HXCS: Hierarchical Classifier System with Accuracy-Based Fitness

by

Aaron D. Wieland, B.Sc.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University
Ottawa, Ontario
March 28, 2001

Abstract

Classifier systems (CSs) are rule-based systems that use a combination of reinforcement and genetic learning to adapt dynamically to an environment. Performance may be optimal for simple problems, but tends to break down when the solution requires extended action sequences or the environment is non-Markov. Hierarchical systems may improve scalability, by decomposing extended action sequences into shorter sub-sequences, and by supplying the internal state necessary to achieve optimal performance in non-Markov environments.

HXCS is a novel hierarchical CS, based partly on Wilson's hierarchical performance and reinforcement algorithm, and partly on XCS, a CS that bases fitness on accuracy rather than predicted payoff. It is tested on a simple, non-Markov maze. HXCS achieves optimal performance periodically but not consistently, and it is discovered that the optimal policies are dependent on infinitesimal differences between payoff predictions. It is suggested that an appropriate bias would make the algorithm more robust and reliable.

Table of Contents

List of Tables

List of Figures

1. Introduction

Classifier systems (CSs) are rule-based systems that use reinforcement learning and Darwinian evolution to dynamically adapt to their environment. The advantages of CSs over certain other forms of machine learning include their suitability for incremental, real-time adaptation, and a simple rule representation that is easy to modify genetically, yet relatively easy to understand. Like many reinforcement learning systems, however, CSs have limited scalability: in complex environments where extended action sequences are required to obtain a goal, reinforcement is time-consuming and error prone.

Various researchers have suggested that CSs would perform better if they were structured hierarchically, each proposing his own hierarchical architecture. To evaluate these highly varied solutions, it is necessary to establish a list of desirable characteristics. The following traits should be included in the curriculum vitae of the ideal hierarchical CS: (1) capable of planning, (2) learns hierarchical relationships dynamically, (3) depends on local versus global information, (4) requires minimal domain knowledge, (5) robust. In fact, these five attributes may be considered prerequisites of the true goal: scalability.

Specifications for hierarchical CSs include the Hierarchical Chunking Algorithm (HCA), the Anticipatory Classifier System (ACS), HCS, MonaLysa, ALECSYS, and Wilson's hierarchical performance and reinforcement algorithm. All but the last fail at least one of the criteria listed above – and the merits of Wilson's algorithm are uncertain, since it has never been implemented.

For this reason, it was deemed important to implement a hierarchical CS based on Wilson's algorithm. Wilson's proposal was published in 1987, however, and CS research has progressed significantly since then. In fact, Wilson himself has designed a new CS, XCS, that has performed well in certain multi-step environments. Unlike traditional CSs that base the Darwinian fitness of a rule on its predicted payoff, XCS bases fitness on the *accuracy* of the prediction. Therefore, a new hierarchical CS has been implemented, HXCS, based on both Wilson's hierarchical algorithm and XCS. In addition to the incorporation of XCS-based algorithms, HXCS differs from the original hierarchical algorithm in one important respect: rules are grouped into distinct reactive and planning populations, instead of existing in a single homogeneous population.

Although the ultimate goal of hierarchical systems is to improve scalability, the motivation for the experiments described here is simply to prove it is possible for HXCS's performance and reinforcement algorithms to solve a simple problem. The first step is to show that two populations, one with reactive rules, the other with planning rules, can cooperate to solve a problem — one whose optimal policy requires the use of internal state. In other words, the goal is to demonstrate that HXCS can achieve *better* performance than a non-hierarchical CS without any internal state, such as XCS.

Therefore, HXCS was tested in a simple non-Markov maze with two aliasing positions. To disambiguate these two positions, indistinguishable on the basis of sensory input alone, requires an additional bit of internal state; thus, the internal message of each planning rule was a single bit, the minimum needed for optimum policy.

Performance was lacklustre for the initial experiments. It was noted that HXCS was unable to distinguish the two scenarios that involve aliasing positions: (1) starting *at* an aliasing position, and (2) passing *through* such a position. After an extra bit was added to encode this information, performance improved noticeably. The system even managed to achieve optimal performance occasionally. But a close examination of these uncommon optimal solutions led to a disturbing insight: optimal performance depended on an infinitesimal difference between the ostensibly identical predictions of the best reactive rule and best planning rule for one position. Solutions that do not include this feature are vulnerable to useless planning steps. It is suggested that a bias be used to eliminate redundant planning steps and achieve optimal performance reliably. An initial attempt to apply such a bias led to decreased performance; it appears that, while the bias is effective at preserving optimal behaviour for a correct solution, it prevents the system from learning this solution in the first place.

It is concluded that the algorithm for HXCS is basically sound, insofar as it is capable of using internal messages to achieve optimum performance in a non-Markov environment, an improvement over what is possible for a non-hierarchical CS without hidden state. Thus, the modest goal set for these initial experiments was satisfied. But performance will not be reliable until an appropriate bias is devised to promote the desired behaviour. Future work will show whether HXCS is truly scalable; for example, whether hierarchical reinforcement is successful at decomposing long action sequences into shorter, more easily learned sequences. Due to the potential instability caused by the

tight coupling between the reactive and planning levels, further modifications will probably be needed.

Chapter 2 provides an overview of CSs. Chapter 3 explains why it is difficult for a CS to learn long action sequences. XCS, a relatively new CS that bases fitness on accuracy instead of predicted payoff, is described in Chapter 4. Previous hierarchical CSs are summarized in Chapter 5. Chapter 6 lists the criteria used to evaluate each hierarchical CS. HXCS's algorithm is described in Chapter 7. Experiments with HXCS in a simple non-Markov environment are presented in Chapter 8. Finally, Chapter 9 discusses the experimental results and the challenges awaiting future research.

2. Classifier Systems

A proper definition of a classifier system (CS) must first explain the larger context: the problem it was designed to solve. Imagine a complex, ever-changing environment with sparse payoff. An organism inhabiting such an environment must cope with perpetual novelty, unreliable (noisy) data, and does not receive any reward until it has completed a sequence of actions. To adapt, the organism must learn continually, but cannot spend too much time analyzing its experience if it is to operate in real-time. It needs to adopt an incremental, model-building approach; the predictions of tentative hypotheses are continually tested and revised, and new hypotheses are generated.

Traditional artificial intelligence techniques, such as expert systems, fare poorly in this kind of environment because of their rigidity; they were not designed for online learning, and attempts to graft a learning algorithm onto them typically yields awkward results. CSs, on the other hand, were designed from the start to be highly adaptable.

What follows is a description of the "standard classifier system." One must remember that the standard classifier system is an almost mythical beast, and is seldom implemented exactly as presented. In fact, much of this thesis will discuss variations of CSs that deviate considerably from the overview given in this section.

A CS is rule-based, and is similar to an expert system in this respect. The rules of a CS are called *classifiers*, and the two terms will be used interchangeably in this thesis. All communication is performed by passing messages, and multiple rules or messages may be active simultaneously. Typically, both the rules and messages have a simple representation, which facilitates the use of building blocks to form new and more

powerful concepts. The CS consists of three interacting subsystems: (1) the performance system, which interacts directly with the environment; (2) the credit assignment system, which evaluates the effectiveness of rules; and (3) the rule discovery system, which is responsible for generating new rules and removing ineffective ones.

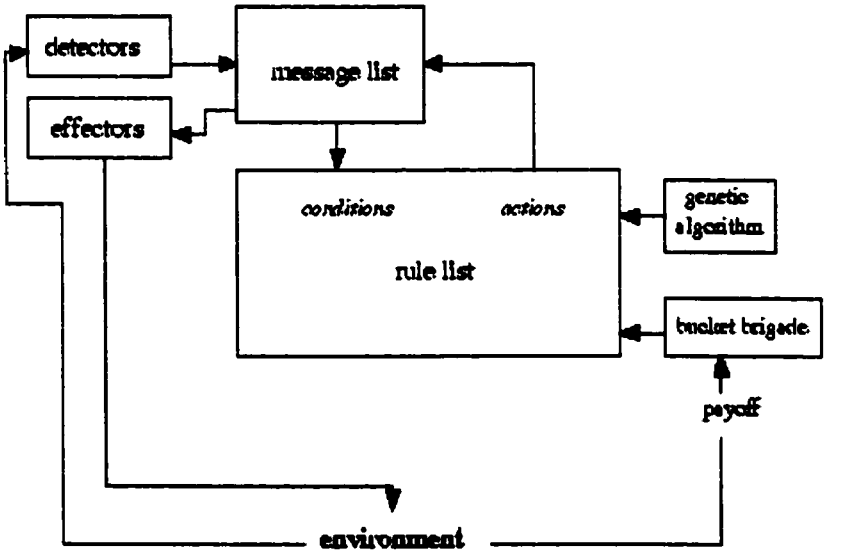

Figure 1: Architecture of a standard classifier system.

The Performance System

The CS perceives its environment with the detectors of its input interface, and acts on its environment with the effectors of its output interface. As mentioned previously, all communication, both between the CS and its environmental interfaces, and within the CS itself, relies on the passing of messages. All messages are posted to a common message list. These messages are produced by the input interface, to convey the current state of the environment, or are produced by the CS's own classifiers, to effect internal processing.

A classifier is a condition/action rule. If all parts of the condition are satisfied by the current list of messages, the classifier is eligible for activation. An activated classifier posts the message specified by its action to the message list. In a typical CS, the conditions and actions are both fixed-length strings, as are the messages. The actions and messages are binary strings. The conditions are ternary strings, because, in addition to the two binary values, each position may have the "don't care" symbol (#). A "don't care" symbol indicates that the condition will match either a 0 or a 1 at that position. For example the classifier 1#00:111 has a condition of 1#00 that is satisfied by either of the messages 1000 or 1100. If it is activated, the message 111 is posted to the message list.

Booker (1988) provides a concise overview of the performance system's execution cycle (p. 165):

(1) Place messages from the input interface on the current message list;

(2) Compare all messages to all conditions and conduct a competition among relevant classifiers to determine which ones will become active;

(3) For each active classifier, generate one message for the new message list;

(4) Replace the current message list with the new message list;

(5) Process the current message list through the output interface to produce system output;

(6) Return to step 1.

The competition mentioned in step (2) deserves some elaboration. A classifier may compete only if its condition is satisfied by the current messages (i.e., it is "relevant" to the current situation). Each competing classifier makes a bid for the right to become

active (the computation of the bid is described in the following section). Classifiers with higher bids have a greater probability of being selected.

It may be observed that the execution cycle permits a high degree of parallelism: many messages may exist concurrently, and these messages may in turn activate many classifiers. Thus, classifiers may be used as elementary building blocks, with clusters of rules acting together to represent complex concepts; each rule represents a different aspect of the current situation. Contrast this with a system that permits only one rule to be active at a time; a distinct symbol is needed for every complex concept – if these concepts are composed of many aspects, the combinatorics soon prove intractable.

One of the Holy Grails of CS research is the emergence of default hierarchies. The CS first learns some generalizations that capture the simplest statistical regularities in its environment. The rules that capture these regularities are overly general, and are inappropriate in many contexts. Eventually, the CS learns more specific rules that cover the special cases where the more general rule does not apply, and outbid the "generals" when appropriate. This process may continue indefinitely, as the system learns progressively more specific "exception" rules, leading to the formation of a default hierarchy.

The Credit Assignment System

This subsystem is responsible for evaluating the effectiveness of the rules. Each rule has a *strength*, a scalar value that represents its estimated utility. If the CS receives a reward (payoff, reinforcement) from the environment, it is obvious that the classifiers activated that cycle contributed to the attainment of this reward, and their strength should

be increased accordingly. But what of previously active classifiers which "set the stage" for obtaining the reward? In a complex environment with sparse payoff, the CS will execute many actions before it gets an external reward. Using minimal information, the CS must decide how credit will be assigned to its classifiers.

One solution is to use a brute-force approach, and record information for every classifier that is active over an extended period of time. Then, after the CS obtains a goal, a post-hoc analysis may be performed to determine which classifiers actually contributed to the end result (obviously, it is undesirable to reinforce classifiers that caused aimless meandering). This approach, sometimes called the *epochal method*, is expensive both in terms of memory and of processing time. Another disadvantage of the epochal method is that it is not always evident where an "epoch" (trial) should begin or end.

Because of the disadvantages of epochal methods, most CSs rely on local methods of reinforcement. If a rule's action improves the current situation, it is reinforced. *Temporal difference methods* are the most widely studied class of local reinforcement algorithms; for these algorithms, an improvement is indicated by an immediate reward from the environment, or a prediction of future reward.

The temporal difference method most often used by CSs is the *bucket brigade algorithm*. It may be explained using a simple economic analogy: Each rule is a "middleman," with *suppliers* and *consumers*. A rule's suppliers are previously active rules whose messages satisfied its condition, thereby enabling it to become active (the input interface may also be a supplier, if the rule's condition matched messages from the

environment). Its consumers are those rules whose conditions were at least partly satisfied by *its* message, *and* who subsequently became active (i.e., won the competition). The rule pays its bid to its suppliers, which decreases its strength (its "capital"). Its strength is increased when its consumers pay it with their bids, or when it receives a reward from the environment.

Each rule is said to be *coupled* to both its suppliers and its consumers. More precisely, two classifiers are coupled if the message of one satisfies at least one of the conditions of the other. It is the coupling of rules that makes it possible for the CS to learn action sequences.

