Appeared in Information Sciences, 178 (2008), pp. 4482-4500

A Comparison between ATNoSFERES and Learning Classifier Systems on non-Markov problems

Samuel Landau,Olivier Sigaud

Institut des Systèmes Intelligents et de Robotique, CNRS FRE 2507
Université Pierre et Marie Curie - Paris 6
4 place Jussieu
F-75 252 Paris Cedex 05
France

Abstract

The automated design of the controller of software agents embedded in an environment is an important class of problems addressed in information sciences. In that class of problems, the case where agents face perceptual aliasing problems is particularly difficult. Within the evolutionary and adaptive approaches to controller design, there are several families of systems capable of dealing with such problems. This paper is devoted to a comparison of the way perceptual aliasing problems are solved by one family, namely Learning Classifier Systems, compared to our own model, ATNoSFERES. We present this model based on an indirect encoding Genetic Algorithm which builds Augmented Transition Network controllers, and we compare it with different Learning Classifier Systems, namely XCSM (a memory-based extension of the most studied Learning Classifier System, XCS) and ACS (an Anticipatory Learning Classifier System). To solve perceptual aliasing problems, the first uses an explicit internal state management mechanism while the second uses a rule-chaining mechanism. To carry out our comparison, we apply these systems to three different benchmark experiments. Our results raise a discussion of the respective properties of the mechanisms used by XCSM and ACS. Then we show that ATNoSFERES is endowed with enough expressive power to represent the mechanisms used by the other two systems to deal with perceptual aliasing problems. We conclude that ATNoSFERES provides a powerful framework to deal with such problems.

Key words: Evolutionary Algorithms, Perceptual Aliasing, Augmented Transition Networks, Learning Classifier Systems

1 Introduction

The software design of decision making systems is a very general problem in information sciences. Roughly speaking, this problem can be split into two categories. First, there are single-step problems, where an agent must make one classification decision from some input. Data mining problems provide good examples of this category. Second, there are multi-step problems, where an agent must make a sequence of decisions in order to reach any arbitrary goal in a given environment. In this paper, we will focus on the second category.

Designing decision making systems by hand is often difficult, whether they belong to the first category or to the second. So far, a lot of work has been devoted to engineering methods, architectures and theoretical frameworks whose purpose is to help human experts deal with these problems by hand. But in recent years, more and more work is focusing on automated techniques able to tune efficient decision making systems while requiring less design effort from human experts.

Evolutionary computing approaches are promising automated design techniques, because they can be applied to a very wide range of application domains with very few feedback from the applications and without any prior domain-dependent knowledge on the problem. In this family of approaches, Learning Classifier Systems (LCSs) [18] are a family of systems based on evolutionary and learning techniques where the decision making units are expressed in the form of rules called "classifiers". This formalism makes them very appealing because the rules produced are generally easy to read, interpret, and eventually modify.

While more and more LCSs are used to solve data mining problems, a lot of LCSs are also used to solve multi-step problems, *e.g.* control agents involved in a sensorimotor loop with their environment. Such agents perceive situations through their sensors as vectors of several attributes, each attribute representing a perceived feature of the environment. As pointed out by [30], LCSs are adaptive architectures based on *Reinforcement Learning* (RL) techniques [55], but endowed with generalization capabilities. Thanks to an LCS, an agent can *learn* the optimal policy – *i.e.* which action to perform in every situation, in order to maximize a reward obtained from the environment. The policy is defined by a set of rules – or classifiers – specifying an action according to some *conditions* concerning the perceived situation.

Email addresses: `Samuel.Landau@free.fr` (Samuel Landau), `Olivier.Sigaud@lip6.fr` (Olivier Sigaud).

URLs: `http://www-poleia.lip6.fr/~landau` (Samuel Landau), `http://www.isir.fr` (Olivier Sigaud).

Standard RL algorithms perform well in situations where the state of the agent-environment interaction is always known without ambiguity. But in real-world environments, it often happens that agents perceive the same situation in several different states, eventually requiring different optimal actions, giving rise to the so called "*perceptual aliasing*" problem. In such a case, the environment is *non-Markov*, and in general agents cannot perform optimally if their decision at a given time step only depends on their perceptions at the same time step.

There are several attempts to apply LCSs to non-Markov problems, relying on different approaches to the problem. In this paper, we will focus on two approaches: an explicit internal state management mechanism, implemented in XCSM and XCSMH [29], and a classifier-chaining mechanism implemented in ACS [50]. So far, there has been no direct comparison between the two kinds of approaches. One of the side-effects of this paper is to provide an indirect comparison through a comparison to our own system, ATNoSFERES.

In previous papers [24,26–28], we have presented a new framework, ATNoSFERES, also used to automatically design the behavior of agents and able to cope with non-Markov problems. ATNoSFERES relies on a pure evolutionary approach instead of the combination of evolutionary and learning techniques used in classical LCSs, but the resulting graph-based representation is semantically very similar to the LCS representation, raising the opportunity of a detailed comparison between the two classes of systems. In this paper, we will show that our approach to the perceptual aliasing problem in ATNoSFERES is endowed with enough expressive power to represent explicit internal state management mechanisms as well as classifier-chaining mechanisms. Thus we will use ATNoSFERES as a tool to compare the mechanisms in a unified framework and claim that our system can express more efficient solutions than the two others taken separately. But we will also mention that compared to the LCSs studied here, ATNoSFERES takes more time to find the solutions.

In the next section, we present the features and properties of the ATNoSFERES model in the general context of evolutionary techniques. Then we emphasize the formal similarity between the representations in ATNoSFERES and in LCSs, and we present in more detail the different approaches used in LCSs to cope with non-Markov problems.

Next, we provide new experimental comparisons differing from the previously published ones in several respect: we make comparisons with new systems and the encoding language is smaller than in previous papers, resulting in a more efficient exploration. First, we compare ATNoSFERES with XCSM and XCSMH on the **Maze10** problem, that was used as a benchmark by [29] to assess the performance of XCSM and XCSMH. Then we compare ATNoSFERES with ACS, relying on a study [38] on two distinct environments, **E1** and **E2**.

We claim that the study presented in this paper gives a first approach to a more accurate understanding of some relevant properties of different classes of non-Markov problems.

From the results, we discuss the respective properties of explicit internal state management mechanisms and classifier-chaining mechanisms on different problems. Since ATNoSFERES has the power to express the two mechanisms, we claim that it is able to choose the most suitable solution depending on the characteristics of the problem. But we also have to discuss the fact that ATNoSFERES converges slower than the LCSs studied here on the benchmarks tested in this paper. Thus we conclude that we should include on-line learning mechanisms in our model, which is our goal for immediate future work.

2 Description of the ATNoSFERES model

2.1 The graph-building process

The control architecture provided by the ATNoSFERES model [24,42] involves an ATN graph [63] which is basically an oriented, labeled graph with an initial node (labelled Start) and a final node (labelled End). Nodes represent states and edges represent transitions of an automaton. 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 [22], finite-state machines [9], neural networks [65] or program trees [20]. Very roughly, we can sketch three categories.

The first one uses the genotype as an encoding of a set of parameters for the structure (using an evolutionary algorithm like Genetic Algorithm [16,7,12] or Evolutionary Strategies [46]), which can give rise to fine-grain evolution. The problem with this approach is that the shape of the structures evolved is directly limited by the number of parameters through which the algorithm searches.

The second one identifies the genotype with the structure evolved (*e.g.* Genetic Programming [20,40] and Evolutionary Programming [9]). It can evolve any structure of any shape, but because of the hierarchies and the dependencies between parts of the structure, the genetic operators must be carefully designed in order to produce correct offspring structures, and this involves strong biases in the way they explore the search-space. For example, when one uses Genetic Programming, one must set a limit to the depth of the pro-

duced program trees. The consequences of these biases are difficult to evaluate, and much theoretical work is actually involved in studying this problem [43].

The third one uses a genotype relying on a language to build the structure (*e.g.* developmental program tree for neural networks [19,14,36]). This last approach combines the advantages of giving rise to fine-grain evolution of the structure while imposing no limitation on the structure evolved, provided that the genetic operators respect the syntactic rules of the language when producing the offspring.

Most of the languages used to build the structures are either rewriting rules inspired from $\mathcal{L}$-systems [35], or context-free grammars [4,5] for the given structure. On the one hand, rewriting rules might be hard to control. For instance, the structures evolved might grow exponentially, unless limited. On the other hand, context-free grammars impose strict hierarchical constraints on the syntax of the genotypes. When designing the genetic operators, these constraints are easy to deal with thanks to the grammar. But during evolution, these strict hierarchies prevent gradual evolution. As a matter of fact, if a change occurs in a part of the genotype that is governed by a grammatical rule close to the root of the context-free grammar, the constraints on grammatical correctness will imply other major changes in the genotype.

In the ATNoSFERES model, we use a stack-based language [25]. Its grammar is contextual and does not suffer from hierarchical string dependencies. Therefore we do not face the problem presented above when designing the genetic operators. Like some particular context-free grammatical approaches (*e.g. Grammatical Evolution* [45]) that are nonetheless specialized on program tree evolution, the interpreter accepts *any* genotype, and *always* produces a correct structure. Furthermore, we do not need to put any limit (especially on the genotype length), because, as we have reported in previous papers [24], some implicit regulation mechanisms seem to operate.

As a consequence of our choices, the graph building process operates in two steps (see figure 1):

(1) **Translation:** The bitstring (genotype) is translated into a sequence of tokens. Translation relies on a *genetic code*, *i.e.* a function

$$\mathcal{G} : \{0,1\}^n \longrightarrow \mathcal{T} \quad (|\mathcal{T}| \leq 2^n) \tag{1}$$

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

(2) **Interpretation:** The token interpreter is fed with the token stream produced by the translator. The tokens are interpreted one by one as instruc-

tions of a robust programming language, dedicated to graph building. 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 between nodes. As in other stack-based languages (*e.g.* Forth [41] or PostScript [1]), the data in the stack can also be directly accessed by some instructions other than push/pop operations (*e.g.* connect, dup: see table 1) and if an instruction cannot be executed successfully, it is simply discarded. When all tokens have been interpreted, the nodes (each one carrying connections to other nodes) are popped from the stack and the graph is ready to use.

