In this paper, we extend our first comparison presented in [12] between XCSM and "ATNoSFERES". The latter is 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. We show in [12] that such an evolutionary approach is able to cope with non-Markov environments; in ATNoSFERES, the goals are represented by a *fitness* measure (instead of classical LCS learning techniques).

In the first section, we present the features and properties of the ATNoSFERES model. 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 GA, with additional operators. Then we show that the graph-based representation is formally very close to LCS representations, and, in particular, to XCSM (§ 3.1). We remind the results of our previous experiments presented in [12] (§4), and the comparison we made (§5), that led us to assume that the lack of internal state generalization in ATNoSFERES explained the sub-optimality of the solutions that were found. A comparison of the performance of ATNoSFERES with and without a way to represent internal state generalization (§ 6) allows us to discuss in detail the validity of this assumption (§7). In the conclusion, we present further additions that could be made to our model so as to reach an even higher performance (§8).

2 Description of the ATNoSFERES model

2.1 Graph-based expression of behaviors

The architecture provided by the ATNoSFERES model [11, 18] involves an ATN graph [24] 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.

Like LCSs, ATNoSFERES binds conditions expressed as a set of attributes to actions, and is endowed with the ability to generalize conditions by ignoring some attributes. But in ATNoSFERES , the conditions and actions are used in a graph structure that provides internal states. Such graphs have already been used by [11] for describing the behavior of agents. The labels on edges consist in 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 at 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. An edge is randomly chosen in this set. If the set is empty, then an action is chosen randomly over all possible actions, the current node remains unchanged, and we do not perform the next two steps.
  3. The actions on the label of the current edge are sequentially performed by the system. Assuming that only one action can be performed by time step, only the last action is actually performed. 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 at the *End* node (E). This node is a general feature of our model and may never be reached. This appears to be the case in all the following experiments (since agents reaching the *End* node stop moving and thus have a very low fitness, see § 4).

Having described how the graphs are used, we now present how they are built.

2.2 The graph-building process

The graph describing the behaviors is built from a genotype 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 circuits [8], finite-state machines [2], neural networks [25] 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 Evolutionary Strategies [19]) and, on the other hand, approaches that use a single structure both as the genotype and the phenotype (such as Genetic Programming [7, 17], Evolutionary Programming [2], L-systems [15], developmental program trees, e.g. [6, 4, 16]).

In the ATNoSFERES model [10], 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. 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.

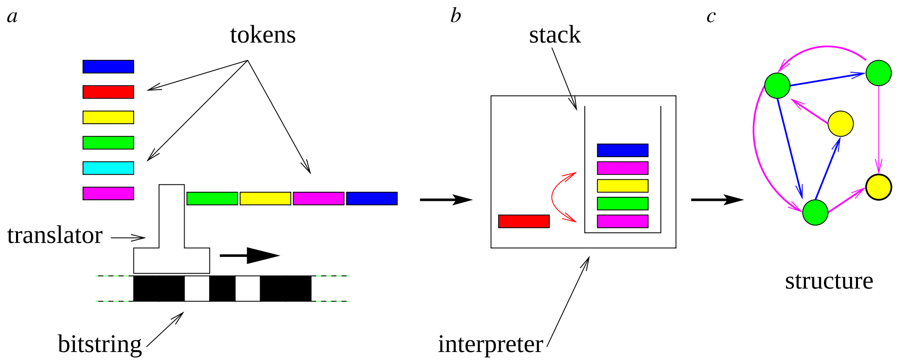


Fig. 1. Principles of the genetic expression we use to produce the behavioral graph from the bitstring genotype. The string is first decoded into tokens (a), which are interpreted in a second step as instructions (b) to create nodes, edges, and labels (c).

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 $\mathcal{G} : \{0,1\}^n \longrightarrow \mathcal{T} \quad (|\mathcal{T}| \leq 2^n)$ where $\mathcal{T}$ is the set 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".

Interpretation Tokens are instructions of the ATNoSFERES graph-building language (see table 1). They are interpreted one by one, while the interpreter is fed with the token stream produced by the translator. The interpretation of each successive token operates on a stack in which parts of the future graph are stored. The construction of the graph takes place during this interpretation process, by creating nodes and connections, and connecting them to the initial Start and End nodes. As in other stack-based languages (*e.g.* Forth, PostScript), the data in the stack can also be directly accessed by some instructions (*e.g.* connect, dup: see table 1), by other means that only push/pop operations.