The bucket brigade algorithm works because a rule may become strong only if it is coupled to other rules that lead to a good payoff. Otherwise, the rule progressively weakens as it pays its bid to its suppliers and receives little in return. However, the bucket brigade algorithm's ability to learn extended action sequences is limited in practice, as will be discussed in a later chapter.

The Rule Discovery System

For a learning system to be truly adaptable, it must be capable of generating new rules, and of eliminating ineffective ones. The new rules could be generated entirely at random, but wild guesses are unlikely to yield useful classifiers. A better approach would be for the CS to use induction to generate plausible "hypotheses" based on previous experience. This could be done by using "building blocks" of proven utility to construct new rules. Fortunately, it is easy to treat each rule as a collection of discrete parts, owing to its simple representation.

A part's usefulness may be estimated as the average strength of all rules which use that part. If a part is useful in many contexts (rules), it is plausible that it will be a useful component of new rules. However, the explicit computation of the average strength of all rules with a given part, for all parts, is too time consuming.

Fortunately, the preferential selection of above-average building blocks may be performed implicitly by selecting high-strength rules, recombining their parts to create new rules. This is precisely what a *genetic algorithm* is supposed to do; it is for this reason, the rule discovery system of the typical CS is a genetic algorithm (GA).

Each cycle of a GA consists of the following steps:

1. Select the parents from current "population," preferentially choosing individuals of above-average "fitness."
2. Create the offspring by randomly recombining the parts of the parents; various mutation operators may then be applied to the children.
3. The offspring are added to the population, and low-fitness rules are removed.

In a CS, the individuals are classifiers, and a classifier's fitness equals its strength.

3. Temporal Credit Assignment Problem

In a single-step problem, where each trial consists of a single decision, deciding how credit (or payoff) should be assigned to each classifier is relatively simple. The winning classifier, whose action was last executed, is rewarded if the system receives a positive reward from the environment; otherwise, it is penalized (Wilson has developed a somewhat different approach that will be discussed later). Other classifiers are neither reinforced nor penalized (unless taxes are applied), as they did not contribute to the current payoff.

The issue of credit assignment is much more complex for multi-step problems. The system may make many decisions (execute the actions of many classifiers) before an environmental payoff is received. When a reward is finally received, it is not obvious which classifiers were responsible; many "stage-setting" classifiers needed to be activated before the final classifier whose action triggered a reward.

One solution to the temporal credit assignment problem requires that the history of every decision is recorded; after a reward is received, some form of post-hoc analysis may be used to estimate the contribution of each rule to the final goal. This approach requires significant bookkeeping, and consumes much memory and processing time.

Most classifier systems restrict themselves to local reinforcement methods, the most common implementation being the bucket brigade algorithm. Though the bucket brigade is ideal for efficient on-line learning, it breaks down when it must reinforce an extended sequence of decisions. The first time the sequence is executed, the classifier whose action was last executed receives an external reward from the environment. Each

time this sequence is executed again, a fraction of this reward is effectively propagated to the previous classifier; many repetitions are required before the classifier with the first action in the sequence receives any part of the reward, and many more are needed before it is fully reinforced (i.e., its strength approaches its asymptotic value).

Wilson (1987) uses a simple simulation to demonstrate this point. Suppose a bucket-brigade sequence of length n is represented by a list of n classifiers, the strength of each having been initialized to zero. Let $S_i(t)$ be the strength of classifier i after time step t, and e be the bid factor, a small constant specifying the fraction of a classifier's strength that is used for its bid. On each repetition of the entire sequence, every classifier pays a fraction of its bid to its supplier. The classifier's strength at time $t + 1$ is therefore

$$S_i(t + 1) = S_i(t) - eS_i(t) + eS_{i-1}(t), \text{ for } t > 0$$

$$S_0(t + 1) = S_0(t) - eS_0(t) + R, \text{ for the } 0^{\text{th}} \text{ classifier, which receives payoff } R$$

directly from the environment.

All strength updates of one repetition of the bucket brigade sequence are considered to occur in a single unit of time. As t increases, all strengths approach an asymptotic value equal to R / e. If the critical level, the point at which a strength is considered sufficiently close to its true value, is considered to be 90% of R / e, the number of time steps necessary to reinforce the chain sufficiently is approximately,

$$t_{90\%} = (3 + 1.2n) / e$$

for $0.1 \leq e \leq 0.4$. Thus, for $e = 0.1$, a classifier ten steps from the environmental reward will require at least 150 repetitions of the sequence before it is reinforced sufficiently.

In fact, learning time is much greater in practice, because the selection algorithm is usually stochastic, and the system is not guaranteed to take the same path each time. Also, the system may wander its environment for a long time before it even has the opportunity to repeat the same sequence, or "chain."

John Holland acknowledged the long chain problem, but believed it could be resolved without any changes to the performance system. "Bridging," or "epoch-marking," classifiers could propagate the external payoff almost directly to early classifiers. A bridging classifier could be persistently active over an interval if its conditions were continually satisfied by: (1) unchanging sensory input from the environment, (2) its own message posted the previous cycle, or (3) a message from another classifier. Such persistent classifiers would represent goals to be carried out by the sequences they bridge. Holland believed that hierarchies of goals could arise naturally.

Riolo (1987) performed experiments to test the effectiveness of bridging classifiers. He concluded that they did indeed greatly reduce the time needed to reinforce long chains. However, no one has ever proposed a simple method for creating these classifiers, and they are not spontaneously generated. Without the addition of special, complex operators, the system would need to maintain many tentative bridges, which could lead to an intractably large population of classifiers.

Wilson proposed an alternative solution to the long chain problem: Decompose action sequences into modules at different levels of abstraction. The bucket brigade algorithm could then be modified so that each sequence at each level of abstraction would

form a separate reinforcement chain. This way, long chains would be replaced with multiple short chains. His proposal will be described in more detail in a later section.

4. XCS: Accuracy-Based Fitness

Observing that attempts to realize the full potential of Holland's framework had been mixed, Wilson decided to implement a simple CS that would be easy to study: a "Zeroth Level Classifier System" (ZCS) (Wilson, 1994). ZCS is basically a standard CS without a message list, like the "reactive CSs" used in other studies. Unlike some researchers, Wilson did not disable the genetic algorithm, as he considered Darwinian evolution essential to the definition of a CS.

ZCS was tested in two environments, Woods1 and Woods7. Each "woods" is a grid of cells containing obstacles ("trees" or "rocks") and food. Woods1 is Markov, that is to say, the environment's next state can be consistently predicted from its current state and the system's chosen action. Woods7 is non-Markov; it contains "aliasing" or "hidden" states which cannot be disambiguated unless the system maintains a history of previous inputs or key events. Since ZCS does not have any form of short-term memory (such as a message list), it cannot possibly achieve optimal performance in this environment.

For each environment, ZCS's performance (measured as the average number of steps to food) was significantly better than random, but markedly worse than the optimum, especially in the case of Woods7. Wilson hypothesized that performance was sub-optimal in the first environment because of premature convergence; ZCS would sometimes discover a sub-optimal path, and by reinforcing it, preclude the discovery of better paths. Of course, optimal performance was not expected in Woods7 for the reason mentioned above.

Shortly after running these experiments, Wilson tried a radically different approach that resulted in a new CS implementation, "XCS" (Wilson, 1995). XCS differs from its predecessor in several respects, but the key differences are (1) switching from a "panmictic" GA to a "niche" GA, and (2) basing classifier fitness on accuracy rather than strength (predicted payoff).

In a traditional CS, the GA selects parents from the entire classifier population. Booker (1982) introduced the concept of "niche," where a niche is approximately defined by a group of classifiers that match a set of environmental inputs. He pointed out that mating high-fitness classifiers from different niches tends to produce offspring that are not fit in either niche. Booker's solution was to limit the GA to the current match set, ensuring that parents are selected from the same niche. XCS used the same technique until Wilson (1998) learned that performance was improved if the GA were instead applied to the action set.

But the most important feature of XCS is accuracy-based fitness. Strength-based fitness creates many problems (Wilson, 1995). The CS's environment contains niches with different payoffs. Classifiers belonging to the more remunerative niches will have a higher strength and will be allocated more resources (classifiers) by the GA. Payoff sharing -- splitting payoffs among the classifiers in the action set -- is sometimes used to mitigate this problem, but this solution creates a new problem: a classifier's strength no longer directly predicts payoff; instead, predicted payoff equals the sum of the strengths of all classifiers in the action set. Since a classifier may belong to multiple action sets with different payoffs, even this interpretation is inaccurate.

Furthermore, even with payoff sharing, the more remunerative niches will receive more classifiers. This is problematic for sequential problems if payoff is discounted (to encourage short solution paths) so that "set up" classifiers are weaker than classifiers at the end of a chain. This problem is solved by the use of a "niche GA" (Booker, 1982), where the breeding population is restricted to the current match or action set. Since competition is restricted to the classifiers within a niche, differences in strength between niches are irrelevant.

The other problem with strength-based fitness is that there is no reason — neither in theory nor in practice — why accurate generalizations should evolve. The GA judges fitness solely by the average payoff received by a classifier; a classifier that receives a payoff of 100 half the time and 500 the rest of the time is judged equivalent to one that consistently receives a payoff of 300. Even worse, if a niche GA is used, overgeneral (inaccurate) classifiers will have more opportunities to reproduce (because they occur in more match/action sets) and will dominate the population.

To solve these problems, Wilson decided to base fitness on accuracy of the predicted payoff instead of the payoff itself. As a result, the focus of the CS was changed from learning how to exploit remunerative niches to learning an accurate internal map of its environment. This map says, "If the current input is x and action a is taken, the payoff will be p." Put more formally, the mapping is $\mathbf{X} \times \mathbf{A} \Rightarrow \mathbf{P}$, where $\mathbf{X}$ is the set of all sensory inputs, $\mathbf{A}$ is the set of all actions, and $\mathbf{P}$ is the set of payoffs. Instead of a *strength* parameter, each classifier in XCS maintains three parameters: (1) *prediction*, the predicted payoff if its condition is satisfied and its action executed; (2) *prediction error*,

the discrepancy between the prediction and the actual payoff received by the system; and (3) *fitness*, the accuracy of the prediction, computed as an inverse function of the prediction error.

XCS is also distinguished by its use of *macroclassifiers*. A macroclassifier is simply a classifier with a *numerosity* that indicates how many normal (micro-) classifiers it represents. Whenever a classifier is added to the population, the system checks whether a classifier with the same condition and action already exists. If so, instead of adding the new classifier, the existing classifier's numerosity is incremented by one. Otherwise, the classifier is added with its numerosity initialized to 1. Deletions are handled similarly by decrementing the target classifier's numerosity by one and removing the classifier entirely if its numerosity ever drops to zero. The algorithms are adjusted to account for each classifier's numerosity. Although macroclassifiers were primarily introduced as an implementation technique for improving performance, they also aid analysis. As the system learns, its population size (in terms of macroclassifiers) tends to shrink as the numerosities of the fittest classifiers are boosted and less fit classifiers are deleted.

Accuracy-based fitness obviously improves rule discovery; the GA eliminates overgenerals. But the performance component also benefits; by evolving a relatively complete map, the risk of premature convergence is reduced.

This map also benefits action-selection, since the system tends to evolve at least one accurate classifier for every niche and action combination. Thus, when making a selection, the system can compare the payoffs, or *system predictions*, for each action.

Each system prediction is a fitness-weighted sum of the predictions for all match set classifiers with a given action.

Exactly how the system chooses an action is not part of XCS's definition, but for current implementations the system alternates between choosing the action with the highest system prediction ("pure exploit") and choosing an action entirely at random ("pure explore") (Wilson, 1996a). It is necessary that every classifier is evaluated occasionally so that its parameters are updated and eventually approach accurate values.

After an action is selected and executed, the parameters are updated for all classifiers in the current action set [A] (for single-step problems or sequential problems that last a single step) or the previous action set $[A]_{-1}$. The prediction, prediction error, and fitness updates use a variant of the Widrow-Hoff rule (i.e., a recency-weighted average) called "moyenne adaptive modifiée" or "MAM". With MAM, the first several updates are straight averages, allowing the parameter to converge to its true value more quickly. The updating of $[A]_{-1}$ begins with the calculation of a payoff P: $P = r_{-1} + \gamma \max_a P(a)$, where r_{-1} is the reward on the previous time step, and γ is the discount factor ($0 < \gamma \leq 1$). If the update occurs in [A], $P = r$, where r is the current reward. P is then used to update the prediction p of each classifier in the action set using the Widrow-Hoff rule with learning rate β ($0 < \beta \leq 1$): $p \leftarrow p + \beta(P - p)$. The prediction error ε is updated via the same technique, using the absolute difference between the estimated and actual payoffs:

$$\varepsilon \leftarrow \varepsilon + \beta(|P - p| - \varepsilon).$$

The fitness update is a three-step process. First, the *accuracy* κ of each classifier is calculated: $\kappa = 0.1(\varepsilon/\varepsilon_0)^{-n}$ for $\varepsilon > \varepsilon_0$, else $\kappa = 1$ (ε_0 specifies an error threshold, below which the classifier is deemed accurate). Then, each classifier's *relative accuracy* κ' is calculated as

$$\kappa' = \frac{\kappa}{\sum_{[A]_1} \kappa}$$

Finally, the fitnesses are adjusted using the Widrow-Hoff rule: $F \leftarrow F + \beta(\kappa' - F)$. The calculation of κ was originally an exponential function (Wilson, 1995) but the power law has been found to work better in multistep problems (Lanzi and Wilson, 2000).

