YACS : Combining Dynamic Programming with Generalization in Classifier Systems

Pierre Gérard[1,2] and Olivier Sigaud[1]

[1] Dassault Aviation, DGT/DPR/DESA
78, Quai Marcel Dassault, 92552 St-Cloud Cedex
[2] AnimatLab (LIP6), 8, rue du capitaine Scott, 75015 PARIS
pierre.gerard@lip6.fr, olivier.sigaud@dassault-aviation.fr

Abstract. This paper describes our work on the use of anticipation in Learning Classifier Systems (LCS) applied to Markov problems. We present YACS[1], a new kind of Anticipatory Classifier System. It calls upon classifiers with a [$Condition$], an [$Action$] and an [$Effect$] part.
As in the traditional LCS framework, the classifier discovery process relies on a selection and a creation mechanism. As in ACS, the selection in YACS relies on the quality of the anticipation. Therefore, YACS looks for classifiers which anticipate well rather than for classifiers which propose an optimal action. The creation mechanism does not rely on classical genetic operators but on a specialization operator, which is explicitly driven by experience. Likewise, the action qualities of the classifiers are not computed by a classical bucket-brigade algorithm, but by a variety of the value iteration algorithm that takes advantage of the effect part of the classifiers.
This paper presents the latent learning process of YACS. The description of the reinforcement learning process is focussed on the problem induced by the joint use of generalization and dynamic programming methods.

1 Introduction

Our work takes place in the *reinforcement learning* framework. We model an agent which acts on this environment and receives a reward and a new perception. More precisely, we use the Learning Classifier Systems (LCS) framework whose principles have been set down by [Holland et al., 1986], [Goldberg, 1989] and [Booker et al., 1989].

More recent achievements in this framework are due to [Wilson, 1994] and [Wilson, 1995], [Dorigo, 1994], [Stolzmann, 1998] and [Lanzi, 2000] among others. Most of these research efforts deal with Markov problems, *i.e.* problems in which the distribution of probability for getting a perception only depends on the previous perception and action. The system we present here is designed to solve such problems, although we envision extending our work to non-Markov problems in the future, as [Cliff and Ross, 1994] and [Lanzi, 1998] do.

In this framework, our basic assumptions are the following:

[1] YACS stands for "Yet Another Classifier System"

- rather than generating new classifiers with random genetic operators and evaluating them afterwards, we drive the classifier discovery process by experience, slightly improving what [Dorigo, 1994] did;
- rather than using a plain reinforcement learning process, the agent performs *latent learning* [Riolo, 1991] to use its anticipation capabilities. This latent learning process can take place even if no reward is given by the environment. The joint use of latent learning and dynamic programming algorithms speeds up the convergence towards an optimal behavior once reward sources are identified. It has already been exploited in DynaQ+ [Sutton and Barto, 1998];
- we want to reduce the number of classifiers as much as possible. So, we never have two classifiers such that one is strictly more general than another.

In the next section, we present the components of YACS. As both our system and Stolzmann's ACS[2] [Stolzmann, 1998] deal with classifiers with an effect part, we will highlight how the learning process detailed in section 3 differs in both systems. In this section, we pay a particular attention to the joint use of generalization and dynamic programming. In section 4, we present the preliminary results obtained on a very simple application.

2 Features of the system

The system we designed uses a set of different classifiers[3]. Each *classifier* is a set of ordered *messages*. Each message is a set of ordered *tokens*. All classifiers share the same structure and message lengths.

Tokens may take discrete values in a range $[0,\ NbPossibleValues - 1]$, in which case they are specialized token, or they may take a # value. We have two kinds of tokens: *action tokens* and *perception tokens*. Action tokens are symbols representing elementary actions, for example the activation level of a particular engine in a robot. Perception tokens are symbols representing elementary perceptions, for example the value given by a particular sensor.

The system deals with two kinds of messages: *action* and *perception* messages, containing respectively action and perception tokens. The range of all tokens at the same place are the same for all messages of the same kind.

Two tokens are said to *match* if at least one is a # token, or if both have the same value. Two messages of the same kind are said to *match* if all their tokens match the corresponding token in the other message.

If every token of a perception message is less general or equal to the corresponding tokens of a second message, and if at least two corresponding tokens are different, the first message is *more specialized* than the second one.

As in ACS [Stolzmann, 1998], a classifier is composed of three parts: the $[Condition]$ and the $[Effect]$ parts are perception messages, the $[Action]$ part is an action message.

[2] Anticipatory Classifier System

[3] A classifier is never added to the classifier set if another one in the set has the same $[Condition]$, $[Action]$ and $[Effect]$ parts.

The tokens of an $[Effect]$ part act as a filter: a # in the $[Effect]$ part is a *don't change* token and means "the elementary perception represented by the token will remain unchanged at the next time step if the classifier is fired"; any other value is interpreted straight-forwardly. When the condition of a classifier matches a perception, we use the *passthrough* operator to predict the next perception if the action of the classifier is chosen: The *passthrough* operator works on perception tokens [4] as follows :

$$passthrough(t_p, t_e) = \begin{cases} t_p \; if \; t_e = \# \\ t_e \; otherwise \end{cases}$$

Applying the *passthrough* operator on perception messages consists in applying the operator on their tokens.