In order to cope with a "blind" evolutionary process (i.e. based on random mutations on a fine-grain genotype), the graph built by the tokens sequence has to be robust to mutations [18]. For instance, the replacement of a token by another, or its deletion, should only have a *local impact*, rather than transforming the whole graph.

Therefore, if an instruction cannot be executed successfully, it is simply ignored, and when all tokens have been interpreted, the graph is made consistent, *e.g.* by linking Start to nodes without input edges (other than self-connected), or nodes without output edges to End.

Since any sequence of tokens is meaningful, the graph-building language is highly robust to any variations affecting the genotype, thus there is no specific syntactical or semantical constraint on the genetic operators. In addition, the sequence of tokens is to some extent order-independent and a given graph can be produced from very different genotypes.

token	resulting actions
stack tokens	manipulate the stack
nop	no action, the token is just discarded
swap	swap the two first tokens
dup	push a copy of the first action or condition token
del	delete the first action or condition token
dupNode	push a copy of the first node token
delNode	delete the first node token[a]
popRoll	pop the token, and puts it on the bottom of the stack
pushRoll	take the token from the bottom of the stack, and push it
structure tokens	create nodes and connect them with edges
node	create a new node and push it
connect	create an edge from the first to the second node token[b], label the edge with the set of conditions token and the list of actions token until the second node, delete the action and condition tokens that were used
startConnect	create an edge from the Start node to the first node token, label the edge with the set of conditions token and the list of actions token until the node, delete the action and condition tokens that were used
endConnect	create an edge from the first node token to the End node, label the edge with the set of conditions token and the list of actions token until the node, delete the action and condition tokens that were used
agent tokens	actions and conditions tokens, specific to the agent
condition?	push the condition on the stack
`action!`	push the action on the stack

Table 1. The graph building language. Here "first" (node, action or condition) refers to the first (node, action or condition) token encountered while going down the stack.

[a] it may be a copy: possible other copies of the node still remain in the stack
[b] they could both be copies of the same node, so it would be a self-connected edge

The graph-building language Table 1 details the tokens that are used to build the graphs. There are three categories of token:

- stack tokens (swap, dup, ...), that manipulate the stack. They are independent from the agent abilities or the structure that is built.
- structure tokens (node, connect, ...), that perform atomic structure building steps. Here they are designed to build graphs. They are also independent from the agent abilities. Some of these tokens use tokens already in the stack.
- agent tokens (actions, conditions), that are specific to an agent, and describe its abilities. These token are just pushed onto the stack.

2.3 Integration into an evolutionary framework

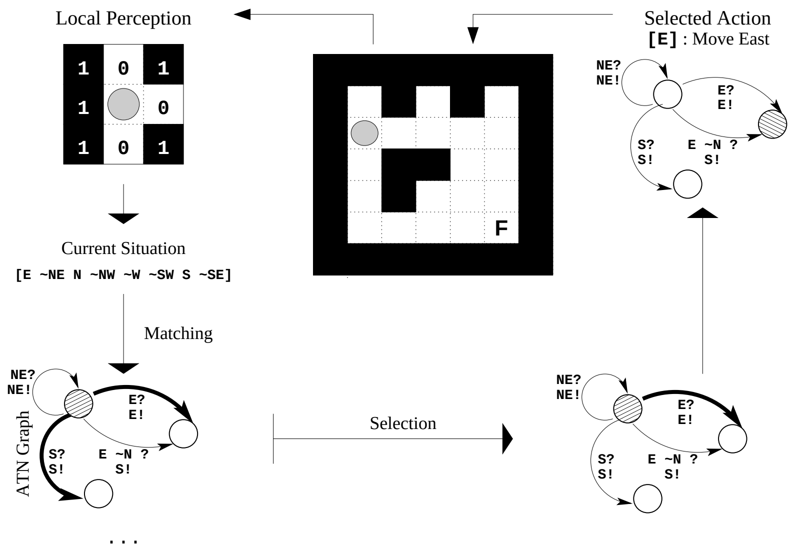


Fig. 2. In this example, the agent, located in a cell of the maze, perceives the presence/absence of blocks in each of the eight surrounding cells. It has to decide towards which of the eight adjacent cells it should move. From its current location, the agent perceives [E ¬NE N ¬NW ¬W ¬SW S ¬SE] (token 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.

In this paper, the ATNoSFERES model has been applied inside an evolutionary algorithm to produce controllers for agents.

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.