The genetic algorithm is applied whenever the average number of steps since the last GA activation for the classifiers in the target action set exceeds a threshold. When applied, it selects two classifiers from the population with probability proportional to their fitnesses, copies them, and applies crossover with probability χ and mutation with probability μ. These offspring are then inserted into the population. If the maximum population size is exceeded, two classifiers are deleted with probability proportional to their *action set size estimate*. That is, the larger the average size of the action sets to which a classifier belongs, the greater its chance of deletion. This pressure tends to equalize action set sizes, ensuring that the various niches have equivalent resources. The deletion probability is increased for very low fitness classifiers that have been updated a sufficient number of times.

Preliminary experimental results led Wilson to develop his *generalization hypothesis*: XCS, due to its accuracy-based fitness and niche GA, tends to evolve accurate and maximally general maps of the payoff landscape, $\mathbf{X} \times \mathbf{A} \Rightarrow \mathbf{P}$. That is,

classifiers evolve to be as general as possible while still satisfying an accuracy criterion. Two niches with the same payoff but different sensory inputs can be combined into a single niche by evolving classifiers whose conditions cover both sets of inputs. By evolving accurate, maximally general classifiers, the number of "concepts" (macroclassifiers) that the population needs is minimized.

Obviously, because fitness is based on accuracy, the system tends to evolve accurate classifiers. But why would they be maximally general? Wilson imagined the following scenario. Suppose there are two classifiers, C1 and C2, equally accurate, with the same action; C2's condition is a generalization of C1's. C1 and C2 are in the same action set, so their fitness is updated by the same amount. But C2 occurs in more action sets and therefore has more opportunities to reproduce. This results in more exemplars (identical copies) of C2, which means that C2 has a higher numerosity. Thus, as a macroclassifier, C2 will get a larger fraction of the fitness update, causing its reproductive advantage to increase further, and so on -- until C1 is eventually displaced. Generalization will continue as long as more general, equally accurate classifiers exist.

XCS successfully learned accurate maximal generalizations for the 6-multiplexer and 11-multiplexer problems. At the end of an average run (about 4000 problems), the system prediction error was near zero, and the population consisted of maximal generalizations (with the highest numerosities and fitness) and slightly more specific versions of the generals that were relatively new and inexperienced.

A comparison between the statistics for the 6- and 11-multiplexer problems was revealing: the 11-multiplexer required about three times as many steps and the final

population was about three times as large. But the ratio of the search space sizes is much greater: $2^{11}/2^6 = 32$. This discovery led Wilson to develop another hypothesis: for XCS, the learning complexity scales according to the number of concepts or generalizations expressed by the solution, and not exponentially with the dimensionality of the problem space.

Next, XCS was tested with "Woods2," a more challenging version of Woods1 with two types of food and two types of obstacle, to determine its ability to master sequential problems. Unlike ZCS, the system approached optimal performance after approximately 1000 problems. An investigation of the population revealed some evidence of generalization, but also many overlapping generalizations. Wilson attributed this apparent violation of the generalization hypothesis to Woods2's "sparseness." That is, the encoding for a classifier condition covers many environmental states that are not present in Woods2. For example, suppose there are two equally accurate classifiers, C1 =1#00:1 and C2 = 1000:1. If the environment includes state 1000 and not 1100, C1 and C2 will occur in exactly the same number of action sets, so the number of reproductive opportunities for each is the same. Thus, even though C1 is formally more general than C2, this added generality has no effect on its interactions with its environment, and it cannot displace C2.

Relationship to Q-learning

XCS's parameter updates are based on Q-learning (Watkins and Dayan, 1992), an algorithm that is currently popular with the reinforcement learning community. Q-learning associates a Q-value with every input-action pair; this is often implemented with a table. After each step, the Q-value of the relevant input-action pair is updated, using the Widrow-Hoff rule, with the sum of the current external reward and the product of a discount factor (between 0 and 1) and the largest Q-value associated with the following input:

$$Q_{t+1}(x,a) = (1-\alpha) * Q_t(x,a) + \alpha * \left(R + \gamma * MAX_b Q(y,b) \right)$$

where x is the current input, y is the subsequent input, α is the learning rate, R is the external reward, and γ is the discount factor.

Previous papers have explored the relationship between Q-learning and special versions of the bucket-brigade algorithm, such as the one used by Dorigo and Bersini's *discounted max very simple classifier system* (1994) or Wilson's ZCS (1994). Instead of a single Q-value, ZCS uses the sum of strengths of the maximum strength action in the match set, and the update applies to all classifiers in the action set rather than a single input-action pair. XCS's update procedure is closer to Q-learning because each classifier uses Q-learning to predict payoff directly. But this is still different from traditional Q-learning because XCS updates multiple predictions at once and these values are averaged to calculate the system prediction.

It has been proven that, for simple environments where the next input and reward are determined solely by the current input and the chosen action, Q-learning can learn an optimal policy; for every input x, the system chooses the action a that maximizes the

discounted sum of rewards it receives (Wilson, 1995). This proof does not apply to XCS; indeed, it has not been proven that any implementation of Q-learning with generalization is guaranteed to converge on an optimal policy. But this ability to generalize allows a solution to be expressed far more compactly than with traditional table-based Q-learning (which requires a slot for every possible input-action combination).

Improving generalization

As mentioned previously, the GA used to occur in the match set (as Booker had originally proposed) until it was found that performance was improved if it occurred in the action set (Wilson, 1998). Wilson attributed the difference to asymmetries in the problem space: given a maximally general classifier, it is not always true that another classifier with the same condition but a different action is also maximally general. By mating classifiers from different action sets, there is a high probability of creating unfit offspring.

Wilson (1998) also introduced a new technique called *subsumption deletion*, or simply *subsumption*, that accelerates generalization. The idea is simple: if an accurate classifier is a more general version of another classifier (i.e., it *subsumes* the other classifier), delete the more specific classifier and increment the more general classifier's numerosity accordingly. The subsuming classifier must meet a minimum experience threshold to ensure that its error estimate is accurate. Whenever the GA is executed, the offspring are checked against their parents for subsumption. Subsumption is also activated in the action set every time step, because a classifier may be subsumed by another that is not its parent.

It may be recalled that, in sparse environments such as Woods2, a classifier could not be displaced by an equally accurate and *formally* more general rival if it matched the same number of situations. Subsumption deletion solves this problem by allowing the more general classifier to subsume the other.

After 2000 explore problems in Woods2, XCS with an action set GA evolved a population of 310 classifiers versus the 500 classifiers of its predecessor. When subsumption was enabled, the population size dropped dramatically to 89 – evidence of powerful generalization. Further experiments suggested that this solution is very close to a minimal cover if the classifiers with low fitness and numerosity are ignored.

For the multiplexer problems, the action set GA made little difference. This was expected, since the multiplexer task is perfectly symmetrical; given an accurate maximally general classifier with an action of 0, a classifier with the same condition and an action of 1 is also accurate and maximally general.

Wilson's generalization hypothesis, that XCS tends to evolve accurate maximally general classifiers, was not aggressive enough for Kovacs, who proposed the *XCS Optimality Hypothesis*: XCS can evolve optimal populations that map all input-action pairs to payoff predictions using the smallest possible set of non-overlapping classifiers; more precisely, XCS eventually evolves a maximally general classifier for each payoff level with a greater numerosity than any other classifier for that level (Kovacs, 1997).

Kovacs experimented with two methods of evolving an optimal population, [O]: (1) condensation, and (2) subset extraction. Condensation is a simple technique that Wilson (1995) had used previously. The system learns normally until it is estimated that

the population contains [O] as a subset. Then the system continues to run with the mutation and crossover rates set to zero. With the genetic operators effectively disabled, no new macroclassifiers can be created; the offspring are perfect clones of their parents. Since the GA continues to breed and delete classifiers, the result is to shift numerosity to the fittest classifiers. Kovacs found that the process could be accelerated by deleting all but the most numerous classifier in each action set.

Condensation is simple and elegant. Unfortunately, no one has yet discovered a reliable method for determining when condensation should be triggered. It should not be activated until the population contains [O], but how can this be detected? Kovacs experimented with various triggering conditions, but he always needed to include a generous buffer (i.e., run a few thousand more problems after the triggering condition is satisfied); even worse, the size of this buffer was highly problem-dependent.

Another solution to the problem of discovering [O] would be to periodically check the population and see if it includes [O] as a subset. A brute force implementation that checked every possible subset for optimality would be far too expensive. Kovacs made a simplifying assumption: if a classifier belongs to [O], then any classifier with a greater numerosity also belongs to [O]. This is not strictly true, but it tends to be true, and is increasingly likely to be true as the population evolves.

- Kovacs's assumption led to his subset extraction algorithm:
- Add classifiers that are accurate and sufficiently experienced to an empty list in order of decreasing numerosity.

- As each classifier is added, check whether the list contains a complete overlapping map. If so, stop.
- If, after all the classifiers have been added, [O] still has not been found, or the classifiers overlap, quit.

Subset extraction is sufficiently fast that it can be executed at regular intervals. It is not guaranteed to detect [O] immediately (since those classifiers might not have the highest numerosities), but eventually it will.

Kovac's experiments revealed that condensation was faster than subset extraction *after it was triggered*, but since it was not activated until long after [O] already existed, subset extraction was effectively faster. A better heuristic for detecting [O] would dramatically improve the usefulness of condensation, however.

Adding internal memory

Without short-term memory, a CS cannot achieve optimal performance in a non-Markov environment. In his ZCS paper, Wilson (1994) suggested that an internal memory register be used as a replacement for the traditional message list. The system would contain a register with b binary bits. Each classifier would be enhanced with an internal condition and an internal action, each having b characters from {0, 1, #}. To qualify for the current match set, a classifier's external condition has to match the sensory input as before, and its internal condition must match the internal register. The execution of an internal action changes the register's bits: wherever the action's bit is 0 or 1, the corresponding register bit adopts the same value, and a # leaves it unchanged. A "no op"

is added to the set of external actions so that a classifier may change the system's internal state without taking any external action.

Cliff and Ross (1995) tested "ZCSM," an implementation of ZCS with internal memory. It was able to use the memory to distinguish between the aliasing states of simple mazes, achieving good but not optimal performance (as with ZCS in mazes with no aliasing states). The system became unstable if the memory register had more bits than necessary.

Lanzi (1998a) added memory to XCS to create "XCSM." XCSM successfully learned an optimal solution for Woods101, an environment with two aliasing states, if it had at least one bit of memory. Adding redundant bits did not affect XCSM's stability, unlike ZCSM. However, XCSM could not learn an optimal policy for a more difficult maze, at least not while exploration continued. When restricted to exploitation only, XCSM could eventually (thanks to covering) achieve good performance, but this approach is undesirable because a system that does not explore can become stuck in a local maximum and learning is very slow (Lanzi, 1998b). Lanzi concluded that XCS's exploration strategy was not suitable for XCSM because exploration occurs not only "in the environment" (external conditions and actions) but also "in the memory" (internal conditions and actions). To use the register effectively, the system has to do two things: (1) evolve an appropriate internal "language", and (2) learn how to use this language. (Wilson, 1999). It is difficult to learn how to use a language that is rapidly changing.

Lanzi (1998b) later developed XCSMH, a variant of XCSM, that does not explore internal actions during action selection. Even during exploration, the system always

chooses the internal action with the highest system prediction; next, a random external action is chosen from the internal/external action pairs matching the selected internal action. The internal state is explored only by the GA. Thus, the language is explored more slowly than its interpretation. XCSMH successfully achieved optimal performance in the more difficult mazes. Curiously, performance was improved when the register had redundant bits, suggesting that the amount of necessary exploration does not increase exponentially with the register size. There are many possible language-interpretation combinations that will solve the problem; any one will do (Wilson, 1999).

Using better representations

Wilson (1999) defines a *predictive regularity* as $(x, a) \rightarrow p$, where x is an input, a is an action, and p is the predicted payoff if the input is x and a is performed. $([x],a) \rightarrow p$ represents the set of all states x_i such that $(x_i, a) \rightarrow p$; this is called a *categorical regularity*. The ideal solution includes only one classifier per categorical regularity.

But XCS's ability to generalize is limited by the expressiveness of the classifier encoding, so this may not always be possible. For example, the traditional encoding is poorly suited to problems with continuous variables; many classifier conditions may be needed to represent a single interval. In this case, the usual string of ternary digits is better replaced with pairs of real values representing intervals either as start and end points or as the centre and spread.

The traditional representation is also less than optimal for non-conjunctive predicates. For example, to represent the condition "A or B" requires (at least) two classifiers, one with a condition A and one with a condition B. More complex conditions

such a "A < B" could require indefinitely many classifiers depending on the ranges of A and B.

Genetic programming (GP) represents its programs with Lisp *s-expressions*, which can easily express virtually anything so long as a suitable set of functions are provided. Wilson (1994) imagined *s-classifiers*, classifiers whose conditions are s-expressions. Lanzi and Perucci have tested XCS with s-classifiers and achieved good results for both the 6-multiplexer and "woods" problems. The flexibility of the s-classifier encoding ensures it is always possible to represent a categorical regularity with a single classifier.