Since any sequence of tokens is meaningful, the graph-building language is highly robust to any variation 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 different genotypes.

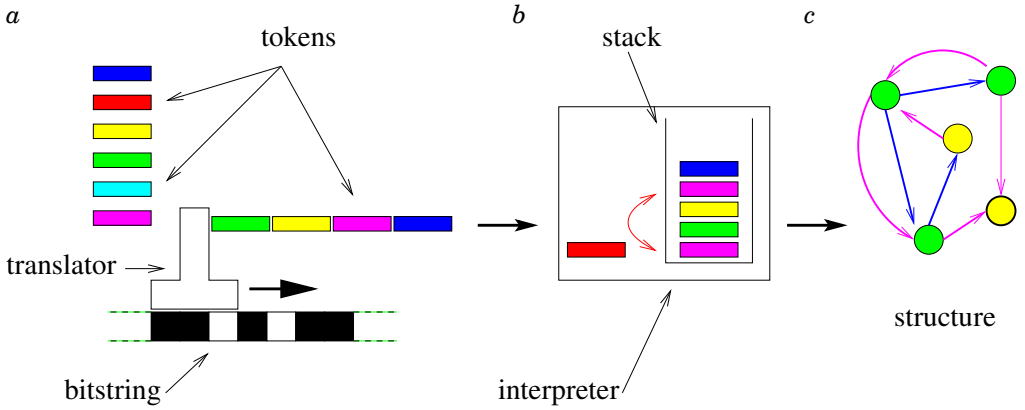


Fig. 1. Principles of the genetic expression used to produce the behavioral graph from the bitstring genotype. The string is first decoded into tokens (a), which are interpreted in a second step as instructions (b) to create nodes, edges, and labels. Finally, when all tokens have been interpreted, the unused conditions and actions remaining in the stack are added to the structure and the structure is popped from the stack (c).

Table 1 details the tokens that are used to build the graphs. There are three categories of token:

- behavior tokens (actions, conditions) that are specific to the description of a behavioral structure. These tokens are just pushed onto the stack.
- structure tokens (*node*, *connect*, ...) that perform atomic graph building steps, eventually using tokens already in the stack. These tokens are independent from the agent abilities.
- stack tokens (*swap*, *dup*, ...) that manipulate the stack. They are both independent from the agent abilities and from the structure built.

token	resulting actions
behavior tokens	actions and conditions tokens, specific to the definition of behavior
`condition?`	push the condition on the stack
`action!`	push the action on the stack
structure tokens	create nodes and connect them with edges
node	create a new node and push it on the stack
connect	create an edge from the first to the second node token [a], label the edge with the set of condition tokens and the list of action tokens from the top to 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 condition tokens and the list of action tokens from the top to the first 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 condition tokens and the list of action tokens from the top to the first node, delete the action and condition tokens that were used
selfConnect [b]	create an edge from the first node to itself label the edge with the set of condition tokens and the list of action tokens from the top to the first node, delete the action and condition tokens that were used
defaultSelfConnect [b]	create an edge for each node in the stack to itself label the edges with the set of condition tokens and the list of action tokens from the top to the first node, delete the action and condition tokens that were used
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 [c]
popRoll	pop the token, and put it on the bottom of the stack
pushRoll	take the token from the bottom of the stack, and push it

[a] if both nodes are copies of the same node, the result is a self-connected edge

[b] additional token

[c] if the stack contains copies of that node, the copies remain in 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. The additional tokens are not strictly necessary since their effect might be obtained by other means. They were introduced in ATNoSFERES in order to cope more easily with generalization.

In [27], we demonstrated that the performance of ATNoSFERES could be increased by using a new structure token, *selfConnect*, endowing our model with the ability to build self-connecting edges from a node to itself more easily. This new token has been used in all the experiments presented below.

2.2 Integration into an evolutionary framework

In this paper, the ATNoSFERES model is applied inside an evolutionary algorithm to produce the controller of agents. Each agent has a bitstring genotype from which it can build a control graph. 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 be added or removed by the evolutionary process, as can condition/action labels on the edges.

To evaluate an agent, we use those graphs as follows (see also figure 3):

- 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 chosen in this set. The first versions of ATNoSFERES were choosing the eligible edges randomly, but then we found that a deterministic choice (*e.g.* choose the first edge in the list of eligible edges) yieldeld better results. 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 step.

(3) The actions on the label of the current edge are sequentially performed by the system. Assuming that only one action can be performed at a time, only the last action is actually performed. When the action part of the label is empty, an action is chosen randomly. In all the experiments described in this paper, the action part of all edges contains at most one action, in order to simplify the comparison with LCSs.

(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).

3 LCSs, non-Markov problems, and ATNoSFERES

3.1 Learning Classifier Systems

As explained in the introduction, LCSs are rule-based systems combining RL algorithms with generalization capabilities. In the context of multi-step problems, the problems tackled by LCS are characterized by the fact that situa-

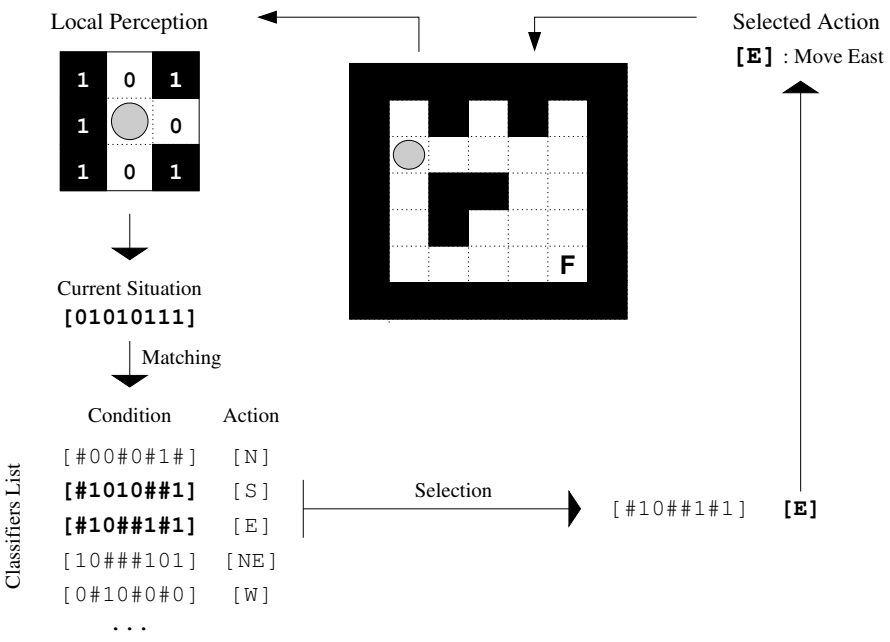


Fig. 2. The sensorimotor loop with a standard LCS. The agent perceives the presence/absence (resp. 1/0) of blocks in each of the eight surrounding cells and must decide towards which of the eight adjacent cells it should move. The agent perceives `[01010111]` (starting north and rotating clockwise). 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.

tions are defined by several attributes representing perceivable properties of the environment. In this context, an LCS builds classifiers so as to define the behavior of an agent as shown in figure 2.

An historical presentation of research on LCSs can be found in [62] and [31]. Here, we will just present the modern view of LCSs resulting from the radical simplifications achieved by Wilson in designing ZCS [60] and XCS [61].

In this view, an LCS evolves a population of classifiers with a `[Condition]` part, an `[Action]` part and some measure of the benefit that the system can draw from performing the action if the condition holds. Generally, this measure is updated thanks to an RL algorithm and the population of classifiers evolves according to a Genetic Algorithm (GA) [12] using this measure as fitness function. Within this framework, the use of "`#`" symbols in the `[Condition]` part of the classifiers results in a generalization capability, since *don't care* symbols make it possible to use a single vector of symbols to describe several situations [1]. Indeed, a *don't care* symbol *matches* any particular value of the considered attribute.

[1] For instance, in a binary language, `[1#0]` matches both `[100]` and `[110]`.

In most early LCSs [18], the fitness of each classifier was defined directly according to the expected utility associated to the classifier. After having defined a very simple LCS called ZCS, [60] found it more efficient to use the accuracy of the utility prediction rather than the expected utility itself. In particular, this fitness measure results in a better coverage of the state space since accurate classifiers predicting a low reward far from the source of reward will be kept. The resulting system, XCS [61], is now the most widely used LCS in the context of Markov problems and data mining problems [2].

Anticipatory Learning Classifier Systems (ALCSs) are a more recent family of LCSs. The first ALCS, ACS, has been developed by [50] and then improved as ACS2 by [3]. It differs from classical LCSs by adding to the `[Condition]` and `[Action]` parts an `[Effect]` part that represents a perceptual anticipation of the consequences of the action upon the environment. ACS relies on an Anticipatory Learning Process (ALP) [50] and has been successfully applied to both Markov [52] and non-Markov [51] environments.

The main feature of ACS with respect to standard LCSs is that their use of anticipation make it possible to use some efficient heuristics helping the system to converge faster, though no explicit performance comparison has been published yet. GÃ©rard and Sigaud have proposed two ALCSs similar to ACS, namely YACS [11] and MACS [10]. These systems have been shown to be faster than ACS and to give rise to a more compact set of classifiers, but they are limited to Markov and deterministic environments.

3.2 Pittsburgh versus Michigan style

A LCS usually uses a GA to replace some classifiers for improved performance. But there are two different ways of using GAs in this context, giving rise to the distinction between *Pittsburgh* style LCSs (*e.g.* [49]) and *Michigan* style LCSs (*e.g.* [17]).

In the *Pittsburgh* style, the GA evolves a population of controllers with their whole list of classifiers. The lists of classifiers of different controllers are combined using crossover operators and modified with mutations. The controllers are evaluated according to a fitness measure and the more efficient ones – with respect to the fitness – are kept. Thus a *Pittsburgh* style LCS evolves a population of controllers. On the contrary, in the *Michigan* style, the GA evolves a population of classifiers belonging to a single controller: classifiers are the individuals combined and modified by the GA. A fitness is computed for each classifier and the best ones are kept. Thus Michigan style LCSs use a GA to perform online learning: the classifiers are improved during the life time of

the agent. Usually, such LCSs rely on utility functions that depend on scalar rewards given by the environment, as defined in the RL framework (see [55]).