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 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

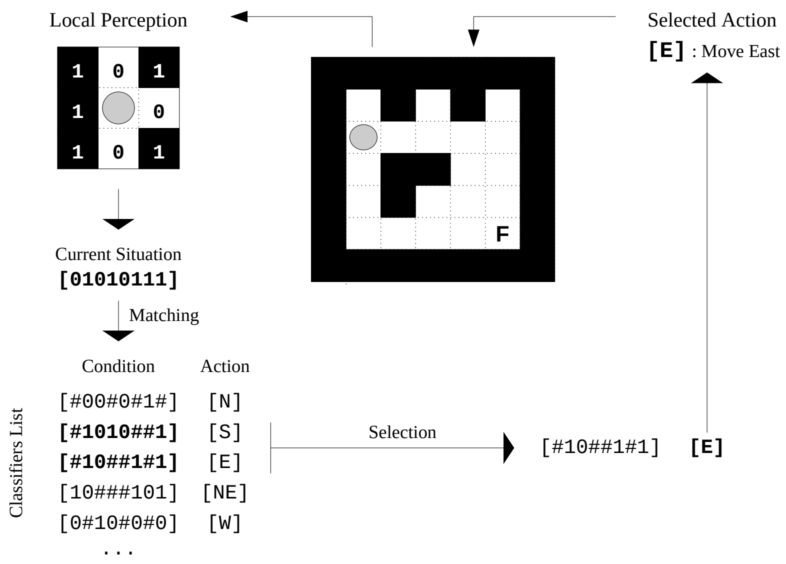


Fig. 3. The agent perceives the presence/absence (resp. 1/0) of blocks in each of the eight surrounding cells (considered clockwise, starting with the north cell). Thus from its current location, the agent perceives [01010111]. Within the list of classifiers characterizing it, the LCS first selects those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

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 build classifiers, which define the behavior of the system as shown in figure 3. Within the LCS framework, the use of "#" 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 attribute.

The main issue with generalization is to figure out on which conditions can the *don't care* symbols be used so that the actions keep accurate. 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, as 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 [20].

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 [22], Wilson found much more efficient to define the fitness according to the accuracy of the utility prediction. The resulting system, XCS [23], 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. [21] proposed a way to probalistically link classifiers in order to bridge aliazed situations. In contrast, Lanzi [14] 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 explicitly represent the internal state of the LCS. The memory register provides XCSM with more than just the environmental perceptions. Thus, dealing with perceptual aliazing is made possible by adding information from the past experience of the agent. As a result of this addition, a classifier contains four parts (see table 6): 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 the 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 "#" symbols. When a classifier is fired, a *don't change* symbol in the internal action results in not changing the corresponding bit in the memory register. Like XCS, XCSM draws benefits from generalization in the external condition, but also in the internal condition and the internal action.

As explained in more details in [12], an ATN such as those evolved by ATNoSFERES can be translated into a list of classifiers. The nodes of the ATN play the role of internal states in XCSM and make ATNoSFERES able to deal with perceptual aliazing. Thus it is natural to compare ATNoSFERES with XCSM. The edges of the ATN are characterized by several informations which can also be represented in classifiers: the source and destination nodes of the edge are respectively equivalent to the internal condition and the 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.

4 First experiments

4.1 The perceptual aliazing problem

In [12], our purpose was to compare the evolutionary use of ATNoSFERES with XCSM with respect to their ability to deal with non-Markov problems. In order to provide that comparison, we experimented our model in the `Maze10` environment, for which [14] provides empirical results obtained with XCSM.

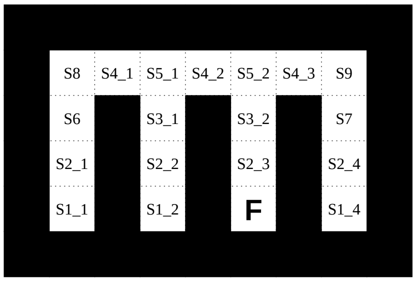


Fig. 4. The Maze10 environment. **F** represents the goal to reach (food). The agent starts from any cell of the maze; a few cells are unambiguous (S_i) but in the other ones the same perceptual situations may require either similar actions or different ones (*e.g.* go north in $S_{2\_\{1,2,4\}}$ but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

4.2 Experimental setup

We tried to reproduce an experimental setup close to that used in [14] 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 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 (with a 1% rate) and insertions or deletions of codons (with a 0.5% rate) 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 a blank cell in the grid, and they have to find the food within a limited amount of time. The agent cannot perceive the food, and it can perform only one action per time step; when this action is incompatible with the environment (*e.g.* go west when the west cell contains an obstacle), it is simply discarded (the agent loses one time step and stays on the same cell). 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; R: remaining time if the food has been found within the time limit). Thus, the selection pressure encourages short paths to food and exploration. At each time step, the current cell is added to the set of already-known cells (in order do compute D and K). The term $2*R$ ensures that a short path including an already-known cell is still preferred to a a longer path with only distinct cells. 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, with $B = 30$ and a 20 time steps limit, 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 is sufficient in most cases to reach high enough fitness values.