Let C be a classifier with a $[Condition]$ part matching the $[Perception]$ message. Then C anticipates the perception $[Perception].passthrough(C.[Effect])$[5] when the action $C.[Action]$ is performed just after $[Perception]$ occurs.

We also use the reverse operator of *passthrough* - the *difference* operator - which works on perception tokens [6] as follows:

$$difference(t_2, t_1) = \begin{cases} \# \; if \; t_2 = t_1 \\ t_2 \; otherwise \end{cases}$$

Given two successive perceptions, this operator allows to compute what an $[Effect]$ part should have been to predict correctly the second one if given the first.

An immediate reward estimate R is associated to each classifier. R reflects the expected *immediate reward* if the classifier is fired[7]. It is estimated from direct experience. Dynamic programming algorithms like value iteration take advantage of immediate reward estimates and information provided by the $[Effect]$ part to compute an optimal policy.

Each classifier also keeps a trace T of *good* and *bad* markers memorizing past anticipation mistakes and successes. The length of this trace is bounded by a fixed memory size m.

When two classifiers share the same $[Action]$ part and if a $[Condition]$ part is more specialized than the other one, the first classifier is more specialized than the other. We do not consider the $[Effect]$ part of a classifier to determine its specialization level because a # in the $[Effect]$ is a *don't change* token and not a *don't care* token as in a $[Condition]$ part.

[4] t_p is the token of a $[Perception]$ and t_e is the token of an $[Effect]$ part.

[5] We use the dot symbol (.) to identify a part of a composed item. For example, $C.[Condition]$ means "the $[Condition]$ part of the classifier C"; $C.R$ means "the R estimate of the classifier C" (its immediate reward estimate); $t.S$ means "the S estimate of the token t". Hence, we always use the "$\times$" symbol for multiplication.

[6] t_2 is a token of a perception occuring just after the perception containing t_1

[7] The default R is 0.

For every classifier, each general token of the $[Condition]$ part keeps an *expected improvement by specialization* estimate S[8] which helps to drive the specialization process (see section 3.3)

The system also uses a set of every perception encountered during the lifetime of the agent. This set only contains one instance of each perception. It is not ordered.

3 The Algorithm

Like [Stolzmann, 1998] and [Witkowski, 1999], we have an $[Effect]$ part in the classifiers. The classifier discovery process builds a set of classifiers which anticipate well rather than classifiers which act optimally. This knowledge about state transitions allows the system to plan its actions or to use a variety of the value iteration algorithm. It becomes able to adjust his policy very fast when a new reward source is discovered.

As in many other works, we divide the life-time of the agent into discrete time steps. During a time step, the agent acts as follows:

1. It gets a reward and a perception from the environment;
2. It learns about the dynamics of its environment and the optimality of actions;
3. It selects an action according to what it learned;
4. It acts correspondingly in the environment.

The *latent learning* process is in charge of discovering adequate classifiers which model the dynamics of the environment. In ACS [Stolzmann, 1998] the ALP[9] modifies at the same time $[Condition]$ and $[Effect]$ parts in order to reflect the changes in the dynamic of the environment. In YACS the $[Effect]$ part alone provides all the information about changes in the environment. A specialized token in the $[Effect]$ part always indicate a change in the environment, regardless of the corresponding token in the $[Condition]$ part. In YACS, the latent learning process can be divided into two simple and separate processes:

- adjusting the $[Effect]$ parts (section 3.2);
- discovering relevant $[Condition]$ parts (section 3.3).

A $[Condition]$ part may specify a state of the environment even if it is not fully specialized. Furthermore, the minimal set of tokens necessary to specify a state does not necessarily correspond to the changing tokens of the perceptions when the classifier is fired. However, in ACS [Stolzmann, 1998], the ALP always specializes the $[Condition]$ part and the $[Effect]$ part at the same time, and only when the corresponding token in the perceptions is changing. As a result (see [Butz et al., 2000a]), some $[Condition]$ parts may be over-specialized. Splitting the anticipatory learning process into two separate processes helps to overcome this problem.

[8] The default S is 0.5.

[9] Anticipation Learning Process

The *reinforcement learning* process takes advantage of the model of the dynamics of the environment computed by the latent learning process. In section 3.5 we present a problem induced when we jointly want to take advantage of generalization and use dynamic programming algorithms like value iteration, and we propose a solution.

The *action selection* uses a winner-take-all strategy (see section 3.6).

3.1 Getting a reward and a new perception

When the system comes to time step t, it gets from the environment the new perception $[Perception]_t$ and the reward value $Reward_t$ resulting from the last selected action.

If the new perception is not present in the set of encountered perceptions, it is added. But, even if we only keep one instance of every encountered perception, this set can become significantly large if the agent gets a lot of a elementary perceptions. In this case, it could be worth taking advantage of generalization to reduce the size of the set.

As the system learns from one step temporal differences, the latent learning process relies on a memory of the last perception. So the system stores the last perception $[Perception]_{t-1}$ and forgets $[Perception]_{t-2}$.

If the $[Condition]$ part of no classifier matches it for a particular $[Action]$ message, one is added to the classifier set. The $[Effect]$ part of the new classifier is set to $\mathit{difference}([Perception]_t, [Perception]_{t-1})$. The $[Condition]$ part is such that:

- it matches $[Perception]_t$;
- it is neither more general nor more specialized than any $[Condition]$ part of the $[Condition]$ part of any other classifier with the same $[Action]$ part.
- it is as general as possible, considering the previous constraints.