Wilson (1999) envisions even more radical experiments, where a classifier's predicted payoff is changed from a constant to value that varies with the input. The condition, rather than being a predicate testing for a match, has become a payoff-predicting function. The solution's population size could be very small – even a single classifier. This is similar to the goal of GP, which is to evolve a single program, but approached along a different path.

Because s-classifiers have variable length, missing input variables are treated as "don't cares." Variable length conditions can also be represented as an unordered collection of <variable, value> pairs, just as with Goldberg et al.'s "messy GA". Lanzi developed a "messy XCS" and successfully achieved optimal performance on the test problems. The evolved population benefits from increased portability; it can be installed in a system with extra sensors and through mutation, the population will evolve to use them.

5. Previous Research on Hierarchical Classifier Systems

Hierarchical Chunking Algorithm

Weiss (1994) wanted to develop a system that used a local learning scheme to achieve performance comparable to that of a global learning scheme. In his case, the local method examined was the traditional bucket brigade algorithm (BBA), and the global method was the *profit sharing plan* (PSP) devised by Grefenstette (1988).

The PSP works as follows: Instead of assigning credit after every execution cycle, as with the BBA, classifiers are reinforced upon the completion of an *episode*, the interval between external rewards. The system must remember all classifiers that were active during an episode; when a reward is received, every one of these classifiers is directly reinforced with the reward. This is in obvious contrast to the BBA, where many repetitions of a sequence are necessary for an external reward to reach the earliest classifiers.

Grefenstette (1988) used a finite state world (FSW) example to demonstrate the limitations of the BBA:

> Consider the example in Figure 1 [Figure 1 here], which shows ten states, including the initial states A and B and the final states H, I, and J: The external rewards generated in states H, I, and J are 1000, 0, and 300, respectively. [...] Each example begins in a randomly selected initial state and ends when a final state is reached. The key feature of this example is that some rule (R_6) can fire in

two possible states, one firing leading to high reward and the other leading to low reward (p. 233).

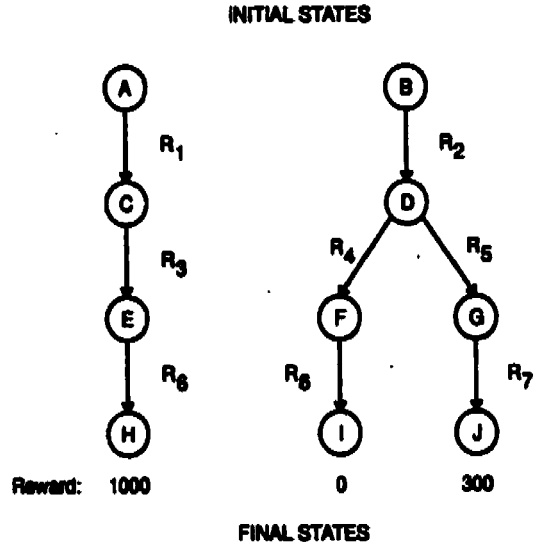

Figure 2: Finite State World example (Grefenstette, 1988, p. 233)

Using the PSP, the system was able to learn that R_5 is better than R_4 in state D; using the BBA, it could not.

The PSP is relatively simple for a global algorithm, but it requires significantly more memory than the BBA, and reinforcement is done in "batches," instead of incrementally, which could be problematic in a real-time system. Also, the PSP may permit the growth of "parasitic" classifiers that serve no useful function, but are reinforced as well as useful rules. And it is difficult to apply the PSP to environments where episodes are not well defined.

Weiss's attempt to combine the best features of local and global learning used the idea of *chunking*, a concept first used in psychological models of memory. Two pieces of knowledge that are highly correlated may be "chunked," allowing them to be treated as a

single memory unit. Chunks may in turn become components of larger chunks, so the chunking process is hierarchical. Appropriately, Weiss's application of chunking to classifier systems is called the *hierarchical chunking algorithm* (HCA). Currently, the HCA is applicable only to reactive CSs; that is, systems where only a single winning classifier is selected each cycle, and there are no internal messages.

The HCA is distinguished by its simplicity. If the strength of the winning classifier is significantly higher than the average strength of its competitors, a new classifier is created that "chunks" the current winner with the winner from the previous cycle. The new classifier has the same condition as the previous winner (since the action sequence begins at the previous state), and its action is formed by appending the action of the current winner to that of the previous winner. Thus, if the previous winner is C_1/A_1 and the current winner is C_2/A_2, where C_i/A_i denotes a condition/action pair, the resulting chunk is C_1/A_1A_2. Conversely, an extended classifier is deleted if its strength is significantly worse than the average.

A classifier's *length* equals the number of actions in its action part. A classifier with a length of 1 is an *elementary classifier* (all rules in a standard CS are elementary classifiers); a classifiers with a length greater than one is an *extended classifier*. The performance cycle is modified in two ways to accommodate the existence of extended classifiers: (1) the competition is hierarchical; among the satisfied classifiers, only the highest-level ones are allowed to compete; (2) when a classifier's action part (which may include multiple actions) is executed, each action is processed immediately by the output

interface, allowing the system to change the environment before the next action in the sequence is executed.

The test problem used for an experimental comparison of the HCA with the BBA and PSP was a navigation task. The CS navigated 10 x 7 grid with obstacles, learning over time how to reach a fixed location (the goal state) from each possible location. The implementation of the PSP was limited so as to reinforce only the last four winning classifiers, instead of every winning classifier in an episode. Instead of the traditional ternary encoding for classifier conditions, a decimal encoding was used, thus avoiding the problems of using a discontinuous encoding to model a continuous environment.

The HCA performed significantly better than the BBA, though not as well as the PSP. It was proven that, for only a small increase in computational complexity, a local learning algorithm can achieve performance comparable to that of a global algorithm.

However, the HCA has some limitations. It is applicable only to reactive CSs which lack the advantages of internal message passing and parallelism. For an action sequence to be represented compactly, there must be an associated "chunk" or extended classifier; if there are many sequences to be learned, much memory is needed to represent each one as a single unit.

An extended classifier essentially "looks for" an initial sensory input, and then executes a sequence of actions. This technique is fine for a static environment, but is less appropriate for domains where the CS's activity could be interrupted at any moment. It is also problematic if the same sensory input applies to different situations, such that an

action sequence that is appropriate for one case is inappropriate for another; this is known as the *aliasing problem*, and it is one of the banes of local reinforcement learning.

But the HCA neatly mitigates much of the long chain problem. If an extended action sequence is represented by a single rule, or a small number of rules, the chains are relatively short. Shorter chains are more quickly reinforced, and are less likely to be disrupted by rule discovery.

Anticipatory Classifier Systems

In psychology, classical reinforcement learning theory posits that stimulus-response (S-R) associations are the fundamental units of learning; if an organism executes a response R in the presence of a stimulus S, and receives an environmental reward after doing so, that S-R association is reinforced. Tolman, however, determined that stimulus-response-stimulus (S-R-S) units were the true basis of learning, a theory supported by his experiments with rats navigating a T-maze. Seward later performed similar experiments that confirmed Tolman's theory (Stolzmann, 1996).

More recently, Hoffman developed a variant of Tolman's theory, *the theory of anticipatory behavioural control* (Stolzmann, 1996). He replaced the concept of an S-R-S unit with that of an S-R-C unit, where C represents the behavioural consequences anticipated by the organism if it executes R in the presence of S. With such units, it is possible for an organism to learn an internal model of its environment even if there is no external reward. When the organism reacts to a stimulus, the anticipated consequence C_{ant} is compared to the actual consequence C_{real}. If they are the same, the unit is reinforced; otherwise S is differentiated, creating a new S-R-C unit.

Hoffman preferred to implement his theory with recurrent neural networks. A problem with neural networks, however, is that they model only input-output behaviour; the internal steps by which intentional learning takes place are not explicitly represented. Since anticipatory behavioural control is an extension of classical reinforcement learning, on which CSs are based, Stolzmann (1996) decided that an *anticipatory classifier system* (ACS) would be an ideal formalization of Hoffman's theory; in a CS, rules are explicitly represented, as are the operations that modify them.

Whereas Hoffman's theory defines two kinds of learning – (1) differentiation of the starting situation (condition) and (2) reinforcement of the rule – Stolzmann's implementation also allows (3) differentiation of the anticipated consequences and (4) diminishing the rule-strength. Unlike a standard CS, ACS does not use a GA for rule discovery. Instead, credit assignment and rule discovery are integrated, and the system uses the immediate environment input to compare the anticipated consequence with the actual consequence; falsely anticipated properties are noted, and a new, correct classifier is created. Thus, ACS uses a form of intentional rule discovery.

Each ACS classifier has three parts (Stolzmann, 1996):

(1) A condition S that defines an environmental *situation*; it is defined as an L-character string, where each character is an element of {0, 1, #} and L is the number of detectors. In other words, it is identical to the condition-part of a standard classifier.

(2) An *response* R, representing that classifier's action. It is an M-character string, where each character is an element of {0, 1, #} and M is the number of effectors. Most CSs

do not allow actions to contain '#'s; in ACS, a '#' at R's i^{th} position means that the response does not change the i^{th} effector.

(3) A *consequence* C that represents the attributes of the anticipated environmental state. Its representation is identical to that of S; a '#' at i^{th} position means that the classifier believes R doesn't change the environmental attribute perceived by the i^{th} detector. The actual anticipated environmental state m_2^{ant} is a function of both the current environmental state m_1^{real} and the anticipated consequence C^{ant}: $m_2^{ant} =$ *passthrough*(m_1^{real}, C^{ant}), where the passthrough function is defined as

$$passthrough(x, y) = z \leftrightarrow \bigwedge_{i=1}^{L}((y_i = \# \rightarrow z_i = x_i) \wedge (\neg y_i = \# \rightarrow z_i = y_i))$$

In other words, the '#'s in C^{ant} are replaced with the bits at the corresponding positions in m_1^{real}.

Thus, each (S, R, C) classifier may be interpreted as: if the current environment state satisfies S, send R to the effectors, make a new message from C and the current environmental state, and add this message to the message list.

The algorithm for a *behavioural act*, or major cycle, is as follows:

(1) The detectors add the current environmental state m_1^{real} to the message list.

(2) Find the set of all classifiers whose S-part matches m_1^{real}.

(3) Run a strength-based competition to determine which classifier in the match set will become active.

(4) The winning classifier c_j sends its reaction R_j to the effectors, causing the CS to execute a motor action.

(5) A new message is formed from the winner's anticipated consequence C_j and m_1^{real}, and is added to the list.

(6) The environment's new state m_2^{real} is added to the list.

(7) The system assumes that m_2^{real} is a direct result of executing R_j and compares m_2^{real} with m_2^{ant}, and applies the equivalent of either credit assignment or rule discovery.

Then the next behavioural act begins, with m_2^{real} replacing m_1^{real}.

Each classifier *c* has two rule-strengths: (1) $s_c^{ant}(t)$, the accuracy of the anticipation, and (2) $s_c(t)$, the predicted environmental reward, equivalent to the strength attribute of a standard classifier. The parameter *t* is the number of behavioural acts executed by the system.

When ACS received an external reward *r(t)*, it adjusts the predicted payoff for all match set classifiers that correctly predicted the next environmental state:

$$s_c(t+1) = (1-b^{real}) * s_c(t) + b^{real} * r(t) * spec(C^{ant})$$

where b^{real} is the *risk factor*, a small constant, and $spec(C^{ant})$ is the *specificity*, the number of non-'#' symbols in C^{ant}.

The strength-based competition is similar to that for a standard CS. Each classifier c_i in the match set makes a bid, and the winning classifier is chosen using roulette wheel selection. The bid is a function of both of c_j's rule-strengths and its specificity:

$$b_c(t) = s_c(t) * s_c^{ant}(t) * 2^{spec(c)}$$

Since all new classifiers are created as more specific versions of their inaccurate parents it is important for specificity to be a component of the bid to ensure that these exception rules override their more general ancestors. Thus, ACS encourages the formation of default hierarchies.

Let $c_j = (S_j, R_j, C_j^{ant})$ be the winning classifier. The rule discovery component of ACS, and the updating of the winner's accuracy, is handled by the following cases:

Case 1: The anticipated state m_2^{ant} matches the actual state m_2^{real}. No new classifier is created.

Case 1.1: R_j changes the current state (i.e., m_2^{real} does not match m_1^{real}).

$s_{c_j}^{ant}(t+1) = (1-b^{ant}) * s_{c_j}^{ant}(t) + b^{ant}$, where b^{ant} is a small constant

Case 1.2: The environmental state has not changed ($m_2^{real} = m_1^{real}$). The winner's accuracy-based strength is diminished, since its reaction was useless.

$$s_{c_j}^{ant}(t+1) = (1-b^{ant}) * s_{c_j}^{ant}(t)$$

If $s_{c_j}^{ant} < \delta_{delete}$, c_j is deleted.

Case 2: m_2^{ant} does not match m_2^{real}.

Case 2.1: All the components of S_j for which m_2^{ant} does not match m_2^{real} are #'s. The rule is not specific enough, so a new classifier is created that is different from c_j only in the non-matching components of S_j and C_j^{ant}; for these components, $S_{new} = m_1^{real}$ (the classifier's assumptions are corrected) and $C_{new}^{ant} = m_2^{real}$ (the classifier's expectations are corrected).

Case 2.2: c_j's assumptions and expectations are not correctable; its strength is decreased as it is for Case 1.2. No new classifier is created.