Most recent LCSs are *Michigan* style systems. As we will explain in the following sections, several research trends attempt to combine the advantages of the *Michigan* approach with those of the *Pittsburgh* approach.

3.3 Learning Classifier Systems on non-Markov problems

Dealing with simple `[Condition][Action]` classifiers does not endow an agent with the ability to behave optimally in perceptually aliased problems. In such 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, page 17, for an example).

In such a case, it is necessary to provide the system with more than just current perceptions. In the general RL framework, several kinds of solutions have been tested.

- The first one consists in adding explicit internal states to the perceptions involved in the decisions of the system. This approach was used by Holland in his early LCSs thanks to an internal message list [18]. But both [44] and [48] reported unsatisfactory performance of Holland's system on non-Markov problems. In the context of more recent LCS research, a different form of the explicit internal state solution was adopted by [6] in ZCSM and by [33] in XCSM and XCSMH.
- The second one, memory window management, is a special case of explicit internal state management where the internal state consists in an immediate memory of the past. Some systems use a fixed size window (see [34] for a review) while others use a variable size window (*e.g.* [37]). The next solution, rule-chaining, can be seen as an alternative view of the variable size window mechanism.
- The third one consists in chaining the decisions, making one decision depend on the decisions previously taken, so as to use a memory of previous actions to disambiguate the current situation. Among LCSs, this solution was used in ZCCS [57], CXCS [56] and ACS [51].
- The fourth one consists in splitting a non-Markov problem into several Markov problems, making sure that aliased states are scattered among different sub-problems. This solution has been investigated first by [59], and then improved by [54]. To our knowledge, no LCS actually uses this solution, despite its very interesting properties.

- The last solution consists in building a finite state automaton corresponding to the structure of the problem, as [39] or [15] do, in a context where the structure of the problem is known in advance. This is the solution chosen in ATNoSFERES, but in a context where the agents do not know anything about the structure of the problem before starting. To do so, we use a Pittsburgh style evolutionary algorithm.

In the remainder of this section, we will examine more closely two strategies that have been used to solve non-Markov problems in LCSs, namely Explicit Internal State Management and Classifier-Chaining.

3.3.1 Explicit Internal State Management

Shortly after the original ZCS [60] was published, an explicit internal state was added [6] to ZCS. The resulting system was called ZCSM, where M stands for Memory.

Following the same idea and suggestions already present in [61], Lanzi [29] proposed XCSM as an extension of XCS with explicit internal states. XCSM manages an internal memory register composed of several bits that explicitly represents the internal state of the LCS. The memory register provides XCSM with more than just the environmental perceptions. Thus, dealing with perceptual aliasing is made possible by adding information from the past experience of the agent. As a result of this addition, a classifier contains four parts: 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 `[Internal action]` parts contain as many attributes as there are bits in the memory register. In order to be selected by the LCS, a classifier must 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 *don't change* symbols "`#`". When a classifier is fired, a `#` in the internal action results in not changing the corresponding bit in the memory register. XCSM benefits from generalization in the external condition (like XCS), in the internal condition and also in the internal action.

Results obtained with XCSM on several benchmark environments were presented in [33], **Maze10** (see figure 4, p. 17) being the most difficult. The very poor performance of XCSM on complex non-Markov problems led Lanzi to propose a further extension, XCSMH, where H stands for hierarchical. The only difference between XCSM and XCSMH is that in the latter the dynamics of the internal state is deterministic even during exploration, and that the

internal state cannot change if the perceptions do not change. This minor modification resulted in a significant improvement in performance (see [33]).

3.3.2 Classifier-chaining mechanisms

LCSs that use a classifier-chaining mechanisms are called corporate classifier systems. The first suggestion of a corporate classifier system appears in [62]. The authors provide a critical review of early LCSs and express the need for a good combination of *Michigan* and *Pittsburgh* approaches. After reviewing a few earlier seminal works, the solution they suggest consists in creating "corporations" of classifiers by linking together classifiers often fired in sequences. To our knowledge, the corresponding system has never been implemented.

More recently, ZCCS, a corporate classifier system inspired from [62] and based on ZCS [60], has been implemented [57], probabilistically linking classifiers into "behavioral sequences". In general, the length of the behavioral sequences is arbitrarily limited to a given parameter. This solution is adequate for solving perceptual aliasing problems since a behavioral sequence can result in bridging aliased situations: the sequences that help solve perceptual aliasing problems are sequences starting before the ambiguous states and ending after these states have been crossed. Since the ambiguous states are hidden in sequences, the agent never needs to guess what to do in an ambiguous context. The success of this solution relies on the probabilistic nature of the sequence building algorithm. Sequences providing a selective advantage (because they bridge an aliased state) are kept, the others are discarded.

The same authors then proposed CXCS [56], a second corporate classifier system based on XCS. In [58], they experimented a lot of simple variations on the original corporation mechanism, exploration rate and corporate representation. Unfortunately, they tested their systems mostly on Markov problems and on a non-Markov hand-crafted "delayed reward task" that is too simple to provide a serious benchmark for comparison.

On a different line of research, another classifier-chaining mechanism was used in ACS in order to deal with non-Markov environments [51]. In that case, the `[Effect]` part of a classifier consisting in a behavioral sequence is intended to represent the perceptual consequence of the sequence of actions. The mechanism is restricted to deterministic domains and applies classifier-chaining in particular situations, whereas CXCS forms many more connections so as to find the relevant ones by chance. Despite its restricted use, its classifier-chaining mechanism makes ACS able to deal efficiently with deterministic non-Markov environments. In order to build behavioral sequences, a new parameter was added to ACS, namely "BS_{max}". BS_{max} represents the maximal length of the behavioral sequences that ACS may build.

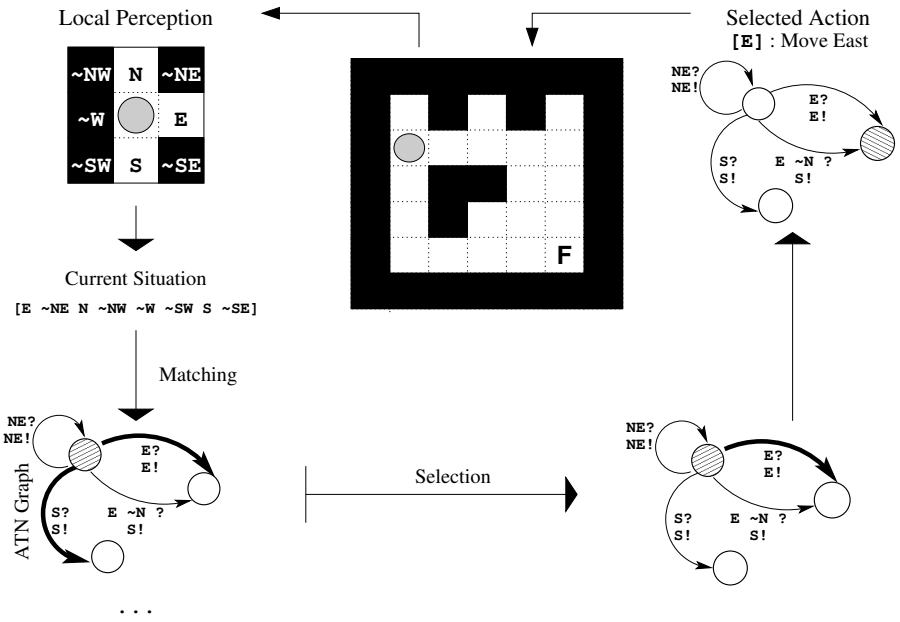


Fig. 3. Using ATNoSFERES to control an agent. In this example, the agent, located in a cell of the maze, perceives the presence/absence of blocks in each of the eight surrounding cells. It has to decide towards which of the eight adjacent cells it should move. From its current location, the agent perceives `[E ¬NE N ¬NW ¬W ¬SW S ¬SE]` (token `E` is true when the east cell is blank). From the current state (node) of its graph, two edges (in bold) are eligible, since the `[Condition]` part of their label matches the perceptions. One is selected, then its action part (move east) is performed and the current state is updated.

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. Indeed, an ATN such as those evolved by ATNoSFERES can be easily translated into a list of classifiers, either using an explicit internal state or a classifier-chaining mechanism. Whatever the LCS formalism, the conditions associated with the edges correspond to the conditions of the classifiers and the actions associated to the edges correspond to the actions of the classifiers.

In the explicit internal state view, the nodes of the ATN play the role of internal states and endow ATNoSFERES with the ability to deal with perceptual aliasing. The source and destination nodes of the edge correspond to internal states.

In the classifier-chaining mechanism, all edges can be translated into classifiers either independently from each other or in chains of any lengths, linking edges ending in a node to another edge starting from the same node.

Thus our model has the potential to express the different mechanisms found in LCSs listed above when confronted to non-Markov problems.

4 Experimental Comparisons

4.1 Experimental setup

All experiments presented in this paper take place in the context of non-Markov multi-step problems. In order to carry out some comparisons, we use benchmark experiments involving an agent looking for some food in a simple maze-like environment (see figures 4 and 6). We tried to reproduce an experimental setup as close as possible to that used in [29] to test XCSM and XCSMH in the **Maze10** environment and in [38] to test ACS in **E1** and **E2** environments, taking into account the specificities of our model. The same setup has been applied to all the experiments presented in this paper.

4.1.1 Perception/Action abilities and Tokens

The agents used for the experiments can perceive the presence/absence of walls or the presence of food in the eight adjacent cells of their environment, these three perceptions being mutually exclusive. They can move in all adjacent cells (the move will be effective if the cell is blank or contains food). Thus, the genetic code includes 24 condition tokens and 8 action tokens that depend on this particular setup. Additionally, it includes 7 stack manipulation tokens and 4 node creation/connection tokens that are independent of the problem. With respect to the encoding used in previous publications, we used a new 6 bit encoding to define these 43 tokens. Previously we used either 7 bit long tokens [28] or different 6 bit long tokens [26,27]. Table 2 details the genetic graph building code.