These conditions allow to add maximally general classifiers without introducing redundancies with already specialized ones.

3.2 Learning to anticipate

This process is the part of the latent learning process which is in charge of discovering accurate $[Effect]$ parts. When the system learns to anticipate, it may create some new classifiers, with suitable $[Effect]$ parts straight-forwardly settled according to experience. As [Witkowski, 1999] does, the algorithm does not only evaluate the classifiers which have been fired, but also takes advantage of experience to evaluate all the classifiers which could have been fired.

At each time step, the $[DesiredEffect]$ message is computed according to the formula:

$$[DesiredEffect] \leftarrow difference([Perception]_t, [Perception]_{t-1})$$

This message corresponds to the anticipation of a classifier that would have anticipated well at the last time step and whose selection (see section 3.6) would have driven the system to act as it actually did. Classifiers can be involved in the anticipation learning process even if they were not actually selected. These classifiers C are such that $C.[Condition]$ matches $[Perception]_{t-1}$ and $C.[Action]$ matches $[Action]_{t-1}$.

Let us consider such a classifier C.

- If $C.[Prediction]$ equals $[Desired\,Anticipation]$, the classifier would have anticipated well, and we add a *good* marker to its trace T of anticipation mistakes and successes.
- In either case, the classifier C would have made an anticipation mistake and we add a a *bad* marker to its trace. Moreover, if no classifier did anticipate correctly, we add a new classifier which anticipates well. As in [Stolzmann, 1998], the anticipation is *covered* and we add a new classifier which is the same as the initial one but its $[Effect]$ part which is set to the $[Desired\,Effect]$. Its trace T only contains a single *good* marker.

Our *anticipation covering* mechanism creates classifiers with relevant $[Effect]$ parts by taking advantage of direct experience rather than genetic algorithms (GA). It differs from the ACS anticipation covering since it does only modify the $[Effect]$ part. The anticipation covering mechanism allows to learn straightforwardly when an action does not change the perceptions of the system, without requiering a dedicated mechanism like the *specification of unchanging components* in ACS [Stolzmann, 1999].

3.3 Learning relevant conditions

In section 3.2 we have explained how, while learning to anticipate, the system adds new classifiers to adjust $[Effect]$ parts. This section explains how $[Condition]$ parts are adjusted.

The MutSpec operator The classifier discovery problem is usually solved by a GA using a creation process driven by mutation and crossover on classifiers selected by their quality. operators. These blind operators do not explicitly take advantage of the experience of the agent. Dorigo's $MutSpec$ operator [Dorigo, 1994] improves the classifier discovery process by driving the specialization of the classifiers according to its experience. Our purpose is to build a classifier system without mutation nor crossover operators. Like McCallum's U-Tree algorithm [McCallum, 1996], our system starts without making any distinction between world states, and incrementally introduces experience driven specializations in $[Condition]$ parts.

At the first time step, the classifier set contains one general classifier for each possible action. The $[Condition]$ and $[Effect]$ parts only contain # tokens.

At each time step, we add *good* and *bad* markers in the trace T of anticipation mistakes and sucesses (see section 3.2) of several classifiers. This trace works as a FIFO list with a finite length m. When the trace is full -*i.e.* its size equals m-, we assume that the anticipation accuracy the classifier has been checked enough and

– if the trace of the classifier only contains *good* markers, it always anticipated well and it does not need to be more specialized;
– if the trace of the classifier only contains *bad* markers, and if it is not the only one matching a particular perception of the encountered perception set for a particular action, it is discarded;
– if the trace of the classifier contains *good* and *bad* makers, it sometimes anticipates well and sometimes not. The classifier *oscillates* and its [$Condition$] part needs to be specialized.

The specialization process is designed to discover relevant [$Condition$] parts, it uses the $MutSpec$ operator introduced by [Dorigo, 1994]. The $MutSpec$ operator selects a joker token of the classifier and produces one new classifier for each possible specialized value of the selected token. The original classifier is discarded.

For instance, when the first token is selected, and assuming that it only may take two values (`0` or `1`) the classifier `[#|#|#|#] [0] [#|#|#|#]` produces two classifiers:

– `[0|#|#|#] [0] [#|#|#|#]`;
– `[1|#|#|#] [0] [#|#|#|#]`.

So, if the [$Condition$] part of the original classifier was matching several states of the environment, each resulting [$Condition$] part will match a subset of these states. We want YACS to be able to choose the token two specialize in such a way that the two resulting subsets have an equal cardinality, in order to reduce the number of specializations and thus the over-specialization.

The expected improvement by specialization estimate Choosing at random the token to specialize as in Dorigo's original work would lead to an over-specialization of the [$Condition$] parts and thus to a sub-optimal number of classifiers. We improve this selection by using the *expected improvement by specialization* estimate S associated to each general token of each [$Condition$] part. This value estimates how much the specialization of the token would help to split the state set covered by the [$Condition$] part into several sub-sets of equal cardinality.