Stolzmann calls ACS a "quasi-reactive" CS. It is reactive because of the "winner-takes-all" competition and the absence of internal messages. But it is possible for ACS to use its internal model of the environment to build chains of classifiers. If a classifier's anticipation of the behavioural consequences of its activation is certain (i.e., it is accurate; $s_c^{ant}(t) \geq 1 - \varepsilon^{sure}$, where ε^{sure} is a small constant), then the internal representation of the previous environmental state can be replaced by the anticipated consequence. This can save time, since the CS does not need to wait for the detectors to send the next message. The bid competition is then based on the anticipated environmental state. When the last classifier receives an external reward, all classifiers in the chain get the reward, as per Grefenstette's profit-sharing plan (PSP) (1988).

An ACS can also save time by building *behavioural sequences*. If (S_1, R_1, C_1^{ant}), (S_2, R_2, C_2^{ant}) is a chain of two classifiers, this behaviour can be encoded in a single new classifier: $(S_1, (R_1, R_2), passthrough(C_1^{ant}, C_2^{ant}))$. The compound reaction (R_1, R_2) tells the system to first send R_1, and next R_2, to the effectors. The consequence of the new classifier must be $passthrough(C_1^{ant}, C_2^{ant})$ rather than C_2^{ant} so that the specific anticipations of C_1^{ant} are retained. A behavioural sequence can have more than two reactions, since it can be formed from classifiers that already contain behavioural sequences.

Finally, an ACS can save time by shortening behavioural sequences. If the two reactions R_i and R_{i+1} are independent (i.e., R_i matches R_{i+1}), $passthrough(R_i, R_{i+1}) = passthrough(R_{i+1}, R_i)$ may be substituted for (R_i, R_{i+1}). However, for this to work, the order in which R_i, R_{i+1} are executed should not matter. For example, to safely change

lanes, the driver must check the other lane for traffic and then move to that lane if it's clear. Although the two reactions are independent, it would be unwise to check for traffic *while* changing lanes.

Stolzmann used ACS to simulate Seward's experiments with rats in a T-maze, just as Riolo had done with CFSC2, a CS capable of lookahead planning (Riolo, 1991). Riolo summarizes the results of the real world experiments as follows:

> In a pre-reward phase, satiated rats are allowed a number of trials in the maze shown in Figure 1 [Figure 2 here], where a trial consists of being placed in the start box S, making a choice to go left or right, and then being removed from one of the end boxes, F or E, depending on the choice made. (there is no food in either end box). Next the rats are not fed for 24 hours, placed *directly* in one of the end boxes with food in it, and allowed to eat for one minute. Then the rat is placed in box S and the direction the rat chooses on the *first* post-eating trail is recorded, i.e., toward the box with the food or not. Typical results are: (1) Rats not given pre-reward trials but with identical end boxes again go toward the box with food 50% of the time, as expected if they guess with no information; (2) Rats allowed pre-reward trials but with identical end boxes again go toward the box with the food 50% of the time; (3) But rats allowed pre-reward trials with boxes that are easily distinguishable for a rat (e.g., one has white walls and the other has dark), choose the path to the box with food 90% of the time, even though that choice of turn-left (or turn-right) has never led to a food reward! [Riolo, 1991, p. 317]

Seward's experiment proved that rats don't learn through the reinforcement of S-R units, because they learned an internal map of the maze in the absence of any external reward.

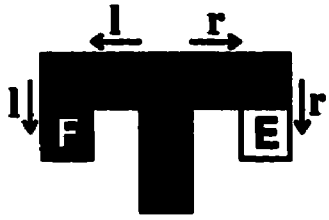

Figure 3: T-maze used in Seward's experiments (Stolzmann, 1996, p. 11)

To permit the ACS to detect the required environmental attributes, it was given five effectors:

d_1: Is it possible to go forward? (0 or 1)

d_2: Is it possible to go left? (0 or 1)

d_3: Is it possible to go right? (0 or 1)

d_4: Is it the black box, the white box, or neither? (1, 2 or 0)

d_5: Is there food? (0 or 1)

The ACS had one effector capable of several actions: move (f)orward, move (l)eft, move (r)ight, and (w)ait. It was allowed to build behavioural sequences with a maximum of two reactions. Neither chain-building nor behavioural sequence shortening were permitted.

Each experiment began with a pre-reward phase consisting of 200 trials in a T-maze without food. The initial rule population consisted of one maximally general classifier per reaction: (#####, f, #####), (#####, l, #####),(#####, r, #####),(#####, w, #####). During this phase, the ACS would learn an internal model of the T-maze, and then would usually build behavioural sequences that would allow it to go directly from S

to F or from S to E. Next, to simulate feeding the rat in box F, classifiers that anticipated the attributes of F were reinforced.

When Seward tested 32 rats in a maze with distinguishable end boxes, four chose the wrong direction. The simulation results were similar, except that wrong decisions were slightly more frequent. This occurred because if ACS chose the wrong direction (right) the first time, it would calculate a better (more accurate) classifier than the one never used for going left. Such premature convergence illustrates the explore/exploit dilemma (Wilson, 1996); the system may choose to usually exploit an existing accurate path instead of exploring other possibilities. This tendency is controlled by b^{ant} and $s^{ant}(0)$.

When the end boxes were not distinguishable, the simulation results were similar to their real world counterparts: the wrong direction was chosen approximately 50% of the time. The experiments with no pre-reward phase were not simulated; it is obvious that an ACS with no experience has a 50% chance of choosing either direction.
Like Weiss (1994), Stolzmann presented his CS implementation as a solution to the locality/globality dilemma.

Weiss's HCA provides one solution to the locality/globality dilemma for reactive CSs. An ACS that uses behavioural sequences but not chains also qualifies as a reactive CS, and is capable of learning the correct behaviour for Grefenstette's environment (Figure 1). As the ACS encounters cases where it fails to correctly anticipate the consequences for a reaction, a new classifier is created that accurately anticipates the consequences. If all the FSW's states are visited at least once, all of the anticipations will

be accurate. As the anticipatory rule-strengths of these classifiers are reinforced, the rules become certain; this allows the system to build behavioural sequences. If the maximum sequence length is at least two, the ACS will learn how to travel to I or J when the current state is D; since the reward for reaching J is 300 versus 0 for reaching I, the classifier that moves the system from D to J will have the higher reward strength and will tend to win the competition. ACS's behavioural sequences are analogous to Weiss's "chunks."

HCS: Keeping it Within the Family

A genetic algorithm may seem like the perfect solution for discovering new rules, but it also poses a serious problem: the GA tends to disrupt the cooperative structures the CS needs to learn. This dilemma has two aspects: (1) the rule clustering problem, and (2) the rule association problem (Shu 1991).

Rule clustering refers to the system's need to learn a group of complementary sub-solutions, instead of optimizing for a single, best solution. A behaviour that is optimal under one set of circumstances may be wholly inappropriate under another. There is a danger that the GA may favour the "best rule" (the one with the highest strength), and allow its pattern to dominate the population. This is akin to the problem of premature convergence with GAs used for function optimization.

To solve a problem of any significant complexity, classifiers need to cooperate with one another. These cooperative structures include default hierarchies and classifier chains. Unfortunately, the GA is oblivious to such bonds, and tends to disrupt them; this is the rule association problem. The GA seeks only to propagate the patterns of rules

with above-average fitness. Part of the problem is that a classifier's fitness traditionally equals its predicted payoff (strength); unlike the accuracy-based fitness of XCS.

Here are a couple of examples showing why the GA is likely to destroy cooperative structures. The bucket brigade algorithm causes classifiers at the end of a chain (close to where the goal is actually attained) to have a higher strength than those at the start of the chain. As a result, the GA tends to select the final classifiers in the chain for reproduction, and delete the important stage-setting classifiers.

A similar problem exists for default hierarchies. Classifiers representing exceptional cases should override the decisions of more general classifiers when appropriate. To this end, the bidding competition is usually biased towards more specific classifiers. Unfortunately, the system cannot distinguish between a true exception and a rule that is merely a less general version of an existing classifier. The result: the "generals" are starved, and are deleted by the GA.

Shu and Shaeffer (1991) decided that the best way to preserve important relationships among classifiers was to make these relationships explicit. Classifiers are grouped as *families* according to relations; these relations could be defined as default hierarchies or classifier chains, for example.

The performance and genetic algorithms were modified to take these groupings into account. A classifier's bid depends not only on its own strength, but also that of its family; a family's strength is the sum of the strengths of its members.

Because it is undesirable for the GA to create competition amongst a cluster of cooperating classifiers, the rule discovery subsystem was modified so that the probability

of a genetic operation between families is much higher than that of one between classifiers within a family. Inter-family crossover can take one of two forms: (1) a random number of classifiers from one family are swapped with those of another; (2) a crossover point is randomly chosen, and every classifier within a family swaps the segment after that point with the corresponding classifier in another family. Deletion always occurs at the family level.

Shu and Shaeffer called their system HCS, because it was a hierarchically structured classifier system. The modifications were intended to reduce both the rule clustering and rule association problems. One form of the rule clustering problem occurs when two classifiers belonging to a solution set are crossed, producing offspring that do not belong to the solution set. Because of the specified relation, however, it is likely that the classifiers will remain in the same family; since mating rarely occurs between classifiers within the same family, the risk of incompatible offspring is low. There is also less chance of premature convergence, which is caused when a "super-classifier" in the initial population quickly crowds out patterns belonging to other solution sets with its offspring. The same situation in an HCS would require a "super-family" in the initial population – a much rarer event.

The rule association problem is addressed by defining relations that explicitly group the classifiers into families containing chains or default hierarchies. Then, it is less probable that the GA will disrupt a cooperative structure, since intra-family competition is rare. In the case of default hierarchies, the problem of distinguishing true exception classifiers from classifiers that are already covered by more general ones disappears; only

true exception classifiers will belong to the family for a given hierarchy. This makes it possible to have a bidding scheme that favours more specific classifiers only locally within the family. This way, the general classifiers aren't starved by more specific variants of the same pattern.

The HCS was tested with a suite of boolean functions. All families within a given population had the same size. Curiously, no relation was specified; the classifiers were grouped randomly. The experiments used a variety of population and family sizes.

It was found that, when the population size was small relative to the problem space, larger family sizes tended to improve performance; the families allowed multiple sub-solutions to develop without interfering with one another. For the same reason, the HCS was more stable overall than a traditional CS.

If the family size was too large, however, performance could degrade, because the population's classifiers would be bound within a relatively small number of families. Also, larger families imply a greater risk of parasitic classifiers, which arise when ineffective classifiers survive because they belong to a strong family.

The HCS offers a middle ground between the Michigan and Pittsburgh approaches (Shu 1991). Like the Pitt method, the HCS encourages and protects cooperating classifier structures, but is less computationally complex, both in time and space. Classifiers are grouped into small structures, not complete programs, and credit assignment is based on individual classifiers as in the Michigan approach.

Hierarchical Credit Allocation

As described in the section about the temporal credit allocation problem, the bucket brigade algorithm (BBA) is poor at reinforcing long action sequences. Holland thought "bridging classifiers" could arise naturally, and would represent behavioural modules (goals) while feeding rewards directly to stage-setting classifiers (Wilson, 1987).

Wilson was not so optimistic. He proposed an alternative solution to the long chain problem: Decompose action sequences into modules at different levels of abstraction. The bucket brigade algorithm could then be modified so that each sequence at each level of abstraction would form a separate reinforcement chain. This way, long chains would be replaced with multiple short chains (Wilson, 1987). The message list, which is homogeneous in a standard CS, becomes hierarchical, with only one message allowed per level at a time (this restriction was included as a simplification).

Modifications to the performance and credit assignment algorithms are required to implement this proposal. Wilson's hierarchical credit allocation (HCA) algorithm is summarized as follows (Wilson, 1987):

At any moment, the system's phase is either "ascent" or "descent". During the "descent" phase, the bidding competition is restricted to classifiers matching both the current environmental input and the last (lowest-level) message posted to the list. The winner's strength is reduced by its bid, and then it posts its message below the lowest-level message. The process is repeated until the winning classifier's message is an external action. In this case, the message is executed through the output interface, and is not posted to the list. Then, the system's phase becomes "ascent."

While ascending, the match set includes classifiers that match both the current environmental input and *any* message in the list. The winner has its strength reduced and message posted as during descent, with an important difference: the winner pays its bid to the classifiers that posted the messages below the one it matched, and those messages are removed from the list.

If payoff is received from the environment, it is paid to each of the classifiers that posted messages currently on the list, as well as to the classifier whose action was just executed, all messages are erased, and the phase becomes "descent."

Wilson's detailed description of his hierarchical performance and reinforcement algorithm is as follows (Wilson, 1987):

1. Obtain the current message E from the environmental input interface.
2. If phase = "descent", form the match set [M] of all classifiers which match both E and the lowest level message on the message list, else

   If phase = "ascent", form the match set [M] of all classifiers which match both E and any message on the message list.
3. Compute the bid *B* of each classifier C in [M].
4. Select a classifier C* from [M].
5. Reduce C*'s strength by the amount of its bid;

   then

   If phase = "ascent", pay an amount equal to *B* to each of the classifiers (if any) which sent messages lower on the list than the message matched by C*, erase those

lower messages, and pay an amount *B* to the classifier whose action was carried out on the previous time-step.

6. If C*'s message is an external action, set phase = "ascent",

   Else post the message on the next lower empty level of the message list and set phase = "descent".