4.3 Results

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 (fig. 6).

The agent whose graph is described in figure 5 has the following behavior: from any vertical corridor, it first reaches the horizontal corridor, then the NE

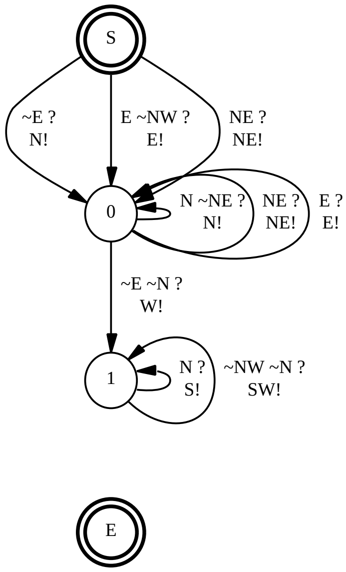


Fig. 5. Graph of the best individual in a representative experiment

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	##

Fig. 6. A LCS-like representation of the graph on figure 5. EC: external conditions, IC: internal conditions, EA: external actions, IA: internal actions .

corner, and finally goes straight to the food. This is a nearly optimal solution. 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).

5 Discussion of the first experiments

In [12], we presented a discussion resulting from the comparison between our model and XCSM. The main points we made were the following.

Minimality of Representation: While 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, ATNoSFERES builds a graph whose number of nodes, edges, and labels on the edges can be minimal to solve the given problem (agreed that we focus on the best agent only). Hence the graph built by ATNoSFERES can be minimal while XCSM model cannot.

Reinforcement Learning and Classifier Selection: One important advantage of LCS with respect to ATNoSFERES is that the forces of classifiers are learnt through a RL algorithm. In order to remedy the fact that ATNoSFERES does not use RL, it is necessary to include into the fitness function elements that carry some information about the actual behavior of the agent (see §4.2).

Readability: As we showed in [12], one important advantage of ATNoSFERES with respect to XCSM is that the ATN resulting from the evolution is very easy to understand. Another key difference is that, in XCSM, the sequence of internal states of the agent during one run is not explicitly 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 next: ATNoSFERES cannot easily 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.

Generalization: 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 action regardless of the internal state. On the contrary, the tokens that have been chosen in those first experiments (see tab. 1) prevent ATNoSFERES from dealing with a default behavior, since connection tokens create edges (i.e. rules) between two nodes (i.e. two internal states). We investigate this point further in §6.

Optimality: Results given in [12] showed that the behavior obtained on Maze10 with ATNoSFERES was not completely optimal, and that obtaining the optimal graph would require a major structural change in the graph with respect to the low selective advantage.

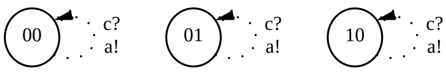
6 New experiments

6.1 Evaluating the need for state generalization

In [12], we concluded from the previous discussion by assuming that the ability of XCSM classifiers to deal with a default behavior, regardless of the internal state was a key advantage of XCSM over our model. Therefore we will now present our attempt to add an internal state generalization property to our model by extending the graph building language, with a new *defaultSelfConnect* token (see table 2).

token	resulting actions
...	
structure tokens	create nodes and connect them with edges
...	
defaultSelfConnect	creates an edge from all the already present nodes to themselves, labels the edges with the set of conditions token and the list of actions token until the first node, deletes the action and condition tokens that were used

Table 2. Extension to the graph building language (see table 1). Here "first" node refers to the first node token encountered while going down the stack



self transitions present on states:	IC	IA
00	00	##
00, 01	0#	##
00, 01, 10	##	##

Fig. 7. Different self-connection combinations and their translation in the XCSM formalism. IC: internal condition, IA: internal action

We emphasize that, since only the already created nodes will be self-connected, the level of state generalization depends on the time when this token gets interpreted. For example, if a *defaultSelfConnect* comes before all the internal nodes of the graph are created, then the next nodes will not be affected by this self-connecting instruction.

As shown in figure 7, self-connecting transitions on nodes is equivalent to the presence of # in internal condition and/or internal action parts in XCSM. More precisely, as the figure shows, if only the transition in node 00 is present, it is equivalent to a completely specified internal condition, while if the same transition is present in all nodes, it is equivalent to a completely unspecified internal condition and internal action.