Let us consider a classifier which tries to anticipate the consequences of an action in several situations. If the value of a particular feature of the perception when the classifier anticipates well is always different from the value when it makes anticipation mistakes, then the [$Condition$] part must be specialized according to this particular feature, and the estimate S will get a high value.

In order to compute the estimates S, each classifier memorizes the perception preceeding the last anticipation mistake [$BadPerception$] and the last perception preceeding the last anticipation success [$GoodPerception$]. Each time the [$Action$] part of a classifier matches [$Action_{t-1}$] and its [$Condition$] part matches [$Perception_{t-1}$], the [$Perception_t$] allows to check the accuracy of the [$Effect$] part.

- If the $[Effect]$ part is *correct*, for each feature of the environment :
  - if a particular token of $[BadPerception]$ equals the corresponding feature of $[Perception_{t-1}]$, then the corresponding estimate S is *decreased* using a Widrow-Hoff delta rule;
  - if a particular token of $[BadPerception]$ differs from the corresponding feature of $[Perception_{t-1}]$, then the corresponding estimate S is *increased* using a Widrow-Hoff delta rule;
- If the $[Effect]$ part is *incorrect*, for each feature of the environment :
  - if a particular token of $[GoodPerception]$ equals the corresponding feature of $[Perception_{t-1}]$, then the corresponding estimate S is *decreased* using a Widrow-Hoff delta rule;
  - if a particular token of $[GoodPerception]$ differs from the corresponding feature of $[Perception_{t-1}]$, then the corresponding estimate S is *increased* using a Widrow-Hoff delta rule;

This process allows every classifier to identify which general token should be first specialized in order to improve the relevance of the $[Condition]$ part.

3.4 The specialization process

The *expected improvement by specialization estimates* allow the classifier specialization mechanism to be driven by experience and are used in the $[Condition]$ specialization process. This estimate allows to drive specialization without managing $[Perception]$ marks as in ACS [Butz et al., 2000b].

When a classifier sometimes anticipate well and sometimes not - *ie* when its anticipation trace contains *good* and *bad* markers - it oscillates and its $[Condition]$ part needs to be specialized. If such a classifier is oscillating, thanks to the *anticipation covering* mechanism, the classifier set contains at least one other classifier with the same $[Condition]$ and $[Action]$ parts and with a different $[Effect]$ part.

The specialization process is very careful : YACS waits that every classifier with the same $[Condition]$ and $[Action]$ parts has been identified as an *oscillating* classifier, and that its anticipation trace is full. At this point, these classifiers elect together the feature to specialize. The estimates S corresponding to each feature of the environment are summed among the classifiers, and the feature with the highest sum is chosen to get specialized. The $MutSpec$ operator is then applied to every classifier.

$MutSpec$ may create classifiers that will never be used nor evaluated because their $[Condition]$ part matches no possible perception. To avoid such classifiers, we remove every classifier which does not match any perception in the set of already encountered perceptions.

In this section, we described how YACS performs latent learning in two separate processes :

- learning accurate $[Effect]$ parts by temporal difference learning;
- carefuly specializing the $[Condition]$ parts.

The set of classifiers discovered by the latent learning process is a model of the dynamics of the environment which provides information about the state transitions. It takes advantage of the generalization to discover regularities and keep the model small.

3.5 Learning to act

In this section, we will describe how YACS takes advantage of the model of the environment to speed up the reinforcement learning process. We first introduce the value iteration algorithm and the way YACS identifies the reward sources. Then we explain how the use of generalization forbids to compute one single quality for each classifier. We finally present two strategies that may be used and our reasons to propose a different one.

Value Iteration To back-propagate the reward, we use a simplified variety of value iteration : a dynamic programming algorithm which solves the Bellmann equations [Bellman, 1957] by iteratively refining qualities for (state, action) pairs by using the formula:

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s')V(s') \quad (1)$$

where

$$V(s) = max_a Q(s,a) \quad (2)$$

$Q(s,a)$ is the quality of action a in state s. γ is the temporal discount factor. $V(s)$ is the desirability value of the state s. $R(s,a)$ is the immediate expected reward when the agent performs action a in state s. T(s,a,s') is the probability for reaching state s' when performing action a in state s.

As our system is designed to deal with deterministic environment, we do not use the transition probabilities and replace the expected future cumulative reward $\sum_{s'} T(s,a,s')V(s')$ by $max_{s'} V(s')$ where s' is a state anticipated when the system performs the action a in state s.

Learning about immediate reward The latent learning provides the information about state transitions. In order to use a dynamic programming algorithm, the system computes the immediate expected rewards corresponding to $R(s,a)$. At each time step, the immediate reward estimates R of every classifier such that $C.[Condition]$ matches $[Perception]_{t-1}$ and $C.[Action]$ matches $[Action]_{t-1}$ are updated according to the formula:

$$R \leftarrow (1-\beta) \times R + \beta \times CurrentReward$$

Here again, the algorithm updates values of classifiers which have not been actually fired.

The generalization problem If YACS would not use generalization, it would be easy to compute a single quality of action for each classifier by using the formula:

$$C.Q \leftarrow C.R + \gamma \times max_{C'} C'.Q$$

In classifier systems without an $[Effect]$ part, a classifier is kept when it helps maximizing reward on the long run, or when it is able to predict the reward. When we use classifiers with an $[Effect]$ part, the decision to keep or remove a classifier ony relies on its ability to predict the next perceptions. It does not take the reward into account.