000000 *swap*	000001 *swap*	000010 *swap*	000011 *swap*
000100 *dup*	000101 *dup*	000110 *dupNode*	000111 *dupNode*
001000 *del*	001001 *del*	001010 *delNode*	001011 *delNode*
001100 *popRoll*	001101 *popRoll*	001110 *pushRoll*	001111 *pushRoll*
010000 *node*	010001 *node*	010010 *node*	010011 *node*
010100 *connect*	010101 *connect*	010110 *connect*	010111 *connect*
011000 *selfConnect*	011001 *selfConnect*	011010 *selfConnect*	011011 *startConnect*
011100 *startConnect*	011101 *startConnect*	011110 *endConnect*	011111 *endConnect*
100000 *goNorth!*	100001 *goSouth!*	100010 *goWest!*	100011 *goEast!*
100100 *goNorthEast!*	100101 *goSouthEast!*	100110 *goNorthWest!*	100111 *goSouthWest!*
101000 *freeNorth?*	101001 *foodNorth?*	101010 *occupiedNorth?*	101011 *freeSouth?*
101100 *foodSouth?*	101101 *occupiedSouth?*	101110 *freeWest?*	101111 *foodWest?*
110000 *occupiedWest?*	110001 *freeEast?*	110010 *foodEast?*	110011 *occupiedEast?*
110100 *freeNorthEast?*	110101 *foodNorthEast?*	110110 *occupiedNorthEast?*	110111 *freeSouthEast?*
111000 *foodSouthEast?*	111001 *occupiedSouthEast?*	111010 *freeNorthWest?*	111011 *foodNorthWest?*
111100 *occupiedNorthWest?*	111101 *freeSouthWest?*	111110 *foodSouthWest?*	111111 *occupiedSouthWest?*

Table 2
The genetic code used in the experiments.

Since $2^6 = 64$, some tokens can be encoded twice or more. The choice to encode one token or another several times as it appears in table 2 is arbitrary: we took into account some soundness constraints such as having the same number of tokens with symmetrical effects (e.g. *dupNode* and *delNode*) but we did not try to improve it over several experiments. Looking for an optimal mapping between tokens and encoding would be computationally too expensive and is currently outside the scope of more theoretical tools such as the ones developed in [43].

4.1.2 Course of Experiments

Each experiment involves the following steps:

(1) Initialize the population with $N = 300$ agents with random bitstrings.
(2) For each generation, build the controller of each agent and evaluate it in the environment.
(3) Select the 20 % best individuals of the population and produce new ones by crossing 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) Go back to step 2 with the new generation.

4.1.3 Fitness function.

Each individual is evaluated in the environment, starting on a blank cell in the environment and looking for food within a limited duration (20 time steps in all experiments described below). The agent can perceive the food only in its immediate neighborhood, 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 has no effect (the agent loses one time step and stays on the same cell).

The fitness of the agent for each run is the remaining time if the food has been found within the time limit, 0 otherwise. Thus, the selection pressure encourages short paths to food. To evaluate one generation, each agent is evaluated once starting on each blank cell, then its total fitness for this generation is the sum of the fitnesses computed for each run.

Each agent is reevaluated at each generation in order to average its fitness over generations. This is necessary because of non-determinism in the automata. Indeed, in a situation where no arc is eligible, or when an edge does not carry any action label, one action is chosen randomly. Thus an automaton will be fully deterministic only in the case where one arc can be elected in any encountered situation, and if all such arcs bear an action to perform.

4.2 Experimental environments

4.2.1 Maze10

We first tested our model in the **Maze10** environment (see fig. 4), for which [29] provides empirical results obtained with XCSM and XCSMH.

This environment is non-Markov, and presents 13 aliased situations (among the 18 free cells) which are perceived as 5 distinct situations. In figure 10, page 21, we show the optimal number of steps necessary to reach food from each starting cell, given a limited perception (an omniscient agent could perform even better). More precisely, in order to compute this optimal policy, we take into account the fact that an agent cannot choose the correct action in a perceptually aliased situation if it cannot determine its actual state from its past. In particular, an agent starting from an ambiguous situation can only choose an action from the probability that each action is the most favorable one across the possible states.

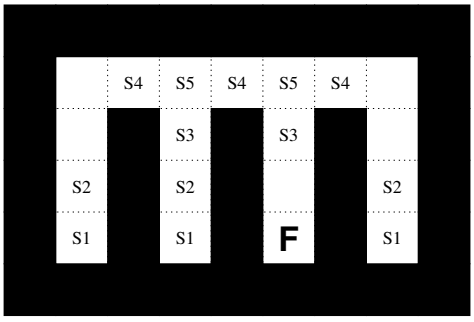
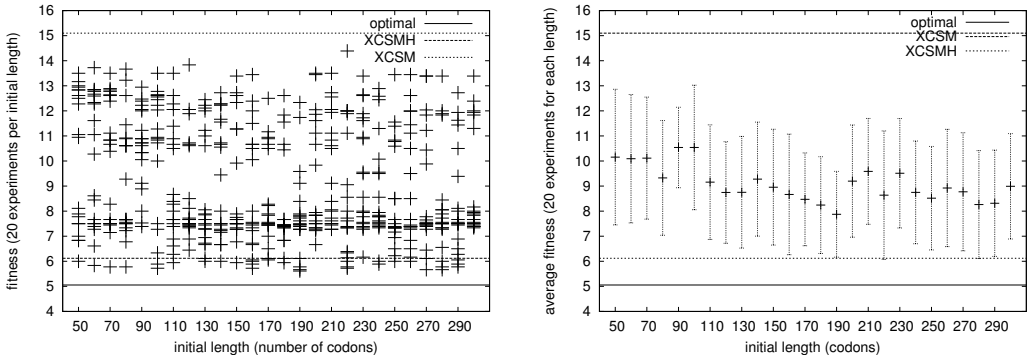


Fig. 4. The **Maze10** environment. **F** represents the goal (food). Other marked cells represent aliased situations (identical names imply the same perception, but eventually not the same optimal action).

The experiments reported in [26] were carried out on various initial genotype sizes, from 50 to 90 codons of 6 bits, and the food was *not* perceived. The original population genotype sizes changed during evolution. Here, we reconducted the experiments including the ability to perceive food, with initial genotype sizes from 50 to 300 codons of 6 bits. Each experiment is stopped after 10,000 generations and 20 experiments have been performed in each experimental situation.

Figures 5 (a) and (b) show the performance of ATNoSFERES on this task compared with the performance of XCSM and XCSMH. Each cross in figure 5 (a) represents the performance of the best automaton obtained after 10,000 generations in one run. There are twenty crosses for each initial length. It can be seen that the performance of ATNoSFERES generally lies between the per-

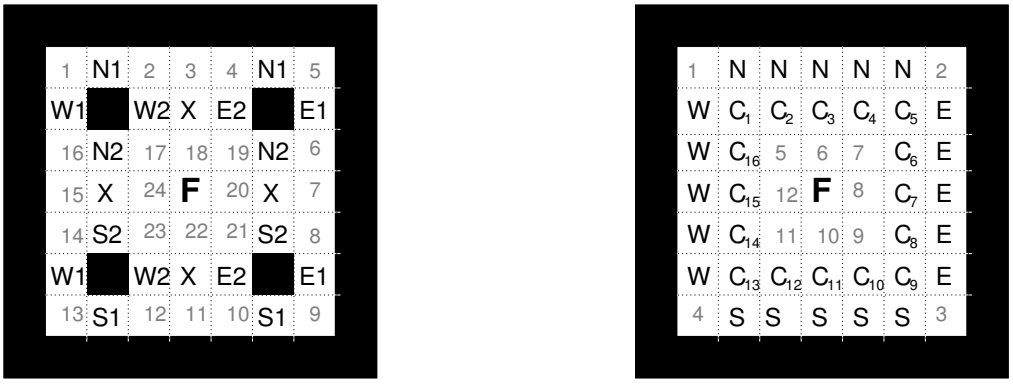


(a) distribution (b) average

Fig. 5. (a) Best average times over complete runs to reach food in **Maze10** experiments as a function of the initial length of the bitstring, compared to XCSM and XCSMH performance. Since the numerical value of XCSM(H) performance was not given in publications, we have extracted it from figure 12 in [29], zooming in 16 times the (vector graphics) curves. (b) Average of the results shown in (a).

formance of XCSM and the performance of XCSMH, but that ATNoSFERES often outperforms XCSMH, going very close to the optimum. This result will be discussed in § 5.

4.3 E1 and E2



(a) **E1** environment (b) **E2** environment

Fig. 6. **E1** and **E2** environments. **F** (food) is the goal. Capital letters marked cells represent aliased situations (identical letters imply the same perception). The remaining non-aliased marked cells are numbered so as to clarify the example given in page 27.

The experiments described below take place in **E1** and **E2** environments (see figure 6) that have been used in [38] to study how ACS deals with non-Markov problems. **E1** presents 20 aliased situations (among the 44 free cells) which are

perceived as 9 distinct situations. **E2** presents 36 aliased situations (among 48 free cells), which are perceived as 5 distinct situations.

In figure 13, page 23, we show the number of steps an optimal agent among several may need to reach food from each starting cell, given that its perception is limited, as explained in § 4.2.1.