6.2 12-Candlestick experimental setup

In our previous experiments on Maze10, the best fitness was about 98% of the maximum theoretic fitness. Since these results are very close to optimality, Maze10 experiments do not provide a large enough opportunity for improvement to clearly probe the efficiency of the new encoding. Therefore, we propose a new candlestick-like maze (see figure 8) where the advantage of the *defaultSelfConnect* enabled language should be more significant with respect to the one without that token.

The agent starts from the top cells of any of the vertical corridors. We consider those starting locations only, because we are focusing on the generalization abilities, rather than searching for a general behavior to solve that maze from any starting cell. While going south along the “candles” from the 12 top cells, thanks to an empty cell on the side, the agent can determine which direction to take afterwards once it reaches the bottom of the candles. Indeed there is no ambiguity for far-right and far-left candles.

The internal state management strategy we have envisioned in designing this experiment is the following. The agent just needs one bit of memory. This bit is set when the agent sees an empty space on its left hand side or on its right hand side, and represents whether it is in the left part or the right part of the candlestick. This informations suffices to choose the right direction when it reaches the bottom of the candles. Given this internal state management strategy, the necessity to generalize on the internal state values becomes clear when one considers all the $\{S_{a\_i}\}$ cells. In all those cells, which are represented by the same perceptual conditions, the agent must go south, *i.e.* choose the same action, *whether it is in the left part or the right part of the candlestick*. In the

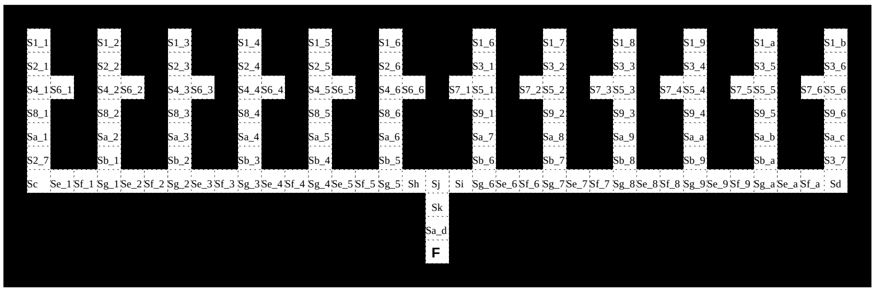


Fig. 8. The 12-Candlestick environment. **F** represents the goal to reach (food) from the 12 higher squares; 5 cells are unambiguous (S_i). In the other ones the same perceptual situations may require either similar actions or different ones (*e.g.* go west in $S_{g\_\{1,2,3,4,5\}}$ but go east in $S_{g\_\{6,7,8,9,a\}}$)

formalism of ATNoSFERES, this means that it must follow the same transition whatever the internal state is.

As a result of this property of the 12-Candlestick maze, only the agents following the internal state management strategy presented above can obtain an optimal performance. Any other strategy implies that the agents make additional steps in order to choose the correct directions. Indeed, disambiguating by visiting the far-left or far-right corners like the strategy obtained in Maze10 would imply a too costly detour with respect to the optimal path. But, as it will become clear in the remainder of that paper, even if this optimal behavior can be eventually obtained without using the *defaultSelfConnect* token, it is nevertheless achieved more often with it.

The experimental setup is similar to the one in our previous Maze10 experiments, except for the genetic encoding and the fact that the agent does not start from all the empty cells. Like before, we need at least 6 bits to define a token, but this time in order to observe the influence of our special token, we use two genetic encodings. In the first encoding, we use the *nop* token, while in the second, it is replaced by the *defaultSelfConnect* token.

The agent is evaluated 6 times starting in each of the 12 top blank cells. The fitness function is the one of Maze10 experiments. In the optimal case, with $B = 50$ and a 30 time steps limit, the fitness is 6732. As for the Maze10 experiments, we made the experiments for 5 different initial genotypes length (300, 360, 420, 480 and 540 bits). During the experiments, the genetic operators might change the lengths of the genotypes.

6.3 12-Candlestick experiment results

Figures 9 and 11 respectively show the best solutions found with and without the *defaultSelfConnect* token. The fitness of both these individuals is 6726 for a maximum of 6732. The missing points are lost when the agent passes through the corner at the foot of the far-right candle (see figure 8, cell S_d), instead of going directly from S_{3_7} to S_{f_a}. Since there are 6 evaluation per "candle", it loses 6 times 1 point.