This way of considering the fitness of a classifier gives rise to a new way of considering generalization. A classifier is too general when a joker token prevents the anticipation to be accurate, regardless of the payoff. It is too specialized if its anticipation ability would remain accurate if some joker were added in its $[Condition]$ part, regardless of the payoff.

For example, let us consider a classifier which could be interpreted in a maze environment as "when the agent percieves a wall on the north, if it tries to move north, nothing will change in its perceptions : it will remain in the same square". Such a classifier is accurate, since it would not anticipate better if a joker token of its $[Condition]$ part was specialized. It is kept and will not be further specialized. The problem is that such an accurate classifier is not too general with respect to anticipation, but it introduces a kind of perceptual aliasing since it matches in several different situations.

Because of the discount factor (γ in formula 1), the qualities associated to the states which are close to the goal are much higher than the qualities associated to the states which are far from the goal. So one can not compute a single quality for a classifier whose $[Condition]$ part matches a several perception recieved at different distances from the goal.

So, as the generalization helps to discover regularities in the environment instead of simply providing a kind of selective attention, some classifier may be fired in several states and thus it becomes impossible to compute a single quality for each general classifier.

Possible Solutions In order to avoid this problem, one solution could be to introduce the detection of over-general classifiers with respect to payoff and to specialize such classifiers. But this solution is not coherent with our approach: the latent learning process, which provides a model of the dynamics of the environment, must take place even in the absence of rewards. We want to model dynamics of the environment with as few classifiers as possible in order to reduce computation time. Thus the model must take advantage of every regularities of the environment. So, classifiers which do not anticipate well should be further specialized.

Another solution is the use of a lookahead planning algorithm rather than a variety of the value iteration algorithm. At each time step, the system would have to build a plan starting with the current perception. [Butz and Stolzmann, 1999] proposes an enhancement to ACS which uses explicit goals to perform bidirectionnal planning, but comes out of the reinforcemnt learning framework. A general classifier used at different levels of the planning process is interpreted in different contexts and this should solve the problem of perceptual aliasing introduced by over-general classifiers with respect to payoff but accurate with respect to anticipation. But as a result, the system may suffer from a lack of reactivity.

Storing values for every already encountered perception The solution we propose is to associate a desirability value to each specialized $[Perception]$ in the already encountered perception list. YACS stores values corresponding to $V(s)$ in the formula 2 - one for each $[Perception]$ - instead of storing qualities corresponding to $Q(s,a)$ - one for each $[Perception]/[Action]$ pair - as it is the case for Q-Learning [Watkins, 1989] and DynaQ+ [Sutton and Barto, 1998].

To compute the quality associated to a $[Perception]$, we first identify all the classifiers C such that $C.[Condition]$ matches $[Perception]$. If such a classifier was fired, the immediate reward would be $C.R$ and the expected cumulative reward would be the value of the perception anticipated by the classifier. In a more formal way, the quality Q associated to a $[Perception]$ can be updated by using the formula :

$$[Perception].Q \leftarrow max_C C.R + \gamma \times [Perception].passthrough(C.[Effect]).Q$$

where $C.[Condition]$ matches $[Perception]$

So the system computes values for each perception and takes advantage of the model provided by the latent learning process to update these values without actually experiencing every transition. When the classifier system anticipates well in every situation, the agent may adapt its behavior quickly to new reward sources.

3.6 Selecting an action

When the system gets a $[Perception]$ from the environment, it selects all the classifiers C whose $[Condition]$ part matches it. These classifiers anticipate the following perception by computing $[Perception].passthrough(C.[Effect])$. Then they compute a quality by adding the immediate reward estimate $C.R$ and the discounted expected reward which is the value of the following perception mutiplied by the discount factor γ. The selected action is the action of the classifier with the highest quality.

4 Preliminary results

4.1 A simple maze problem

We use the maze problem described in figure 1 in order to evaluate our algorithm. This maze illustrates a state transition diagram with nine states and four possible transitions starting from each state.

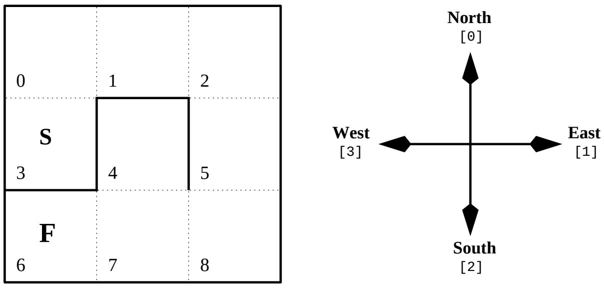

Fig. 1. A simple maze problem

The agent is always situated in a square. It can perceive the abscence or presence of walls in each cardinal direction (N, E, S and W). For instance, the agent percieves `[1|0|0|1]` in square 0 and `[1|1|0|0]` in square 2. The agent is given four possible actions: to move in any of the four cardinal directions.

An experience is divided into several trials. For each trial, the agent starts in square S. At each time step, it can make every possible action, including dumb actions like hitting a wall. When it comes to square 6, it gets a reward of 1 and it is given the opportunity to learn about the state transition. Then a new trial starts in square 3.