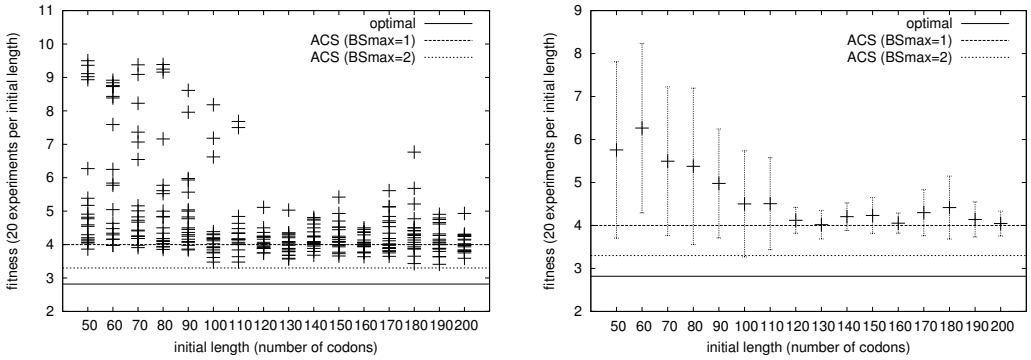
Before comparing ACS with ATNoSFERES, we have to emphasize a major difference between the way both systems deal with these environments. In ACS experiments, as they are described in [38, § 4.1 and 4.2], the only movements tested in each free position are transitions towards surrounding free cells (for example, if the cell to the north contains an obstacle, going north is not considered as a possible move, thus it cannot be selected). This constitutes a kind of prior domain-dependent knowledge about consistent perceptions-actions bindings, which significantly biases the learning process by reducing the number of possibilities. In [47], we have shown that prohibiting the use of this bias can severely impair some learning algorithms. For instance, the U-Tree algorithm [37] which works well in non-Markov mazes (such as those studied here) if the agent is prevented from bumping into walls, might grow an infinitely deep tree if it keeps bumping into the same wall in an aliased situation.

In ATNoSFERES, on the contrary, any action can be used at any time. When the corresponding move is impossible, the agent stays where it is and loses a time step (it is penalized only in an indirect way, through the fitness function).

The experiments reported in [28] were carried out on various initial genotype sizes (from 50 to 150 codons), but the codons were 7 bit long. The experiments presented here are reconducted with 6 bit long codons, and the genotypes are between 50 and 200 tokens long (with step 10) in **E1**, and between 50 and 300 tokens long (with step 10) in **E2**. As with **Maze10**, each experiment is stopped after 10,000 generations. For **E1** (resp. **E2**), 20 experiments (resp. 10 experiments) are performed in each experimental situation.

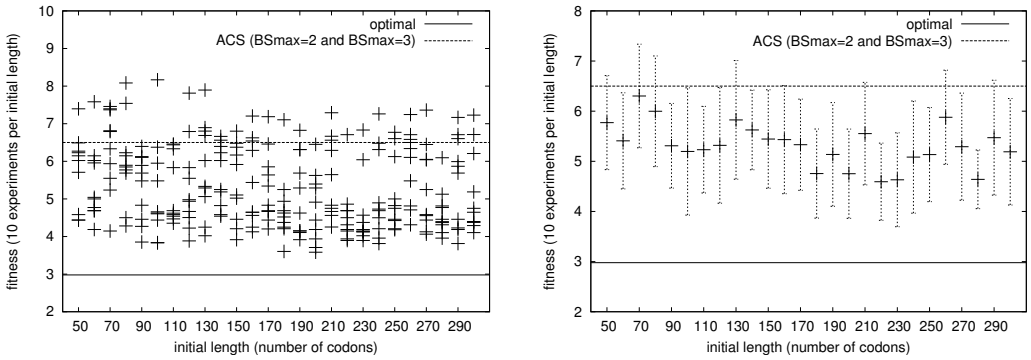
Figures 7 and 8 give the respective fitness values obtained by the best automata in **E1** and **E2** experiments, depending on initial lengths of the genotypes, and the average of these values, compared with the values published by [38] on ACS.

From figure 7 (a), it can be seen that in **E1**, ATNoSFERES easily reaches the performance of ACS in the case where $BS_{max} = 1$, but never reaches the performance of ACS with $BS_{max} = 2$, which is very close to the optimum. On average (see figure 7 (b)), ATNoSFERES reaches the performance of ACS in the case where $BS_{max} = 1$ as soon as the initial length of the genotype is over 110.



(a) distribution (b) average

Fig. 7. (a) Best average times over complete runs to reach food in **E1** experiments as a function of the initial length of the bitstring. (b) Average of the results shown in (a).

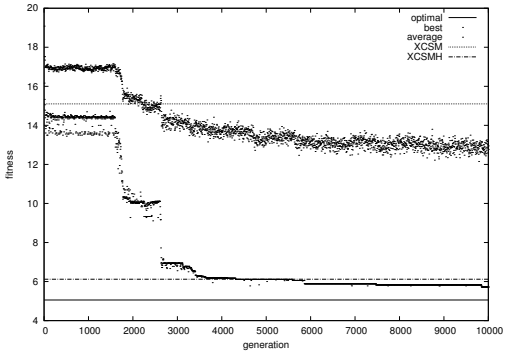


(a) distribution (b) average

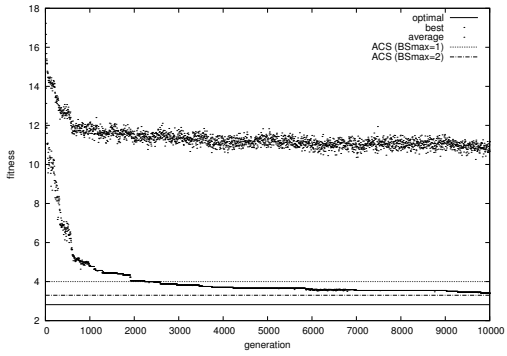
Fig. 8. (a) Best average times over complete runs to reach food in **E2** experiments as a function of the initial length of the bitstring. (b) Average of the results shown in (a).

On the other hand, in **E2**, the performance obtained with ATNoSFERES (see figure 8) is significantly better than the one obtained with ACS with $BS_{max} = 2$ and $BS_{max} = 3$. Indeed, ATNoSFERES is always performing better than ACS on average.

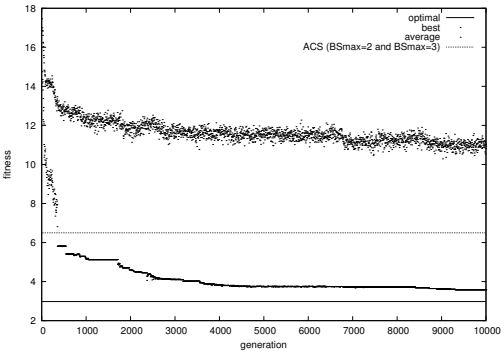
Figure 9 gives the evolution of the best fitnesses. We see that gradual improvements occur in the three environments.



(a) Maze10



(b) E1



(c) E2

Fig. 9. Representative evolution of the best fitness in **Maze10**, **E1**, and **E2** experiments as a function of generations; the shape and smoothness of the curves have been chosen as representative evolutions. The thickness of the curves (particularly manifest in **Maze10** and **E1**) is due to the non-deterministic behavior of agents. In **E1** and **E2**, it seems that the pressure towards deterministic behavior is stronger.

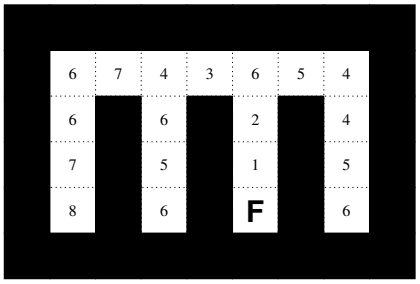


Fig. 10. An optimal policy in **Maze10**, represented by the number of steps needed to reach food from each start cell. The optimal average number of steps to food is 5.0555.

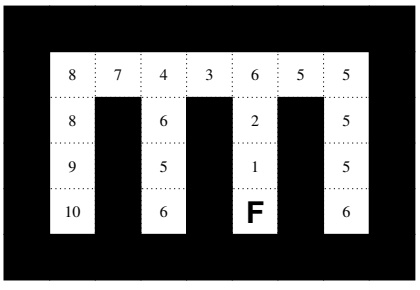


Fig. 11. Best policy found with ATNoSFERES in **Maze10** in 10,000 generations (see figure 10 for an optimal policy). Its average number of steps to food is 5.6111.

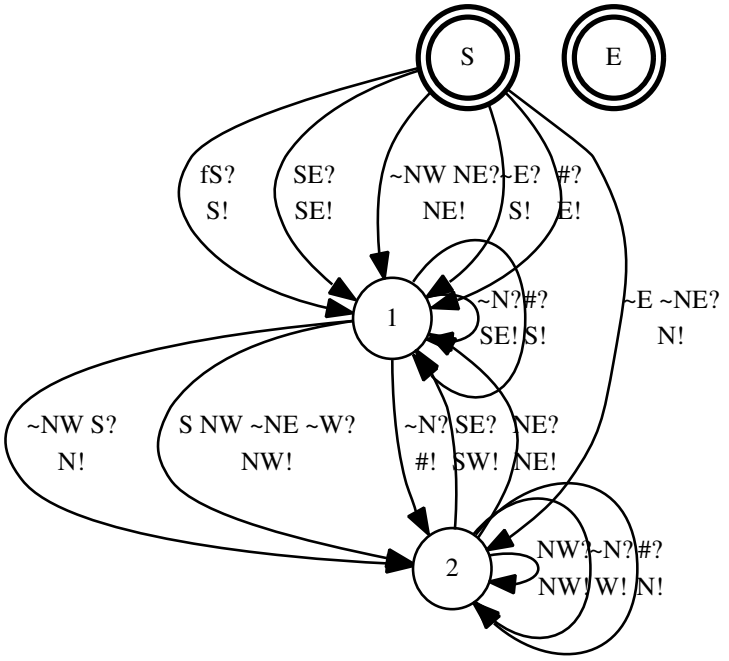


Fig. 12. The best automaton found with ATNoSFERES in **Maze10** experiment (after 10,000 generations). Its average number of steps to food is about 5.6111

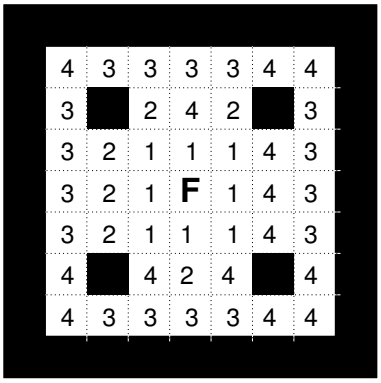
4.4 Representative solutions

4.4.1 Maze10 environment

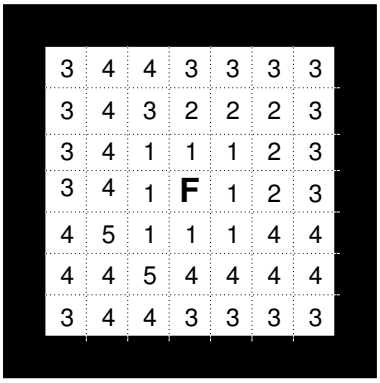
As in previous experiments [26], the graph of the best automaton (see figure 12) has two internal nodes. This automaton is representative for the best solutions found with this environment. By scanning all the best solutions corresponding to crosses in figure 5 (a), we see that some good solutions have three internal nodes, but most good solutions have two internal nodes. The solutions with only one internal node have the worst performance (more than 10 steps to food on average). Thus the best solutions are the ones connecting two nodes with the good set of edges.