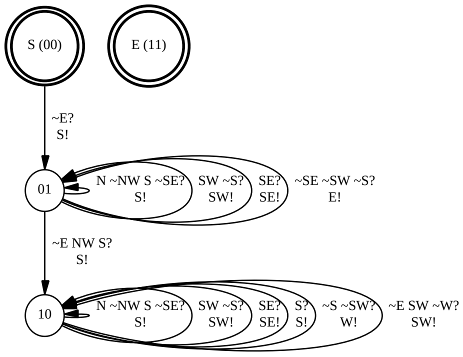


Fig. 9. 12-Candlestick. Graph of the best individual using the *defaultSelfConnect* token. The token appears 3 times, after the 2 nodes were created, so there are 3 identical self-connecting edges for both nodes.

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

Fig. 10. A LCS-like representation of the graph on figure 9. The three last classifiers correspond to self-connecting edges in the graph. EC, IC, EA, IA: see figure 6.

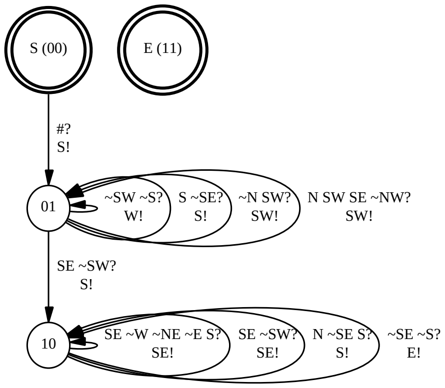


Fig. 11. 12-Candlestick. Graph of the best individual *not* using the *defaultSelfConnect* token.

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

Fig. 12. A LCS-like representation of the graph on figure 11. EC, IC, EA, IA: see figure 6.

6.4 Discussion of 12-Candlestick results

As figure 13 shows, the average fitness obtained with the *defaultSelfConnect* token in the 12-Candlestick experiment is significantly better than the one obtained without that token. The difference between the performances in both

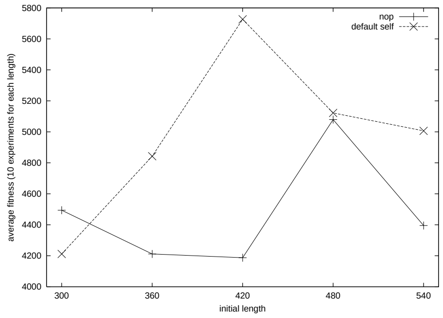


Fig. 13. 12-Candlestick. Average fitness with and without self token

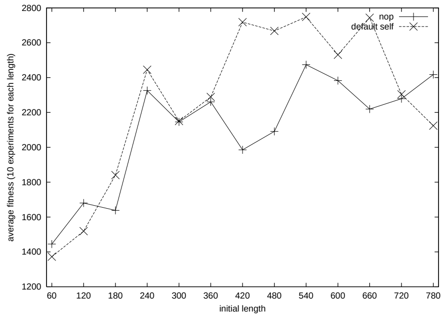


Fig. 14. Maze10. Average fitness with and without self token

cases according to the different initial lengths of the bitstrings is given on figure 15 and 16. We can clearly see from these figures that the *defaultSelfConnect* token conveys a selective advantage to our agents.

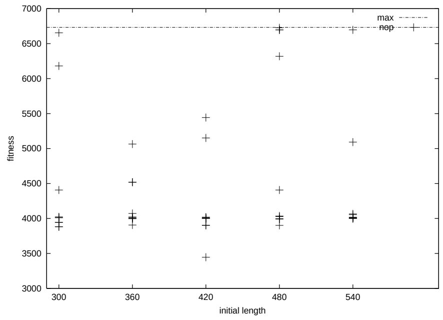


Fig. 15. 12-Candlestick. Fitness without *defaultSelfConnect* token

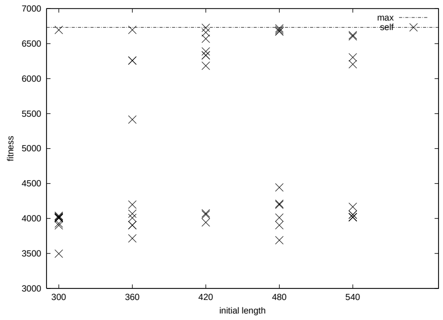


Fig. 16. 12-Candlestick. Fitness with *defaultSelfConnect* token

The purpose of making these experiments on the 12-Candlestick maze is to check that the *defaultSelfConnect* token is working well and provides the property for which it was designed. But, since the 12-Candlestick maze is specially designed to favor the use of the *defaultSelfConnect* token, it is also necessary to check whether or not this property can be generalized to Maze10, even if we expect less significant results.