This goal can be reached optimaly in 9 successive time steps.

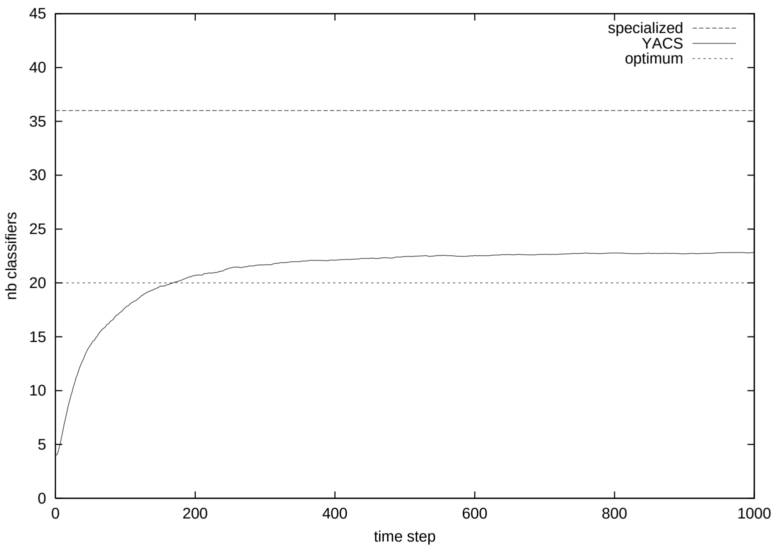

Fig. 2. Average evolution of the number of classifiers

4.2 Presentation of the results

The results we present here have been obtained with a learning rate β of 0.1 and a memory size m of 3.

Latent Learning We let the system move in the maze for 1000 time steps without any reward. During this time, it moves randomly and performs latent learning to model the dynamics of the environment. The figure 2 shows the evolution of the average number of classifiers over 1000 experiments.

The figure 3 shows the evolution of the number of classifiers on a representative single experiment.

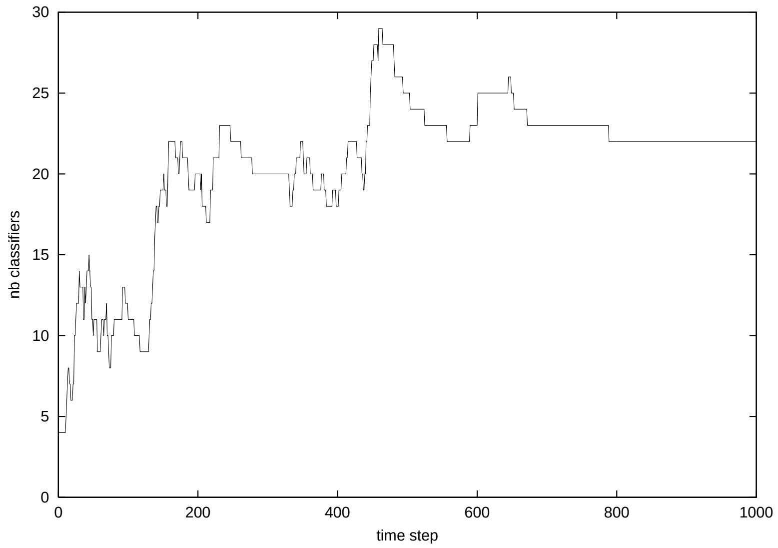

Fig. 3. Evolution of the number of classifiers for one experiment

The minimal number of classifiers to accurately model the whole environment is 20. The number of fully specialized classifiers needed to model the dynamics of the environment is 36. A Q-table would consider 64 $(state, action)$ pairs. The number of classifiers produced by YACS converges arround 23.

YACS finds very accurately regularities like "when there is a wall on the north, moving north does not produce any changes in the perceptions", but sometimes specializes the classifiers in a non optimal way in early time steps, because of some partially representative samples of experience. As a result, the number of classifiers may be sub-optimal.

Reinforcement Learning After these 1000 time steps of exploration, the system is given a reward when it reaches the goal. The figure 4 shows the average number of time steps the system needed to achieve successsive trials with reward, over 1000 experiments. The figure 5 shows the number of time steps the system needed to achieve successive trials for the same single experiment as for figure 3. The number of time steps of the first trial is high because the system has not already been rewarded yet and thus has no way to find the optimal way to the goal.