7. If the message of step 6 was an action, take it.

8. If payoff *R* is received from the environment, pay amounts *R* to each of the classifiers who sent messages now on the list, erase all messages, pay an amount *R* to the classifier whose action was just taken, and set phase = "descent".

In essence, the system behaves as though it is progressively drilling down from high-level goals to lower-level goals, and finally to the motor actions that achieve those goals.

Figure 4 provides an example of how the algorithm solves a problem. At t_0, the phase is "descent," the environmental state is E_1, and the message M_1 is added to an empty message list, which is assumed to represent an internal goal such as obtaining food. At t_1, the environmental state is unchanged, so the match set includes classifiers matching both E_1 and M_1. The winning classifier's message, M_2, is then posted below M_1. At t_2, the winner is chosen from classifiers matching E_1 and M_2, and its message M_3 is appended to the list. At t_4, the competition includes classifiers matching E_1 and M_3. This time, the winner is an external action rather than an internal message, and the action is executed. The environmental state changed to E_2 and the system's phase becomes "ascent." At t_5, the match set includes all classifiers matching E_2 and *any* message in the list. Once again, the winner is an external action, which is executed, changing the

environmental state to E_3. Because of the external action, the phase is still "ascent," and the match set at t_5 includes all classifiers matching E_3 and any message on the list. The winner's message M_4 is internal; its condition matched M_3, so M_3 is replaced by M_4, and the phase becomes "descent." One could say that the behavioural module represented by M_3 has been completed.

Eventually, at t_{15}, the system executes an action that is rewarded by the environment. The classifiers that posted messages M_1 and M_5 (currently on the list) receive the full reward R, and so does the classifier that executed the last action. The arrows in the diagram indicate the flow of reinforcement signals; e.g., when M_4 is posted, its classifier pays its bid to the classifiers that posted M_3, as well as to the classifier whose action was just executed (because these rules were responsible for activating M_4's classifier). The hierarchical distribution of payoffs means that the average payment sequence length is O(log n) rather than O(n), where n is the number of decision steps.

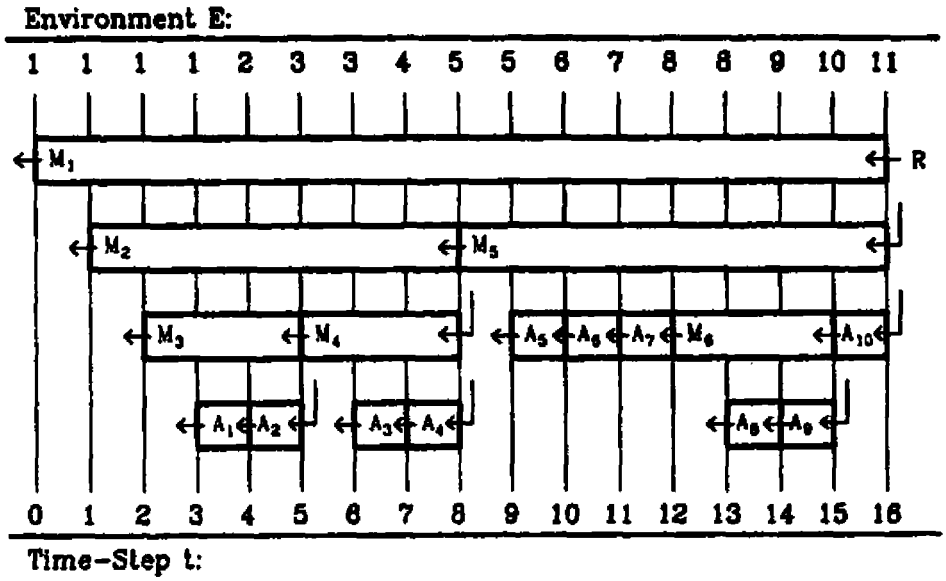

Figure 4: Example of Hierarchical Performance and Reinforcement Algorithm (Wilson, 1987)

MonaLysa: A Motivationally Autonomous Animat

Donnart and Meyer (1994, 1996) wanted to design a control architecture that was capable of learning situated plans. Traditional planning algorithms construct a complete, static, up-front plan that is then used to control a robot's behaviour. Donnart and Meyer, on the other hand, believed plans should be more dynamic: the robot acts reactively at first, and plans are subsequently abstracted from its experience; instead of plans that are fixed in advance and slavishly followed, they are built incrementally, may be adapted at any time, and may be overriden in response to the robot's immediate sensory information.

The resulting architecture was named *MonaLysa*, an amusingly awkward acronym that stands for MOtivatioNalLY AutonomouS Animat. Though it is sufficiently flexible to be adapted to any domain, the implementation was built to solve navigational problems. The virtual animat has proximate sensors that allow it to sense the presence or absence of an obstacle immediately in front, 90° to its right, or 90° to its left. It can estimate the coordinates of its current position and the direction of the goal to be reached. The animat has three actions: move forward, move 90° to the right, and move 90° to the left, where the size of a move equals the animat's length.

MonaLysa's architecture consists of five modules, shown in Figure 5: (1) the reactive module, (2) the planning module, (3) the context manager, (4) the auto-analysis module, and (5) the internal reinforcement module. The cooperation of these modules enable the animat to reactively escape any obstacle by skirting it, and to analyze its path afterwards to discover a plan that will enable it to avoid the obstacle from a distance in the future.

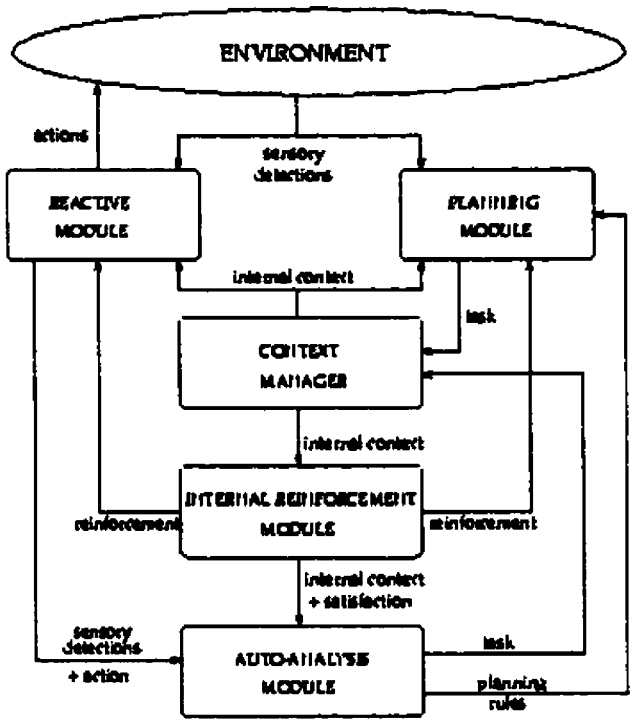

Figure 5: The MonaLysa architecture (Donnart and Meyer, 1996, p. 2).

The reactive module decides which motor action to perform at every moment. It contains rules that respond to a combination of the current environment input and the animat's own internal context. Because the implementation was designed to solve navigational tasks, the internal context is the direction of the current goal, as specified by the context manager. Each reactive rule has the form:

```
If <sensory information> and <direction of current goal> Then <action>
```

A concrete example of such a rule R would be $100 \mid 001 \Rightarrow 01$, where "100" indicates that there is an obstacle ("1") in front of the animat but both sides are clear ("0"). "001" indicates that the current goal is 45° to the right. The action, "01", means "move 90° to the right."

For each pair of conditions, *<sensory information>* and *<direction of current goal>*, there are three rules, one for each action. From the matching rules, a winner is chosen probabilistically based on strength. When the system is initialized, all 192 possible rules (8*8*3) are created in advance, and the number never varies. Donnart and Meyer (1996) mention that a genetic algorithm has been used to discover more general rules.

The purpose of the planning module is to decompose tasks into subtasks based on the current sensory input. Like the reactive module, it contains rules. Each rule has the form:

```
If <sensory information> and <current task> Then <subtask>
```

where *<sensory information>* includes information from the proximate sensors, the animat's coordinates, and its current orientation, and *<current task>* is the topmost task in the context manager's pile. If the rule is triggered, *<subtask>* is added to the context manager's pile, on top of *<current task>*. A task is represented as a pair of coordinates that specify the initial and final (goal) positions. For example, the planning rule P: $5,1 \mid 001 \mid 000 \mid 3,0;3,5 \Rightarrow 5,1;5,2$ means "if the current position is (5,1), the animat is headed north-east ("001"), there are no obstacles ("000"), and the current task is moving to (3,5) from (3,0), then the new sub-goal is to move from (5,1) to (5,2)."

Each planning rule has two strengths, a local strength that determines its probability of being triggered when its condition is satisfied, and a global strength that determines its probability of being deleted should the population exceed its maximum size. The planning module is initially empty; it acquires rules as they are generated by the auto-analysis module.

The context manager is responsible for supplying an internal context to the other modules. For the reactive module, this is the direction of the current goal; for the other modules, it is the current task. The context manager contains a pile of tasks, with the current task at the top. New tasks are added by the auto-analysis module and the planning module. If the task is posted by the planning module, the corresponding goal is described by its final coordinates. The auto-analysis module posts a "skirting" task when an obstacle is detected; the corresponding goal is the projection of the animat's current position onto the line that must be crossed to skirt the obstacle (the projection, and corresponding direction, varies as the animat moves).

The auto-analysis module analyzes the animat's current behaviour to alter the current task and to create planning rules that will improve future behaviour. It is responsible for detecting obstacles, triggering skirting behaviour, and discovering the *salient states* in the environment through which the animat can travel to avoid obstacles efficiently in the future.

Here, an obstacle is defined as "any material element that prevents execution of the best action the robot can perform in order to move in the direction of its current goal" (Donnart & Meyer, 1996, p. 3). If the animat detects a material element, it locates the best reactive rule that could be activated for the current goal direction if it did not detect any material elements. If this rule's move cannot be executed because of the material element, the element qualifies as an obstacle, and a skirting task is sent to the context manager. This process is repeated for the second best rule, etc. Regardless of whether the material element is an obstacle, the animat then chooses a reactive rule that is

satisfied by the actual sensory input and current goal direction (of course, if an obstacle was detected, and a skirting task was posted, the goal will have changed). If a material element prevents the actual action, the rule's strength is set to zero, and candidates from the other matching rules are selected until one with a legal action is found. (In general, a rule with an illegal action has its strength reduced by a fraction; for environments with noisy inputs, this fraction should be less than 1 (Donnart, personal communication).

A skirting subtask is added to the top of the context manager's pile whenever an obstacle is detected. The subtask specifies the line that the animat must cross to circumvent an obstacle. It has the form *<coordinates of the place><direction vector of the straight line to be crossed>*. If a new obstacle is detected, a new skirting subtask is stacked on top of the current one. In the process of trying to achieve the current goal, the animat may achieve a task lower in the pile; in this case, all higher tasks are also erased.

By triggering reactive rules and generating skirting subtasks, the animat is guaranteed to eventually find a path that leads to its goal. This path may be relatively inefficient, however. In order to extract an efficient plan from its trajectory, the auto-analysis module analyzes the path taken and extracts its salient states (landmarks, in the case of a navigational application). The system has a domain-specific heuristic function that is used to evaluate its *internal satisfaction* (the degree to which a rule brought the animat closer to or further from its goal) after each move. The system calculates the difference between the internal satisfactions of successive actions, and identifies a *satisfaction state* wherever this gradient is positive. Then, the process is applied recursively to the imaginary path formed by these satisfaction states until no more states

can be eliminated. The *salient states* are the final set of satisfaction states, and are used to generate planning rules that will allow the animat to skirt obstacles more efficiently in the future.

Finally, there is the internal reinforcement module (perhaps its original name, "the internal retribution module," [Donnart & Meyer, 1994] was deemed too ruthless). Its purpose is to modify the strengths of the rules in the reactive and planning modules. A reactive rule is reinforced whenever it is triggered; the value of the reinforcement depends on the internal satisfaction derived from executing the rule's action:

$$S(R)_{u+1} = (1-\alpha) * S(R)_u + \alpha * satisf$$

$$satisf = \frac{dist\_goal(u) - dist\_goal(u+1) + \max\_dist}{\max\_dist * 2}$$

where $S(R)_u$ is the strength of rule R after u triggers, *dist_goal(u)* is the estimated distance to the current goal (the Euclidean distance, for the test implementation), and α and *max_dist* are constants. If *max_dist* is set to the maximum distance by which a move can bring the animat closer to or further from its goal, *satisf* will always be between 0 and 1.

Rules in the planning module have two strengths, one local and the other global. The local strength estimates the usefulness of decomposing a task into the proposed subtask. Even if this decomposition is useful, however, the rule is not valuable if the task rarely occurs; the global strength is used to suppress rules that are unlikely to be used.

The idea behind the calculation of the local strength is straightforward: the shorter the path to the goal when the planning rule is applied versus when it is not, the higher the rule's strength should be. To calculate the local strength of rule *P1*, *LS(P1)*, that

decomposes a task T into a subtask $T1$, one must first calculate the average cost of all paths tested by the animat that achieve T without decomposition, $AC(T)$:

$$AC(T)_{u+1} = (1-\alpha) * AC(T)_u + \alpha * C$$

where u is the number of times T was achieved without using any planning rules, and C is the cost (number of moves) of the $u+1th$ path. Then, calculate the average cost of all paths that achieve T that use the rule $P1$:

$$AC(P1)_{u+1} = (1-\alpha) * AC(P1)_u + \alpha * C$$

where u is the number of times task T was achieved using planning rule $P1$, and C is again the cost of the $u+1th$ path. It may be observed that the average costs are computed incrementally. These updates occur when task T is achieved; the cost C of the path is used to "reinforce" the average costs of all the planning rules that decomposed T; this

$$LS(P1) = \frac{AC(T)}{AC(P1)}$$

implies the sequence of planning rules must be memorized, and that the update is a form of profit sharing algorithm (Grefenstette, 1988). Finally, the local strength of P1 is computed.