6.5 Second Maze10 experimental setup

The experimental setup is that of our previous Maze10 experiments (see section 4.2), except for the genetic encoding. Like in the previous experiments, in order to observe the influence of our *defaultSelfConnect* token, both genetic encoding tested differed only by one codon: *nop* for the first, and *defaultSelfConnect* for the second, as we did in section 6.2

6.6 Second Maze10 experiment results

The best solutions, respectively with and without the *defaultSelfConnect* token, are shown in figures 17 and 19. The best fitness for both genetic encodings (with and without the *defaultSelfConnect* token) are close to each other, respectively 4448 and 4436 for a maximum of 4500. This best fitness was found in 3 experiments for the first, and only one time for the second, over the 50 experiments run for each genetic encoding. As in our previous Maze10 experiments, most of the points are lost when reaching the NE corner when coming from the west of the maze, and other points are also lost when the agent goes in or out of some columns by passing through the cell on top of it, instead of using a diagonal move.

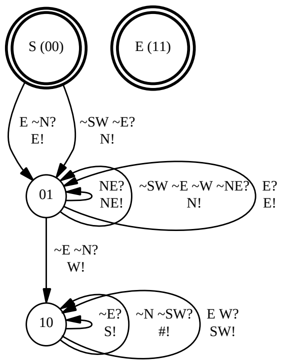


Fig. 17. Maze10. Graph of the best individual using the *defaultSelfConnect* token for Maze 10. The token appears 1 time, and is applied on one node only.

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

Fig. 18. A LCS-like representation of the graph on figure 17. EC, IC, EA, IA: see figure 6.

7 Further discussion

From figures 13 and 14, it is clear that ATNoSFERES obtains better performance both on 12-Candlestick and on Maze10 with the *defaultSelfConnect* token than without it.

The optimal behavior is never reached, however (the best solution is 0.1% below this optimum). The best individual on the 12-Candlestick, presented on figure 9, uses the *defaultSelfConnect* token as expected, so as to represent a completely unspecified internal state. This result seems to support our initial assumption according to which being able to generalize on the internal state is important to solve such behavioral problems, and results on Maze10 seem to confirm the generality of the assumption.

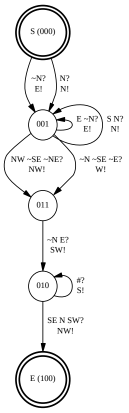


Fig. 19. Maze10. Graph of the best individual *not* using the *defaultSelfConnect* token. The edge between 010 and 100 is actually never crossed.

EC								IC	EA	IA
E	NE	N	NW	W	SW	S	SE			
#	#	0	#	#	#	#	#	000	E	001
#	#	1	#	#	#	#	#	000	N	001
1	#	0	#	#	#	#	#	001	E	###
#	#	1	#	#	#	1	#	001	N	###
#	0	#	1	#	#	#	0	001	NW	011
0	#	0	#	#	#	#	0	001	W	011
1	#	0	#	#	#	#	#	011	SW	010
#	#	#	#	#	#	#	#	010	S	###
#	#	1	#	#	1	#	1	010	NW	100

Fig. 20. A LCS-like representation of the graph on figure 19.EC, IC, EA, IA: see figure 6.

But a closer examination of the best individuals obtained through other experiments reveals that our assumption must be refined. In particular, the best individual on Maze10, presented in figure 17, uses the *defaultSelfConnect* token when there is only one node in the stack. More generally, it appears that the *defaultSelfConnect* token has been used many times to add only one self-connecting transition in the graph, or none at all (if no node is present when *defaultSelfConnect* is interpreted).

As explained in figure 7, since the interpretation of that token results in the addition of self-connecting transitions only to the nodes already present in the stack, it does not always result in the equivalent of full generalization on the internal state. In particular, when there is only one self-connecting transition in the graph, the interpretation of the *defaultSelfConnect* token results in the equivalent of a fully specified internal condition part followed by an unspecified internal action part.

Interestingly, however ATNoSFERES can obtain one self-connected node without using the *defaultSelfConnect* token. But doing so requires that much more constraints on the bitstring are fulfilled. Thus a self-connected node is much less likely to happen without the *defaultSelfConnect* token.

So it may be that ATNoSFERES gets a better performance with the *defaultSelfConnect* token than without it just because having self-connecting transitions is a beneficial property and having that token significantly increases the probability to have that property.