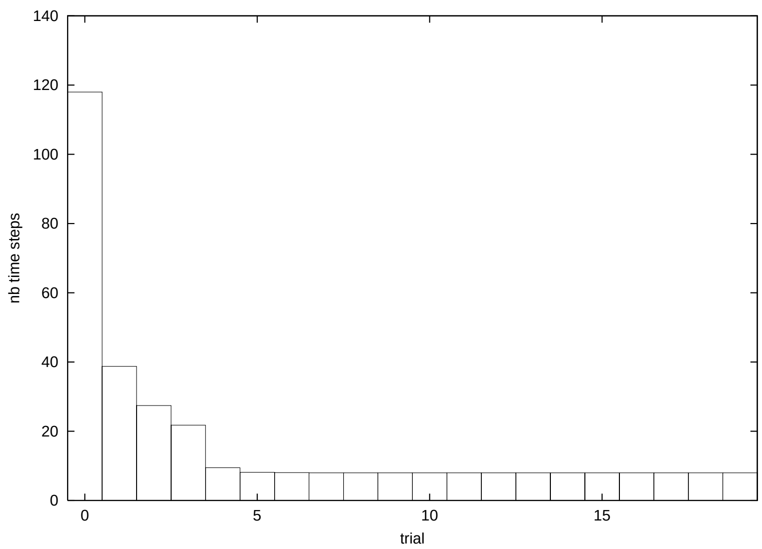

Fig. 4. Average number of time step to reach the goal in successive trials

After this first trial, the system has identified a reward source and takes advantage of its model of the environment to quickly find the optimal path to the goal. The agent does not immediately behave optimaly because the system does not perform a complete value iteration each time step. In order to get a good reactivity/planning tradeoff, only one step of value iteration is performed each time step. As a result, the agent needs several time steps to adjust its policy.

In the figure 4, the average convergence looks slower because the average values take into account several experiments like the one shown in figure 6. In such experiments, the model of the environment was not perfectly accurate after the 1000 time steps of exploration. The system needed some more sub-optimal trials to adjust the model.

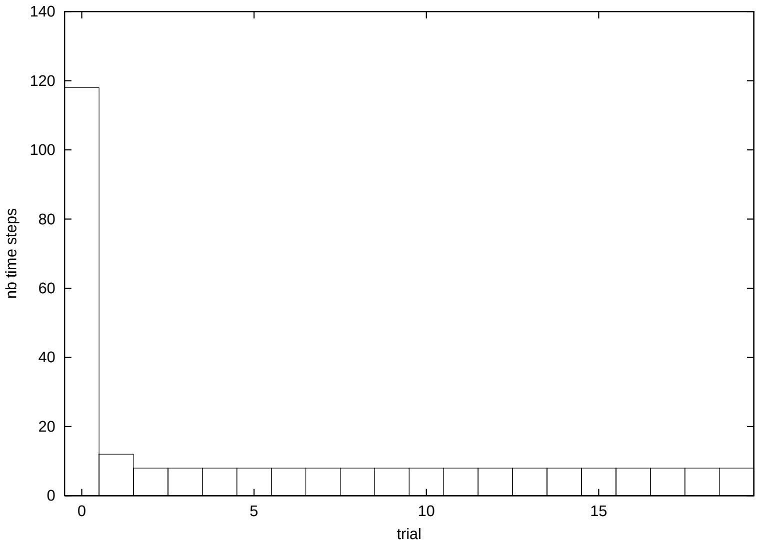

Fig. 5. Number of time steps to reach the goal in successive trials for one experiment.YACS was able to build an accurate model of the environment within the 1000 exploration time steps

5 Discussion

The results presented in section 4 are encouraging but YACS still suffers two major drawbacks.

5.1 The exploration/exploitation tradeoff

In YACS, the reinforcement learning process works fine only if the model provided by the latent learning is complete and accurate. If the classifiers are not specialized enough, the system may "imagine" transitions which are incorrect in the actual environment, and which may prevent the system to find the way to the goal. Over 1000 experiments, the system was 17 times unable to reach the goal within 1000 time step after the exploration steps and the first trial leading to the identification of the reward source. These experiments are not taken into account in the figures 2 and 4. This drawback forbids YACS to simultaneously build the model while exploiting safely a partial model of the environment.

5.2 The need for a generalization mechanism

Having an efficient generalization mechanism is important in the LCS approach since generalization is one of the most original features of LCS with respect to basic reinforcement learning algorithms. Even if YACS is able to stop the

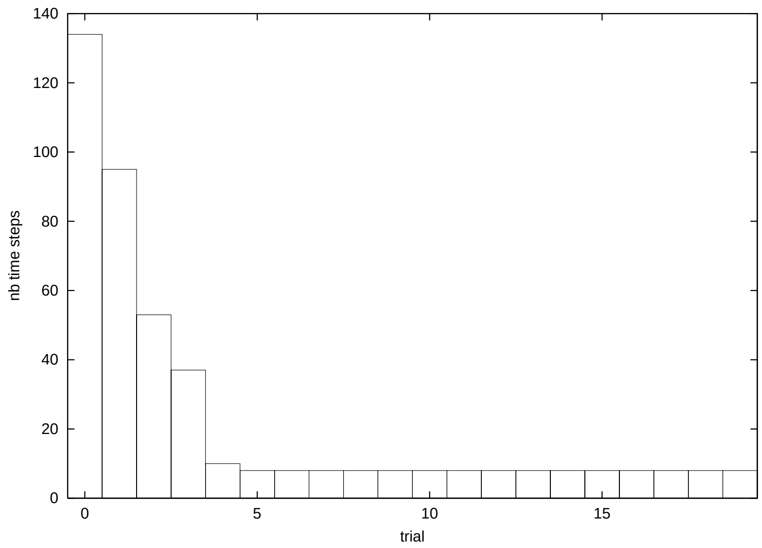

Fig. 6. Number of time steps to reach the goal in successive trials for one experiment. YACS was not able to build an accurate model of the environment within the 1000 exploration time steps

specialization process when there is no more need for more specialized classifiers, it still lacks a specific generalization process. This leads our system to sub-optimality problems:

– choosing to specialize a bad token in early time steps may lead to a sub-optimal number of classifiers;
– in a changing environment, the adaptation of YACS relies on the condition covering mechanism, and leads to over-specialized classifiers and a sub-optimal number of classifiers.