On each time step when the current task is one that was generated by the planning module, the planning module decides whether to decompose the task into a subtask by triggering a matching rule. The decision is a probabilistic choice based on the local

strengths of the eligible rules (the strength for the "no decomposition" option is always 1), weighted by the exploration-exploitation coefficient.

The global strength of *P1* is calculated as follows: $GS(P1) = LS(P1) * GS(T)$.

$GS(T)$ is the average of the global strengths of all rules P that could post T on the context manager's pile:

$$GS(T)_{u+1} = (1-\alpha)*GS(T)_u + \alpha * GS(P)_{v+1}$$

where u is the number of times task T was achieved, and v is the number of times rule P was triggered.

The test environment for the simulation consisted of a square. The animat was to learn how to reach a goal position from a start position. The achievement of its goal marked the end of an iteration. The environment could contain any number of obstacles; e.g., a barrier, a U-shaped dead-end, or a double-spiral. Each experiment had two phases: (1) reactive mode, where the planning rules were learned but not used; and (2) planning mode, where the planning rules were allowed to generate sub-goals.

For each environment (e.g., no obstacles, barrier, dead-end, double-spiral), MonaLysa quickly learned the reactive rules need to skirt any obstacles; performance was good within 200 iterations, and typically far fewer. When the environment was changed every 50 iterations (without resetting the system's rules), the animat quickly adapted, proving that the reactive rules learned in one environment were sufficiently useful in others.

Similar tests against the animat's planning rules proved that it was capable of remembering multiple plans. If the animat learned to skirt a dead end efficiently by travelling on the left, and an obstacle was added to the left side, it quickly learned that it was more efficient to favour a path to the right (which had apparently been learned previously). If yet another obstacle was added to the right side, the animat would soon return to travelling along the left path.

MonaLysa was also tested in the "real world," as the controller for a Khepera robot. The turn angle was changed from 90° to 45° to compensate for the unreliable sensors. The robot could stop and choose another action before it had completed a 5.5 cm "step" in response to changed sensor information. The results were similar to those for the virtual simulation.

In summary, the MonaLysa is capable of rapidly learning how to efficiently navigate around complex obstacles. It avoids the temporal credit assignment problem for both reactive and planning rules. The reactive rules are reinforced immediately after they are triggered, thanks to the system's internal satisfaction function. The planning rules benefit from the hierarchical structure of the tasks and subtasks they implement; when a task is achieved, the reward is used to reinforce every rule that contributed to the result, either directly to the task or to one of the subtasks. This approach of memorizing a behavioural sequence and directly applying a reward to every member is akin to Grefenstette's (1988) profit-sharing plan. For this reason, Donnart and Meyer (1996) call their method a *hierarchical profit-sharing algorithm.*

ALECSYS: Shaping Behaviour

Reinforcement learning tends to be slow. Many iterations are needed before the system learns appropriate behaviour. As the search space grows, so does the learning complexity. In the case of CSs, complex problems usually require large classifier populations, which means that the generation of the match set, rule discovery, etc. take longer.

One approach to improving CS performance is to execute many operations in parallel. The various components of the CS algorithm could become separate processes assigned to different processors. In some cases where a process applies an operation to a group of classifiers, its operation can be applied to each classifier independently (e.g., rule matching). This process can therefore be split into multiple processes, each one assigned to its own processor and responsible for a subset of the classifier group. There is still only one flow of control – for example, the match set must be formed before the bid competition begins – but it is applied to many data simultaneously.

While this form of low-level parallelism can accelerate learning, its scalability is limited. As new nodes are added to a network, the communication overhead grows, so that the speedup is less than linear. Also, it does not reduce the complexity of the problem faced by the CS. Complexity is increased when the CS must solve problems with multiple goals. One solution is decompose the problem by allocating a separate CS to each goal. These CSs may run concurrently, allowing a higher-level parallelism: multiple flows of control. Now, however, there is an additional problem: how will the behaviours of these CSs be coordinated?

ALECSYS is a CS architecture that implements both forms of parallelism, low- and high-level (Dorigo and Sirtori, 1991; Dorigo, 1995). It is the high-level parallelism that is relevant to the study of hierarchical CSs. Like the MonaLysa, it has been tested in both simulated and physical environments. The robot, whether virtual or real, is called an "AutonoMouse." The simulation environment is two-dimensional, and contains three objects: a moving light source, a predator that appears periodically and can only be "heard," and the AutonoMouse's lair (Dorigo, 1995).

In this environment, the AutonoMouse is expected to learn a couple of basic behaviours. First, a "playing" behaviour; the AutonoMouse likes to follow the moving light source. Second, a "hiding" behavior; when the AutonoMouse hears a predator, it should reach its lair as quickly as possible and stay there until it leaves. Then, there is the global behaviour that emerges from coordinating the two basic behaviours; in this case, the desirable behaviour is as follows: "The simulated AutonoMouse plays with (follows) the light source. When it happens to hear a predator, it suddenly gives up playing, runs to the lair and stays there until the predator goes away" (Dorigo, 1995, p. 18).

Two learning architectures were tested: a "monolithic" architecture and a "switch" (or hierarchical) architecture. The monolithic architecture consists of a single low-level parallel CS ("CS-global") spread across three transputers. The switch architecture uses three hierarchically-organized CSs: two CSs ("CS-play" and "CS-hide") to learn the basic behaviours, and a third one ("CS-switch") to learn how to switch between them; one transputer is allocated to each CS. The architectures are illustrated in Figures 6 and 7.

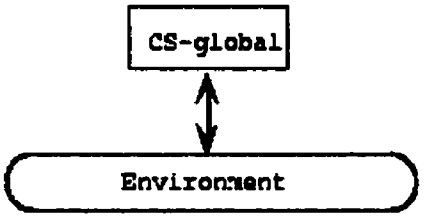

Figure 6: The monolithic architecture (Dorigo, 1995, p. 20).

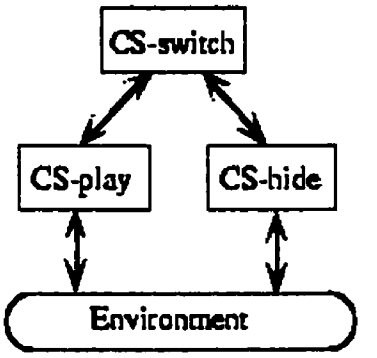

Figure 7: The switch architecture (Dorigo, 1995, p. 20).

Playing behaviour requires knowledge of the light's position and distance. Knowledge of the predator's presence and the position and distance of the lair is needed for the hiding behaviour. For the monolithic architecture, this information is combined as a single sensory message; the classifier conditions have the same length as the messages. An advantage of the switch architecture is that CS-play does not need the predator or lair information and CS-hide does not need to know about the light, so the sensory messages and classifier conditions for each of the CSs are shorter than for CS-global. When classifier for one of the basic CSs triggers a motor action, it also sends a bit to CS-switch; so CS-switch's sensory message has two bits, one for each basic CS. The meaning of these bits is not predefined but learned.

A trainer, or reinforcement program (RP), observes the AutonoMouse and sends reinforcement signals after each action. RP combines RP-play, which rewards the agent if its last move does not increase its distance from the light source, and RP-hide, which rewards the agent if it reduces its distance from the lair (or, if it is already in the lair, stays still). (In some experiments, RP would also punish the agent if its behaviour was

undesirable.) RP uses RP-hide if the predator is present, RP-play otherwise. (It is worth noting that RP plays a role similar to that of the MonaLysa's internal satisfaction function; it encodes special domain knowledge that is used to reinforce the system on every step. It doesn't matter whether the heuristic function is "inside" the animat or in an intelligent critic "outside".)

For the switch architecture, reinforcement is complicated by the need for a *shaping policy*. Two policies were tested, "holistic" and "modular". The holistic policy treats the entire learning system as a black box; each reinforcement is given to all three CSs. The disadvantage of this approach is that the "wrong" reinforcement can be given to a component CS. For example, a correct action can result from two wrong actions, or a CS can be punished because of another CS's mistake.

The modular policy requires a step-wise approach. The basic CSs are trained first, possibly in parallel, until they achieve good performance. Then, they are frozen, and CS-switch is trained. RP-play is used to train CS-play, RP-hide to train CS-hide.

The experiments showed that the switch architecture always outperformed the monolithic one, possibly because its classifiers were shorter, hence its search space was smaller. The modular shaping policy outperformed the holistic shaping policy when both were given comparable computational time.

6. Criteria for an Ideal Hierarchical Classifier System

Now that an overview of several implementations of hierarchical CSs has been presented, it is time to construct a list of desirable attributes, against which each of these control architectures may be compared. The ideal hierarchical CS should: (1) be capable of planning, (2) learn hierarchical relationships dynamically, (3) depend on local versus global information, (4) require minimal domain knowledge, and (5) be robust.

(1) Capable of planning

Planning implies the ability to decompose a task into subtasks. By doing so, classifier chains are shortened; shorter chains are reinforced more quickly, and are less vulnerable to disruption. The system is aware of its current task, and uses this internal context to help decide its next action.

(2) Learn hierarchical relationships dynamically

The hierarchical relationships are not pre-defined by a designer, but are instead learned by the system from accumulated experience. (If the system is capable of planning, these relationships describe how tasks, or behavioural modules, may be decomposed into subtasks.) The value of these relationships are continually re-evaluated, and may evolve over time.

(3) Depend on local versus global information

In an earlier section, the strengths and weaknesses of a global learning scheme (e.g., Grefenstette's profit-sharing plan) were compared with that of a local one (e.g., the bucket brigade algorithm). Global learning schemes offer potentially better performance but require extra memory and computation. Systems that use local

algorithms are most suitable to incremental, real-time learning, and are arguably more "animat-like" in the sense that they are more similar to primitive biological organisms.

(4) Require minimal domain knowledge

Many sequential problems do not offer the system any reward until a final goal is attained. This makes it difficult to determine how "stage-setting" rules should be reinforced. To aid learning, domain knowledge may be added to the architecture, in the form of special operators or a function that computes a reinforcement signal after every action. Obviously, an architecture that does not require such knowledge is more flexible and easier to set up, especially when the domain is ill-understood.

(5) Robust

This attribute may be considered a side-effect of (2). Hierarchical plans should not be brittle; if the environment changes, the system should be capable of adapting existing plans to new circumstances. Also, it should handle noise gracefully.

The true goal of these five characteristics is to improve scalability. Hierarchical decomposition is a means of making learning systems more scalable. Long chains that are difficult to reinforce are broken down into shorter chains that are more manageable. In order to be truly scalable, the depth of the hierarchy should be unbounded; extra hierarchical levels may be added as further decomposition is required. Furthermore, the various tasks or behavioural modules should serve as building blocks for new modules, allowing a module to be re-used by multiple higher-level modules. If these relationships are learned automatically, the human programmer is absolved of the need to hand code an

arbitrarily complex hierarchy. By depending on local versus global information, the space and time complexity required for learning remains manageable for complex problems.

It is now possible to compare each hierarchical CS against the above list of desirable attributes. Weiss's hierarchical chunking algorithm (HCA) and Stolzmann's anticipatory classifier system (ACS) may be considered equivalent for the purpose of the comparison, since they are both reactive CSs that can "chunk" ordinary classifiers to create new rules whose action part is a behavioural sequence. Neither CS is capable of explicit planning; there is no internal context describing the system's current goal. Hierarchies are learned dynamically, though the concept of hierarchy is limited to a flat sequence of motor actions. Both CSs depend solely on local information to create new chunks, as they need remember only the previously active rule. They do not use any special domain knowledge. They are scalable in the sense that larger and larger chunks may be evolved over time, resulting in arbitrarily short classifier chains. On the other hand, in a complex environment with many implicit goals and sub-goals, the number of chunks required to represent every useful action sequence may be prohibitively large.

It is interesting to apply the robustness criterion to HCA and ACS. Because each "hierarchy" is encoded as a simple action sequence, it is unlikely that these CSs could adapt rapidly to a changing environment. Consider a CS that is not limited to a flat hierarchy. Suppose that it needs to accomplish a task T that has been decomposed into subtasks A, B, and C; A is achieved by executing the motor action sequence A_1, A_2, A_3, B via B_1, B_2, B_3, and C via C_1, C_2, C_3. If the environment changes such that A can no

longer be achieved using A_1, A_2, and A_3, but is instead satisfied by executing A_4, A_5, A_6, the system can adapt by modifying the relationships between A and the reactive rules it depends on; the rest of the hierarchy remains intact. Contrast this robustness with that of a CS that depends on a flat hierarchy. A classifier that could be said to represent task T would directly encode the action sequence A_1, A_2, A_3, B_1, B_2, B_3, C_1, C_2, C_3. After the environment changes, the classifier is mostly useless, except as a source of genetic material for new classifiers. Since reinforcement learning generally permits much faster adaptation than rule discovery, the CS will adapt much more slowly than one capable of hierarchical planning. Perhaps the population also includes classifiers that chunk the sub-sequences (e.g., B_1, B_2, B_3), in which case adaptation is accelerated; of course, if a chunk exists for every potentially useful action sequence, there is the risk of a combinatorial explosion.