Hence, what this more detailled study seems to reveal is that, when several internal states are necessary to solve a non-Markov problem, it is important that the system keep the possibility to specify (condition, action) transitions without being compelled to change its internal state.

As a result of these new findings, we still cannot definitively conclude yet on whether it is having the ability to generalize on the internal state or having the ability to represent stable internal state that is the most beneficial property in the problems studied in this paper.

8 Conclusion and Future Work

In the context of a comparison between XCSM and ATNoSFERES, we have studied in this paper the importance of the ability to represent generalized internal states. In order to do so, we have introduced a new *defaultSelfConnect* token which adds a self-connecting transition to all the nodes already present in the stack. We have also presented a new maze experiment specially designed to advantage systems able to generalize on the internal state.

Our experiments have shown that the performance of our system is significantly better with this addition. But, while this result seems to support the conclusion that being able to generalize on the internal state is a significant property of adaptive algorithms, a closer examination of what really happened during the experiments reveals that our *defaultSelfConnect* token has also been used for a different purpose than just generalizing on the internal state. That token seems to have another interesting property than the one for which it was designed.

In order to be able to conclude more accurately on the relative role played by the internal state generalization property and the stable internal state property, more experiments will be necessary. We believe that going into an even more detailled comparison between XCSM and ATNoSFERES on the two experiments presented above will help identifying further what is really necessary to reach an optimal behavior. In particular, we should try to assess the distinctive roles of generalization on the internal condition and on the internal action parts, by adding one or two specialized tokens representing each property independently of the other.

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 (1998), pp. 796–812.
[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, F. H. Bennett III, D. Andre, and M. A. Keane, *Automated Design of Both the Topology and Sizing of Analog Electrical Circuits Using Genetic Programming*, in Artificial Intelligence in Design'96, J. S. Gero and F. Sudweeks, eds., 1996, pp. 151–170.
[9] J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, eds., *Genetic Programming 1996: Proceedings of the First Annual Conference*, Stanford University, CA, 1996, MIT Press.
[10] S. Landau and S. Picault, *Stack-Based Gene Expression*, Technical Report LIP6 2002/011, LIP6, Paris, 2002.
[11] 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.
[12] S. Landau, S. Picault, O. Sigaud, and P. Gérard, *A Comparison between ATNoSFERES and XCSM*, in Langdon et al. [13], pp. 926–933.
[13] W. Langdon, E. Cantu-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. Schultz, J. F. Miller, E. Burke, and N. Jonoska, eds., *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, New York, july 9-13 2002, Morgan Kaufmann.
[14] P. L. Lanzi, *An Analysis of the Memory Mechanism of XCSM*, in Proceedings of the Third Genetic Programming Conference, 1998.
[15] A. Lindenmayer, *Mathematical Models for Cellular Interaction in Development, parts I and II*, Journal of theoretical biology, 18 (1968).
[16] S. Luke and L. Spector, *Evolving Graphs and Networks with Edge Encoding: Preliminary Report*, in Koza [9], pp. 117–124.
[17] D. J. Montana, *Strongly Typed Genetic Programming*, in Evolutionary Computation, vol. 3, 1995.
[18] S. Picault and S. Landau, *Ethogenetics and the Evolutionary Design of Agent Behaviors*, in Proceedings of the 5th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI'01), N. Callaos, S. Esquivel, and J. Burge, eds., vol. iii, 2001, pp. 528–533.
[19] H.-P. Schwefel, *Evolution and Optimum Seeking*, John Wiley and Sons, Inc., 1995.
[20] R. S. Sutton and A. G. Barto, *Reinforcement Learning, an introduction*, MIT Press, Cambridge, MA, 1998.
[21] A. Tomlinson and L. Bull, *CXCS*, in Learning Classifier Systems: from Foundations to Applications, P. Lanzi, W. Stolzmann, and S. Wilson, eds., Springer Verlag, Heidelberg, 2000, pp. 194–208.
[22] S. W. Wilson, *ZCS, a Zeroth level Classifier System*, Evolutionary Computation, 2 (1994), pp. 1–18.
[23] S. W. Wilson, *Classifier Fitness Based on Accuracy*, Evolutionary Computation, 3 (1995), pp. 149–175.
[24] W. A. Woods, *Transition Networks Grammars for Natural Language Analysis*, Communications of the Association for the Computational Machinery, 13 (1970), pp. 591–606.
[25] X. Yao, *Evolving Artificial Neural Networks*, Proceedings of the IEEE, 87 (1999).