Thanks to the ability to perceive food, the fitness reached is higher than in our previously published experiments. A comparison between figure 10 and 11 shows that the policy of the best automaton is very close to the optimal policy. Two steps are lost for every cell in the left-wing column because the agent goes to the bottom of the second left-wing column to check whether the food is there while this is not necessary: the left-wing top corner cell is sufficient to disambiguate the situation. Other steps are lost in the top of the right-wing column because the agent goes north instead of going west or north-west, which would be optimal.

4.4.2 E1 and E2 environments

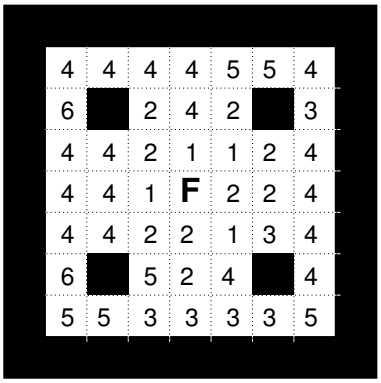


(a) **E1** environment

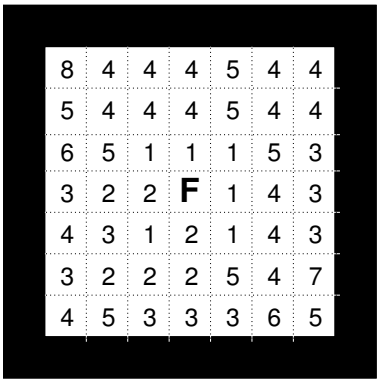


(b) **E2** environment

Fig. 13. An optimal policy in **E1** (resp. **E2**), represented by the number of steps needed to reach food from each start cell. Other equivalent policies can be obtained at least by applying all possible rotations and symmetries to all the numbers given. In **E1**, the optimal average number of steps to food is 2.8181 steps. In **E2**, it is 2.9792 steps.



(a) **E1** environment



(b) **E2** environment

Fig. 14. Best policies found with ATNoSFERES in **E1** (resp. **E2**) in 10,000 generations, represented by the number of steps needed to reach food from each Start cell (see figure 13 for an optimal policy).

Figures 13 and 14 show some optimal policies on **E1** and **E2** environments respectively, and the best policies found on these environments with ATNoSFERES. On figure 14, we can see that in some situations where food is visible the agent needs more than one step to reach it, though a more efficient behavior is obvious. ATNoSFERES has more difficulties in finding these simple rules than an RL algorithm combining exploration and exploitation would. This will be discussed in § 5.4.

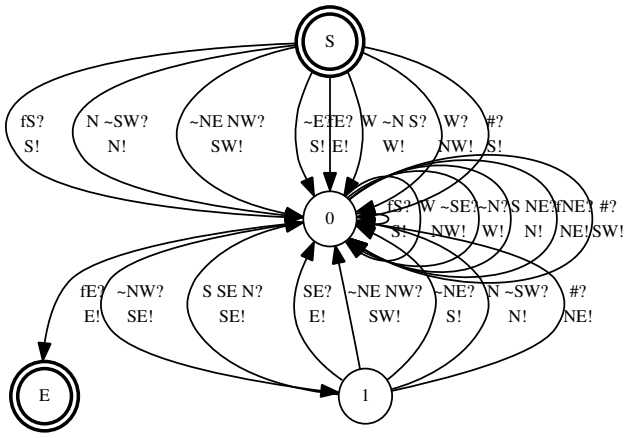


Fig. 15. The best automaton found with ATNoSFERES in **E1** experiment (after 10,000 generations). Its average number of steps to food is about 3.4091

The best automaton shown in figure 15 is representative of the best solutions found in **E1** environment. We obtained similar results in previous experiments [28] with a 7 bit encoding, but with a slightly better fitness (about 3.3 steps to food on average, performing as well as ACS with $BS_{max} = 3$).

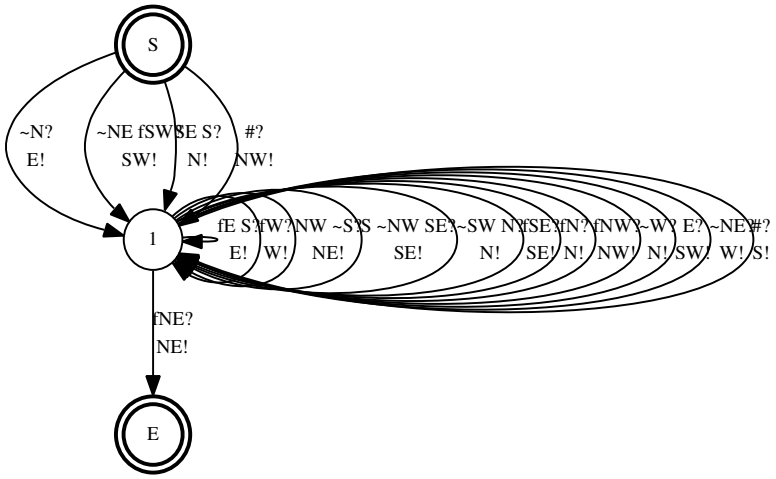


Fig. 16. A representative good automaton found with ATNoSFERES in **E1** experiment (after 10,000 generations). Its average number of steps to food is about 3.75

In **E1**, 46% of the automata performing better than ACS with $BS_{max} = 1$ contained only a single node (in addition to the Start and End nodes that always exist in ATNoSFERES graphs), which means that a reactive behavior already performs well in this environment (see figure 16 for a representative example).

Figure 17 gives the best automaton found in **E2** environment. Representative best automata all have at least two internal nodes. Automata with only one internal node are the least efficient, but they generally perform better than ACS with $BS_{max} = 2$ or 3. It is clear from our experiments that a good

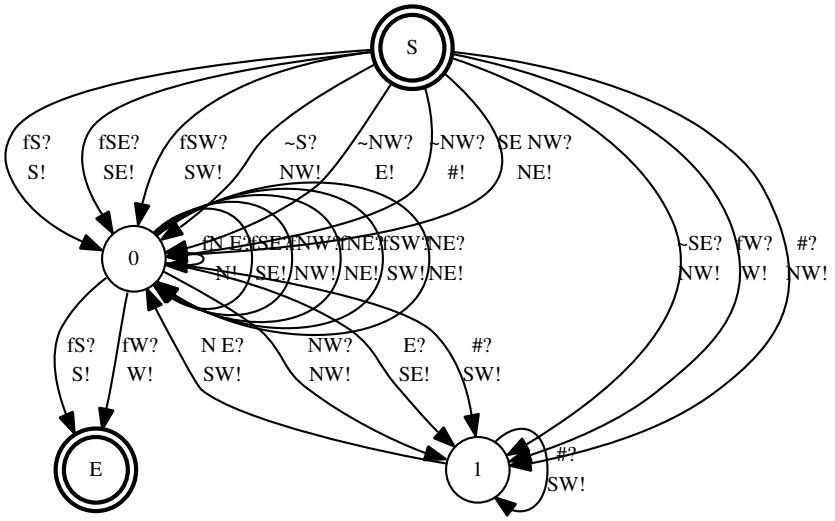


Fig. 17. The best automaton found with ATNoSFERES in **E2** experiment. Its average number of steps to food is about 3.5833

automaton in **E2** requires more nodes than in **E1**. This seems to imply that reactive and nearly reactive behaviors perform much worse in **E2** than in **E1**. This fact will be discussed in the next section.

A closer study of the automata obtained in **E1** and **E2** environments reveals that the main improvement mechanisms consists in adding one node when they contain a single node, and then add more edges to make profit of relevant situation-action couples. For instance, the performance is increased each time ATNoSFERES builds a new edge binding the perception of food to a move toward food. As we will discuss in the next section, this implies that good solutions generally contain more edges connected to the same nodes than poorer solutions.

5 Discussion

5.1 Nearly Markov versus highly non-Markov problems

As a result of the new experiments performed in this paper, it appears that different subclasses of non-Markov problems should be distinguished more accurately than is usually done. Indeed, some problems, like **E1**, are actually non-Markov, but in such a way that reactive behaviors can still perform well on such problems, as shown by the performance of all automata with one node, exemplified in figure 16. We call them "nearly Markov problems".

In **E1**, our study has shown that an evolutionary process can gradually grow a set of correct rules (which are to some extent independent from each other), even more if the agent is tested from each cell. Thus an agent can start with a few rules that are efficient for a few cells, and add from one generation to another new rules that are useful for additional cells. Such reactive solutions already perform well in **E1**, and are easier to find for ATNoSFERES than solutions implying several internal states. Indeed, building more complex solutions requires both additional nodes and consistent edges, with appropriate conditions and actions. We meet again the *structural cost* mentioned in [26]: "simple", incremental good solutions are preferred to structurally complex optima. As a matter of fact, ACS with $BS_{max} = 2$ performs better than ATNoSFERES on **E1**.

On the contrary, other environments like **E2** and **Maze10** should be called "highly non-Markov", since reactive policies perform very poorly on such problems, due to the location and the number of aliased situations.

Our comparative study has revealed that ACS performs very well on **E1** and relatively poorly on **E2**, while ATNoSFERES performs consistently on both environments. In the remainder of this discussion section, we will try to explain why that is so.