A generalization mechanism should offer the opportunity for the system to reconsider early choices of tokens to specialize, and to discover reliable and accurate classifiers.

6 Conclusion and future work

Our work takes place in a recent trend on LCS research which focuses on the use of anticipation in order to improve the quality of classifiers and increase the learning speed. As most researchers in this trend [Stolzmann, 1998] and [Witkowski, 1999], we have designed YACS, a system which combines latent learning and reinforcement learning. We have highlighted difficulties due to generalization when we use latent learning through dynamic programming algorithms. We also presented some experimental results which were dealing with

YACS involved in a simple maze environment, but YACS is also used for bigger applications as in [Sigaud, 2000].

But YACS still lacks a generalization process. Recent works like [Lanzi, 1999] and [Butz et al., 2000b] has shown that this generalization concern is shared by other researchers. [Butz et al., 2000b] proposes a solution using the selection pressure of a GA to favor the emergence of reliable and general classifiers in ACS. Since we are reluctant to use the blind search mechanisms of GAs, we will investigate alternative solutions in the short term future.

References

[Bellman, 1957] Bellman, R. E. (1957). *Dynamic Programming.* Princeton University Press, Princeton, NJ.

[Booker et al., 1989] Booker, L., Goldberg, D. E., and Holland, J. H. (1989). Classifier systems and genetic algorithms. *Artificial Intelligence*, 40(1-3):235–282.

[Butz et al., 2000a] Butz, M. V., Goldberg, D. E., and Stolzmann, W. (2000a). Introducing a genetic generalization pressure to the anticipatory classifier system part i: Theoretical approach. In *Proceedings of the 2000 Genetic and Evolutionary Computation Conference (GECCO 2000).*

[Butz et al., 2000b] Butz, M. V., Goldberg, D. E., and Stolzmann, W. (2000b). Investigating generalization in the anticipatory classifier system. In *Proceedings of the Sixth International Conference on Parallel Problem Solving from Nature.*

[Butz and Stolzmann, 1999] Butz, M. V. and Stolzmann, W. (1999). Action-planning in anticipatory classifier sytems. In *Proceedings of the 1999 Genetic and Evolutionary Computation Conference Workshop Program.*

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

[Dorigo, 1994] Dorigo, M. (1994). Genetic and non-genetic operators in ALECSYS. *Evolutionary Computation*, 1(2):151–164.

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

[Holland et al., 1986] Holland, J. H., Holyoak, K. J., Nisbett, R. E., and Thagard, P. R. (1986). *Induction.* MIT Press.

[Lanzi, 1998] Lanzi, P. L. (1998). Adding memory to XCS. In *Proceedings of the IEEE Conference on Evolutionary Computation (ICEC98).* IEEE Press.

[Lanzi, 1999] Lanzi, P. L. (1999). An analysis of generalization in the XCS classifier system. *Evolutionary Computation*, 2(7):125–149.

[Lanzi, 2000] Lanzi, P. L. (2000). Toward optimal performance in classifier systems. *Evolutionary Computation Journal.* in print.

[McCallum, 1996] McCallum, R. A. (1996). Learning to use selective attention and short-term memory. In Maes, P., Mataric, M., Meyer, J.-A., Pollack, J., and Wilson, S. W., (Eds.), *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, pages 315–324, Cambridge, MA. MIT Press.

[Riolo, 1991] Riolo, R. L. (1991). Lookahead planning and latent learning in a classifier system. In Meyer, J.-A. and Wilson, S. W., (Eds.), *From annimals to animats: Proceedings of the First International Conference on Simulation of Adaptative Behavior*, pages 316–326, Cambridge, MA. MIT Press.

[Sigaud, 2000] Sigaud, O. (2000). Using classifier systems as adaptive expert systems for control. In Stolzmann, W., Lanzi, P.-L., and Wilson, S. W., (Eds.), *LNCS : New trends in Classifier Systems*. Springer-Verlag.

[Stolzmann, 1998] Stolzmann, W. (1998). Anticipatory classifier systems. In Koza, J., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D., Garzon, M., Goldberg, D., Iba, H., and Riolo, R., (Eds.), *Genetic Programming*. Morgan Kaufmann Publishers, Inc., San Francisco, CA.

[Stolzmann, 1999] Stolzmann, W. (1999). Latent learning in khepera robots with anticipatory classifier systems. In *Proceedings of the 1999 Genetic and Evolutionary Computation Conference Workshop Program*.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

[Watkins, 1989] Watkins, C. J. (1989). *Learning with delayed rewards*. PhD thesis, Psychology Department, University of Cambridge, England.

[Wilson, 1994] Wilson, S. W. (1994). ZCS, a zeroth level classifier system. *Evolutionary Computation*, 2(1):1–18.

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

[Witkowski, 1999] Witkowski, C. M. (1999). Integrating unsupervised learning, motivation and action selection in an a-life agent. In Floreano, D., Mondada, F., and Nicoud, J.-D., (Eds.), *5th European Conference on Artificial Life (ECAL-99)*, pages 355–364, Lausanne. Springer.