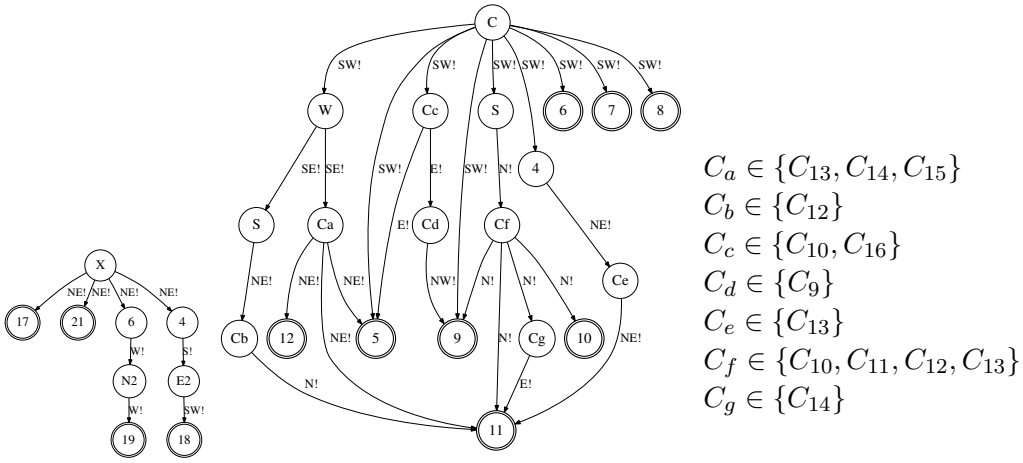
5.2 Limits of classifier-chaining

The first point to explain is the fact that ACS performs relatively poorly on **E2**. On first thoughts, one might consider that the maximal length of sequences in ACS is the key. One could expect that setting BS_{max} to more than 3 in **E2** could fix the problem. A closer examination, however, reveals that this is not so.

In [38], the authors show that setting BS_{max} to 3 is enough to let ACS build a completely reliable model of **E2**, under the form of a list of `(situation, action, next situation)` classifiers. They mention that increasing the maximum length of the behavioral sequence "does not improve the 'steps to food' performances".

One explanation of the fact that building longer action sequences does not improve the performance is that these sequences specify a blind series of actions performed without interruption and without checking the situation perceived before its end. Once a sequence is elected, the agent will at least perform the number of actions specified in the sequence, unless it finds the food during the sequence. Since the number of steps to food given by the optimal policy in **E1** and **E2** is generally less than 4, it is very unlikely that letting the

agent perform sequences of 4 actions or more will help reaching the optimal performance.



(a) **X** in **E1** (b) **C** in **E2** (c) **C** aliases

Fig. 18. Optimal policies beginning in the **E1** and **E2** environments presented in figure 6, page 18, on the aliased **X** (resp. **C**) cell. Edges represent transitions, labelled by the performed action. Nodes represent observations, labelled by the cell identifier given in figure 6. Doubly-circled nodes represent next to final observations, linked to the final **F** cell, which is not represented here for readability purpose. The complete policy structures starting from all ambiguous cells would be unreadable, but all local policies share a similar structure for both problems.

A second explanation of the relatively poor performance of ACS on **E2** with respect to **E1** comes from an analysis of figure 18. This figure shows one optimal policy among several from selected ambiguous cells in the **E1** and **E2** environments. As it can be seen, optimal performance with a classifier-chaining mechanism is not possible in **E2** because the following happens:

- either we call upon 1-classifier-long sequences in the initial situation but the optimal policy also requires at the next time step classifiers with the same condition part (see the C_c node on figure 18).
- or we call upon longer sequences starting with the same condition part.

In both cases, the agent cannot choose the correct decision unit from the ambiguous situation. Thus the classifier-chaining mechanism is formally unsuitable in front of such structures since the agent must decide in advance which sequence of classifiers will be fired before having encountered the perceptions informing it of the correct choice. On the contrary, this mechanism is appropriate in the case of **E1** where the structure of the optimal policy is a simple list of action sequences connected to the same root node.

Indeed, in small environments like **Maze10**, **E1** and **E2**, the main issue for the agent consists in discovering as fast as possible where it is from an initially

ambiguous situation and then follow the shortest path to the goal. In **E1** this can be achieved with a list of classifier-chains while this is not the case in **E2**.

5.3 One-to-one versus many-to-many associations

As clearly explained in [58], associations in CXCS are direct and one-to-one while they are many-to-many in XCSM(H). More generally, in all systems relying on a classifier-chaining mechanism, the association can be said one-to-one because the corporation mechanism connects only one classifier to another. By contrast, an explicit internal state management mechanism builds many-to-many associations since all classifiers sharing the same `[Internal condition]` part can be fired after all classifiers that specify the corresponding `[Internal action]` part. When one compares the mechanisms in CXCS to the one in XCSM(H), one might consider that a one-to-one association gives a greater control on the dynamics of the internal state than a many-to-many association.

Indeed, in XCSM(H), the internal state value is very unstable and in [33] some special purpose mechanisms in XCSMH had been designed so as to overcome this problem. But our design of ATNoSFERES reveals that the problem does not come from many-to-many associations *per se*, but from the way they are represented in XCSM(H). Indeed, ATNoSFERES is clearly able to build many-to-many associations (when more than two edges are connected through the same node), as well as one-to-one associations (when just two edges are connected through the same node).

Thus ATNoSFERES combines the advantages of both approaches.

- Like XCSM(H), it is able to build many-to-many associations and we have seen in the previous section that this property is necessary to yield an excellent performance on environments like **E2** and **Maze10**.
- Like CXCS and ACS, it is endowed with a good control over the internal state dynamics, and can eventually build one-to-one associations, which makes the *ad hoc* mechanisms added in XCSMH unnecessary.

Being able to build both kinds of associations, ATNoSFERES is also able to select the most appropriate combination of them thanks to the GA. This explains why ATNoSFERES can outperform both XCSMH on some environments and ACS on others. But the fact that ACS with $BS_{max} = 2$ outperforms ATNoSFERES on **E1** seems to imply that ATNoSFERES encounters more difficulties to build efficient one-to-one associations when they are the most necessary, *i.e.* in nearly Markov problems. This can be easily explained since building exactly two edges connected by one node requires a lot of precise constraints on a genotype.

In our results, the behavior obtained on all environments with all systems are never fully optimal, despite the relative simplicity of these environments. Thus a performance comparison must consider two dimensions:

- how close to the optimum did the system get?
- how long did it take to get there?

We have shown that, with respect to the first point, ATNoSFERES can be compared favorably with the LCSs studied here. But, with respect to the second point, one important advantage of LCS over ATNoSFERES is that LCSs learn to act thanks to an RL algorithm.

Indeed, if we compare the number of elementary runs necessary to reach a good performance with LCSs against ATNoSFERES, the difference is clear (see table 3). In this table, the LCS column gives the average number of elementary runs after which the LCSs converge to the performance indicated, known as the best performance of the corresponding system (see § 4).

envir.	LCS type	perf.	LCS	$PR(\%)$	NG	NT	NT/LCS
Maze10	XCSM	15.1	7,000	100	8.42	45,000	6.42
Maze10	XCSMH	6.12	6,500	7.30	4909	360.10^6	55,400
E1	ACS $_{(BS_{max}=1)}$	4	4,400	30.92	5115	218.10^6	50,000
E2	ACS $_{(BS_{max}=2 \text{ or } 3)}$	6.5	2,000	83.84	835	14.10^6	7,000

Table 3
Comparison between ATNoSFERES and LCSs on the basis of the average number of trials necessary to reach the performance given in column "perf.". PR is the percentage of evolution runs with ATNoSFERES that outperform the corresponding LCS. NG is the average number of generations after which this happens (see figure 9, but the average has been computed from an exhaustive computation on all the runs presented on figures 5, 7 and 8). Thus, with 300 individuals per generation, the average number of evaluation runs necessary to outperform the corresponding LCS is $NE = 300 * NG * 100/PR$, and the average number of elementary runs NT for an environment with NS start cells ($NS = 18$ for **Maze10**, 44 for **E1** and 48 for **E2**) is: $NT = NS * NE$. Thus NT gives the average number of elementary runs needed by ATNoSFERES to outperform the corresponding LCS. Finally, NT/LCS gives a good approximation of the factor by which the corresponding LCS is faster than ATNoSFERES to reach its best performance. Note that no performance comparison is given on **E1** against ACS with $BS_{max} = 2$ since ATNoSFERES never outperforms it.

From this table it is clear that ATNoSFERES still needs several orders of magnitude more runs than XCSMH and ACS to converge (from 7000 to 200,000 times more).

This can be easily explained by the fact that ATNoSFERES evolves automata thanks to a blind GA process while ACS and XCSMH rely on an RL algorithm extracting information about the environment from its experience. From this comparison, it is clear that a direction for improving ATNoSFERES consists in endowing it with RL capabilities. This is our immediate agenda for future work.

A source of inspiration in that direction comes from the SAMUEL system [13]. Like ATNoSFERES, SAMUEL is a *Pittsburgh* style system based on a single chromosome GA, but it also includes Lamarckian operators that endow it with basic learning capabilities. As a result, as claimed by the author, "SAMUEL represents an integration of the major genetic approaches to machine learning, the Michigan approach and the Pittsburgh approach". Most of the operators used in SAMUEL can be transposed in ATNoSFERES, the main difference being that ATNoSFERES does not provide a high level symbolic representation and that SAMUEL does not include any dedicated mechanism to solve perceptual aliasing problems.

6 Conclusion

In previous papers, we have emphasized some advantages of ATNoSFERES over LCSs. In particular, ATNoSFERES builds minimal controllers, which results in their improved readability.

In this paper, we have applied ATNoSFERES to non-Markov benchmark environments that have been investigated with ACS, XCSM and XCSMH. This study gave us the opportunity to compare two mechanisms designed to deal with non-Markov problems, namely explicit internal state management and classifier-chaining. We have explained why the second was formally unable to reach optimality in some environments.

From that comparison, a new advantage of ATNoSFERES over the LCSs studied here is revealed. Since its representation has enough expressive power to include both one-to-one and many-to-many associations to solve non-Markov problems, the GA can select the most suitable approach or combination of approaches depending on the nature of the problem at hand.

This explains why ATNoSFERES can outperform both ACS and XCSMH on different environments. But the price to pay with that richer formalism is probably a slower convergence to good solutions. However, as long as no RL mechanism is included in ATNoSFERES, a fair comparison along that line cannot be provided. We are actively working on integrating RL mechanisms into ATNoSFERES, so we hope to provide more accurate comparisons soon.

Finally, we would like to highlight the fact that the comparative studies we have carried out with ATNoSFERES in this paper were indirect. We have compared ATNoSFERES with XCSM and XCSMH on one environment and ATNoSFERES with ACS on other environments, relying on the experiments presented in the available literature. A more direct performance comparison between XCSMH and ACS on the same environments would certainly be of interest, but results with these systems have never been published yet. A lot of work deserves to be done to provide more global comparisons between several classes of systems, including LCSs. We strongly believe that such comparisons would greatly enhance the understanding of the current state of the art in LCS research and, more generally, in the evolutionary computation approach to multi-step problems in information sciences.

References

[1] Adobe Systems. *The PostScript Language Reference Manual*. Addison-Wesley, Reading MA, 1985.

[2] E. Bernardó, X. Llorà, and J. M. Garrel. XCS and GALE : a comparative study of two Learning Classifer Systems with six other learning algorithms on classification tasks. In P.-L. Lanzi, W. Stolzmann, and S. W. Wilson, editors, *Proceedings of the fourth international workshop on Learning Classifer Systems*, pages 337–341, 2001.

[3] M. V. Butz. *Anticipatory Learning Classifier Systems*. Kluwer Academic Publishers, Boston, MA, 2002.

[4] N. Chomsky. Context-free grammars and pushdown storage. Quarterly Progress Report 65, MIT Research Laboratory in Electronics, 1962.

[5] N. Chomsky and G. Miller. *Handbook of Mathematical Psychology 2*, chapter Introduction to the Formal Analysis of Natural Languages, pages 269–321. Wiley and Sons, New York, 1963.

[6] D. Cliff and S. Ross. Adding memory to ZCS. *Adaptive Behavior*, 3(2):101–150, 1994.

[7] 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.

[8] A. Drogoul and J.-A. Meyer, editors. *Intelligence artificielle située – cerveau, corps et environnement*, Paris, 1999. Hermès. (In French).

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

[10] P. Gérard, J.-A. Meyer, and O. Sigaud. Combining latent learning and dynamic programming in MACS. *European Journal of Operation Research*, 160:614–637, 2005.

[11] P. Gérard, W. Stolzmann, and O. Sigaud. YACS: a new Learning Classifier System using Anticipation. *Journal of Soft Computing*, 6(3-4):216–228, 2002.

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

[13] J. J. Grefenstette. Lamarckian learning in multi-agent environments. In R. Belew and L. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 303–310, San Mateo, CA, 1991. Morgan Kaufmann.

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

[15] E. A. Hansen. *Finite Memory Control of Partially Observable Systems*. PhD thesis, University of Massachusetts, Amherst, MA, 1998.

[16] 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.

[17] J. H. Holland. Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In *Machine Learning, An Artificial Intelligence Approach (volume II)*, pages 593–623. Morgan Kaufmann, 1986.

[18] J. H. Holland and J. S. Reitman. Cognitive Systems based on adaptive algorithms. *Pattern Directed Inference Systems*, 7(2):125–149, 1978.

[19] 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.

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

[21] J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors. *Proceedings of the Third Annual Conference on Genetic Programming*, University of Wisconsin, Madison, Wisconsin, USA, 1998. Morgan Kaufmann.

[22] J. R. Koza, F. H. Bennett, D. Andre, and M. A. Keane. Automated Design of Both the Topology and Sizing of Analog Electrical Circuits Using Genetic Programming. In J. S. Gero and F. Sudweeks, editors, *Artificial Intelligence in Design'96*, pages 151–170, 1996.

[23] 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.

[24] S. Landau and S. Picault. ATNoSFERES: a Model for Evolutive Agent Behaviors. In *Proceedings of the AISB'01 Symposium on Adaptive Agents and Multi-Agent Systems*, pages 95–99, 2001.

[25] S. Landau and S. Picault. Stack-Based Gene Expression. Technical report LIP6 2002/011, LIP6, Paris, 2002.

[26] S. Landau, S. Picault, O. Sigaud, and P. Gérard. A comparison between ATNoSFERES and XCSM. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 926–933, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[27] S. Landau, S. Picault, O. Sigaud, and P. Gérard. Further Comparison between ATNoSFERES and XCSM. In Stolzmann et al. [53], pages 99–117.

[28] S. Landau, O. Sigaud, S. Picault, and P. Gérard. An Experimental Comparison between ATNoSFERES and ACS. In W. Stolzmann et al., editors, *IWLCS-03. Proceedings of the Sixth International Workshop on Learning Classifier Systems*, LNAI, Chicago, july 2003. Springer.

[29] P.-L. Lanzi. An Analysis of the Memory Mechanism of XCSM. In Koza et al. [21].

[30] P.-L. Lanzi. Learning Classifier Systems from a Reinforcement Learning Perspective. *Journal of Soft Computing*, 6(3-4):162–170, 2002.

[31] P.-L. Lanzi and R. L. Riolo. A roadmap to the last decade of Learning Classifier Systems research (from 1989 to 1999). In P.-L. Lanzi, W. Stolzmann, and S. W. Wilson, editors, *Learning Classifier Systems: from Foundations to Applications*, pages 33–62. Springer-Verlag, Heidelberg, 2000.

[32] P.-L. Lanzi, W. Stolzmann, and S. W. Wilson, editors. *Learning Classifier Systems. From Foundations to Applications*, volume 1813 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, 2000.

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

[34] L.-J. Lin and T. M. Mitchell. Memory approaches to reinforcement learning in non-markovian domains. Technical Report CMU-CS-92-138, Carnegie Mellon University, School of Computer Science, 1992.

[35] A. Lindenmayer. Mathematical Models for Cellular Interaction in Development, parts I and II. *Journal of theoretical biology*, 18:280–315, 1968.

[36] S. Luke and L. Spector. Evolving Graphs and Networks with Edge Encoding: Preliminary Report. In Koza [23], pages 117–124.

[37] R. A. McCallum. *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, University of Rochester, Rochester, NY, 1995.

[38] M. Métivier and C. Lattaud. Anticipatory Classifier System using Behavioral Sequences in Non-Markov Environments. In Stolzmann et al. [53], pages 143–163.

[39] N. Meuleau, L. Peshkin, K.-E. Kim, and L. P. Kaelbling. Learning finite-state controllers for partially observable environments. In *Fifteenth Conference on Uncertainty in Artificial Intelligence. AAAI*, pages 427–436, 1999.

[40] D. J. Montana. Strongly Typed Genetic Programming. In *Evolutionary Computation*, volume 3, pages 199–230. 1995.

[41] C. H. Moore and G. C. Leach. FORTH - A Language for Interactive Computing. internal publication, Mohasco Industries, Amsterdam NY, 1970.

[42] 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.

[43] R. Poli and W. B. Langdon. Schema Theory for Genetic Programming with One-point Crossover and Point Mutation. *Evolutionary Computation Journal*, 6(3):231–252, 1998.

[44] G. G. Robertson and R. L. Riolo. A tale of two classifier systems. *Machine Learning*, 3:139–159, 1988.

[45] C. Ryan and M. O'Neill. Grammatical evolution: A steady state approach. In J. R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1998 Conference*, University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998. Stanford University Bookstore.

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

[47] O. Sigaud and P. Gérard. Contribution au problème de la sélection de l'action en environnement partiellement observable. In Drogoul and Meyer [8], pages 129–146. (In French).

[48] R. E. Smith. Memory exploitation in learning classifier systems. *Evolutionary Computation*, 2(3):199–220, 1994.

[49] S. F. Smith. *A Learning System based on Genetic Adaptive Algorithms*. PhD thesis, Department of Computer Science, University of Pittsburg, Amherst, MA, 1980.

[50] W. Stolzmann. Anticipatory Classifier Systems. In Koza et al. [21], pages 658–664.

[51] W. Stolzmann. Latent Learning in Khepera Robots with Anticipatory Classifier Systems. In Wu [64], pages 290–297.

[52] W. Stolzmann. An introduction to Anticipatory Classifier Systems. In P.-L. Lanzi, W. Stolzmann, and S. W. Wilson, editors, *Learning Classifier Systems: from Foundations to Applications*, pages 175–194. Springer-Verlag, Heidelberg, 2000.

[53] W. Stolzmann, P.-L. Lanzi, and S. W. Wilson, editors. *Proceedings of the International Workshop on Learning Classifier Systems (IWLCS'02)*, LNAI 2661, Granada, september 2002. Springer-Verlag.

[54] R. Sun and C. Sessions. Multi-agent reinforcement learning with bidding for segmenting action sequences. In J.-A. Meyer, S. W. Wilson, A. Berthoz, H. Roitblat, and D. Floreano, editors, *From Animals to Animats 6: Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior*, pages 317–324, Paris, 2000. MIT Press.

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

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

[57] A. Tomlinson and L. Bull. A zeroth level corporate classifier system. In Lanzi et al. [32], pages 306–313.

[58] A. Tomlinson and L. Bull. An accuracy-based corporate classifier system. *Journal of Soft Computing*, 6(3–4):200–215, 2002.

[59] M. Wiering and J. Schmidhuber. HQ-Learning. *Adaptive Behavior*, 6(2):219–246, 1997.

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

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

[62] S. W. Wilson and D. E. Goldberg. A critical review of Classifier Systems. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 244–255, Los Altos, California, 1989. Morgan Kaufmann.

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

[64] A. S. Wu, editor. *Proceedings of the 1999 Genetic and Evolutionary Computation Conference (GECCO'99)*, 1999.

[65] X. Yao. Evolving Artificial Neural Networks. *Proceedings of the IEEE*, 87:1423–1447, 1999.