The robustness problem is even greater if a subtask is depended upon by many tasks. HCA and ACS can "reuse" a subtask only by including a copy of the associated action sequence within every chunk that uses it. If an environment change affects the means by which a subtask is achieved, every chunk that embeds the action sequence that was used to accomplish that subtask becomes invalid. Even if one chunk is "corrected" via the GA or other rule discovery mechanism, the other chunks remain invalid and must be addressed separately. In contrast, for a CS capable of hierarchical planning, the correction of the plans for a subtask automatically benefits all tasks that depend on it. Object-oriented design principles provide a useful analogy; clients should depend upon abstract interfaces rather than concrete implementations. By depending upon

abstractions, implementations may be changed without affecting the clients. If the clients depend upon a concrete implementation, a change forces all of them to recompile (Martin, 1995).

Shu and Shaeffer's hierarchical classifier system (HCS) requires a relation that defines how classifiers are grouped into families. If the relation were chaining, each family would represent a behavioural module. (It is not obvious *how* such a relation would be enforced; Shu and Shaeffer's experiments did not specify any relation.) HCS is not capable of planning, since there is no internal context that indicates the current task; families protect useful relationships, but do not direct the execution of action sequences. Relationships are learned dynamically, but hierarchies are limited to one level; there are only families and classifiers. The architecture could be extended to allow nested families, but these sub-families would not be sharable between families. Thus, families are not reusable building blocks, making HCS's robustness and scalability suspect. HCS does not require global information, nor does it depend on special domain knowledge.

ALECSYS does not learn hierarchical relationships dynamically; instead, the decomposition of goals into sub-goals is done in advance by the designer. Nor is the system capable of planning, for its goals are independent of one another; e.g., the animat is either "playing" or "hiding" at any moment – it never plans to play and then hide in order to accomplish a goal. Learning is strictly local. On the other hand, ALECSYS does depend on special domain knowledge. This knowledge is embedded within the reinforcement program, which determines how to reward the animat after each action. ALECSYS is scalable because the pre-defined hierarchy has an arbitrary depth. The

system was not tested in changing environments, but it has been proven that it can handle noisy real-world environments, even when the robot's sensors and motors are intentionally damaged.

The remaining two architectures, MonaLysa and Wilson's hierarchical CS, are exceptional in that they are both capable of planning. MonaLysa uses its experience to dynamically create hierarchical relationships. Planning rules are reinforced via a hierarchical profit sharing plan; this requires the memorization of extended sequences, hence the learning scheme is global. To adapt MonaLysa to a particular problem requires a significant amount of domain knowledge; the internal satisfaction function and the auto-analysis module are both problem-specific. MonaLysa is both robust and scalable; existing rules are rapidly adapted to a new environment, and tasks may be recursively decomposed into sub-tasks indefinitely.

Wilson's architecture is theoretical; it is not explained how behavioural modules are discovered. Classifiers are reinforced by a hierarchical bucket brigade algorithm; although the hierarchical message list may contain multiple messages, they are all relevant to the current action, so the learning scheme is essentially local. The architecture does not require any special domain knowledge. Although Wilson's CS has never been implemented and tested, its hierarchically structured behavioural modules form natural building blocks that should be easy to adapt to new situations. The possibility for arbitrarily deep hierarchies implies scalability.

7. HXCS: A new hierarchical classifier system

Wilson's hierarchical performance and reinforcement algorithm (Wilson, 1987) has the potential to satisfy most of the criteria that are desirable in a hierarchical CS. It has never been implemented, however. As a consequence, some missing details need to be supplied by the implementer.

For example, Wilson's algorithm assumes that classifiers have two different kinds of messages (actions): external actions, that act directly on the environment, and internal messages, that affect only the CS's internal state. Obviously, the CS must be able to distinguish the two. One approach is to extend messages with "tag bits" that indicate the message type (Riolo, 1990). Another approach is to place classifiers with external actions and classifiers with internal messages in distinct populations (similar to the MonaLysa architecture). The latter approach may be extended further so that classifiers with internal messages are divided into multiple populations, each representing a particular level of abstraction.

Wilson's paper on hierarchical credit allocation was published long before he began studying XCS and accuracy-based fitness. Since XCS has proven superior to traditional, strength-based implementations, it makes sense to update his hierarchical algorithm with XCS-derived modifications. This remainder of this section will describe an implementation of a hierarchical CS, based on both the hierarchical credit allocation algorithm and on XCS, named "HXCS."

The concept of "level" is fundamental to the definition of HXCS. The CS contains multiple populations, including a reactive population (the lowest level) and an arbitrary

number of planning populations. The reactive population is Level 0, the first planning population is Level 1, the second planning population is Level 2, etc. The level of an action set is the same as that of the population from which it was formed. The level of a message (internal action) is the same as that of its action set.

Classifiers in the reactive population have the form <internal message><sensory input>:<external action>, interpreted as, "if <sensory input> matches the current sensory input and <internal message> matches the Level 1 message currently on the list, execute <external action>." Planning classifiers have the form <internal message><sensory input>:<internal message>, which means, "if <sensory input> matches the current sensory input and <internal message> matches the message one level above my own, post <internal message> to the list." A sensory input is sometimes referred as an "environmental input," "environmental state," "environmental message," or a "detector message." External actions are also called "effector messages," "motor actions," or simply "actions." Internal messages are sometimes called "internal actions" or "messages."

The following is a step-by-step description of performance and reinforcement algorithm for HXCS:

1. Obtain the current message E from the environmental input interface.
2. If phase = "descent", for each population, form the match set [M] of all classifiers which match both E and the last message in the list, else

If phase = "ascent", for each population, form the match set [M] of all classifiers which match both E and the message at the level one step above that of the population.

3. If phase = "descent," compute the prediction array for the match set of the active population and select the winning action (the selection policy is unspecified; it could be "roulette wheel," "pure explore," or "pure exploit") .

   If phase = "ascent," compute the prediction array for the match set of each population. Select an action from each prediction array. Form a new array by pooling the selected actions; the system prediction for each action equals the maximum system prediction of its population's match set. Select the final winner from this array.

   Let A* be the winning action.

4. If phase = "ascent", update all action sets for messages lower on the list than the message that satisfied the match set for A*'s population, erase those lower messages, and update the action set for the action that was carried out on the previous time-step.

5. If A* is an external action, set phase = "ascent",

   Else post the message on the next lower empty level of the message list and set phase = "descent".

6. If A* was an external action, take it.

7. If payoff *R* is received from the environment, update all action sets for messages now on the list, erase all messages, update the action set whose action was just taken, and set phase = "descent".
8. Return to step 1.

Action sets are updated as in XCS. The payoff used to update the classifier predictions is the external reward if one was received. Otherwise, the payoff is the discounted maximum system prediction. During "descent," the maximum is selected from the prediction array for the active population. During "ascent," when there may be multiple competing populations, the maximum is selected from the pooled values of *all* the prediction arrays.

The genetic algorithm also executes as in XCS, in the winning action set, either for the current cycle (if a reward was received and the episode has ended) or the previous cycle. The GA is not used in the experiments described in the following section.

There are a couple of important differences between HXCS and Wilson's original algorithm. First, there are multiple populations rather than a single one. Only classifiers from the population one level below that of a given message may match that message, whereas Wilson allowed any classifier to be satisfied by a message. This meant that a classifier could participate at multiple levels of abstraction. It is hypothesized that allowing each classifier to participate at only one level of abstraction leads to a more stable solution. XCS is an improvement over earlier CSs partly because it tends to suppress classifiers that belong to multiple niches with differing payoffs, and tends to evolve classifiers that accurately map the payoff landscape. It is plausible that preventing

classifiers from belonging to niches at different hierarchical levels yields a similar benefit.

The other major difference between HXCS and the original hierarchical allocation algorithm is that the former is based on XCS instead of the traditional strength-based CSs. Action selection, parameter updates, and rule discovery are all performed as in XCS. In the original algorithm, the winner of the selection process was an individual classifier; in HXCS, as in XCS, the winner is an action. Similarly, parameter updates are applied to action sets instead of a single classifier.

8. Experiments

HXCS's hierarchical message list provides internal state, making it possible to learn an optimal policy for non-Markov environments. Recent experiments with CSs have tested this ability using "woods" environments. A woods is a grid-based environment; a cell may be empty, or contain an obstacle (a "tree" or "rock") or food. The virtual organism that navigates the woods is called an "animat." The first experiments with woods were run by Wilson (1985), who used a simple, memory-less CS.

One example of a non-Markov woods is Woods101, more generally known as "McCallum's Maze" (Lanzi, 1998), as shown in Figure 8. A "T" represents a "tree" (obstacle) and the "F" indicates food (the goal). The animat is aware of the eight adjacent cells and nothing else, which is why the two cells indicated in the figure are indistinguishable. The animat's detectors encode a blank cell as "00," a tree cell as "01," and the food cell as "11." Each detector message is a concatenation of the codes for all adjacent cells, beginning with cell to the north and proceeding clockwise. For example, the detector message for the two aliasing positions is "0101000001000001." On each time step, the animat may move to any adjacent cell that does not contain a tree. The eight possible actions are encoded as three-digit binary strings, with "000" representing a move to the north, "001" a move to the north-east, etc.

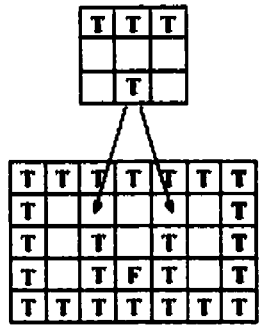

Figure 8: The Woods101 environment. Aliasing positions are indicated by the arrows. (Lanzi, 1998)

HXCS's architecture allows for an arbitrary number of planning populations, but only one is used in the following experiments; this means that the message list contains a single slot. The genetic algorithm is disabled; the intent is to test the performance and reinforcement algorithm without the added complication of rule discovery. The reactive and planning populations are initialized with a set of classifiers covering every possible combination of input and action. Although a classifier condition is encoded as a ternary string, it is more intuitive to describe the part that matches the current sensory input in terms of the grid positions it matches. Figure 9 assigns an alphabetic label to every open position in Woods101.

T	T	T	T	T	T	T
T	*c*	*d*	*e*	*f*	*g*	T
T	*b*	T	*j*	T	*h*	T
T	*a*	T	F	T	*i*	T
T	T	T	T	T	T	T

Figure 9: The Woods101 environment, with labels assigned to the open positions.

Since there are two aliasing positions in Woods101 that need to be distinguished, internal messages need by only a single bit. The conditions of the planning and reactive classifiers used in the experiments are listed in Table 1. Internal messages being one bit, the number of planning classifiers is 8 (conditions) * 2 (actions) = 16. There are eight

external actions, so the number of reactive classifiers is 9 (conditions) * 8 (actions) = 72. Note that the condition of a reactive classifier has the form <internal message>,<sensory input>. All the reactive classifiers ignore the internal message, save those that match the aliasing positions. Note that because there is one classifier per combination of sensory input and action, every action set will contain exactly one classifier.

Table 1: Planning and reactive classifier condition used with HXCS in Woods101. The letters correspond to the labels in Figure 9. "a | i" means that the condition matches both a and i.

Planning classifier conditions	Reactive classifier conditions
a | i	#, a | i
b	#, b
c	#, c
d | f	0, d | f
e	1, d | f
g	#, e
h	#, g
j	#, h
	#, j

At the start of each problem, the animat is moved to a random open position; the problem ends once the food is attained. Upon reaching the food, the system receives a reward of 1000; all other moves have a reward of zero. Each experiment consists of two periods, a learning period and a test period. The learning period covers the first 13,000 problems, 6500 in explore mode and 6500 in exploit mode; the system alternates between the two modes. During exploitation, the action with the largest system prediction is always chosen from the list of candidates. During exploration, the action is chosen completely at random with probability p_s, deterministically as per exploitation otherwise. (Normally, the two modes are also distinguished by the possible application of the

genetic algorithm during exploration, but this distinction is not relevant here since rule discovery is disabled.) The test period consists of 2500 problems, all in exploit mode.

The parameters used are listed in Table 2.. The probability of selecting an action at random during exploration, p_s, is 1.0 for both the planning and reactive levels unless otherwise indicated. Performance is reported as a moving average of the number of steps taken to reach the food over the last 50 exploitation problems. Unless specified otherwise, all results are averaged over ten experiments.

Table 2: Parameters used in the experiments. For more information on their use, see Butz and Wilson (2000).

Parameter	Description	Value
β	Learning rate	0.2
γ	Discount factor	0.71
ε_0	Error threshold (accuracy criterion)	0.01
P_I	Initial prediction	10.0
ε_I	Initial prediction error	0.0
F_I	Initial fitness	0.01

Before presenting the results of experiments with HXCS, however, it is important to present the results for an unmodified XCS, using the same parameters listed above. The only difference, other than the non-hierarchical architecture, is that the conditions of the (reactive) classifiers are not matched against internal messages. For this reason, the population contains eight distinct conditions, hence 64 macroclassifiers. Of course, XCS cannot possibly learn an optimal policy, since it has no internal state to disambiguate the aliasing positions. XCS's performance in Woods101 is shown in Figure 10.

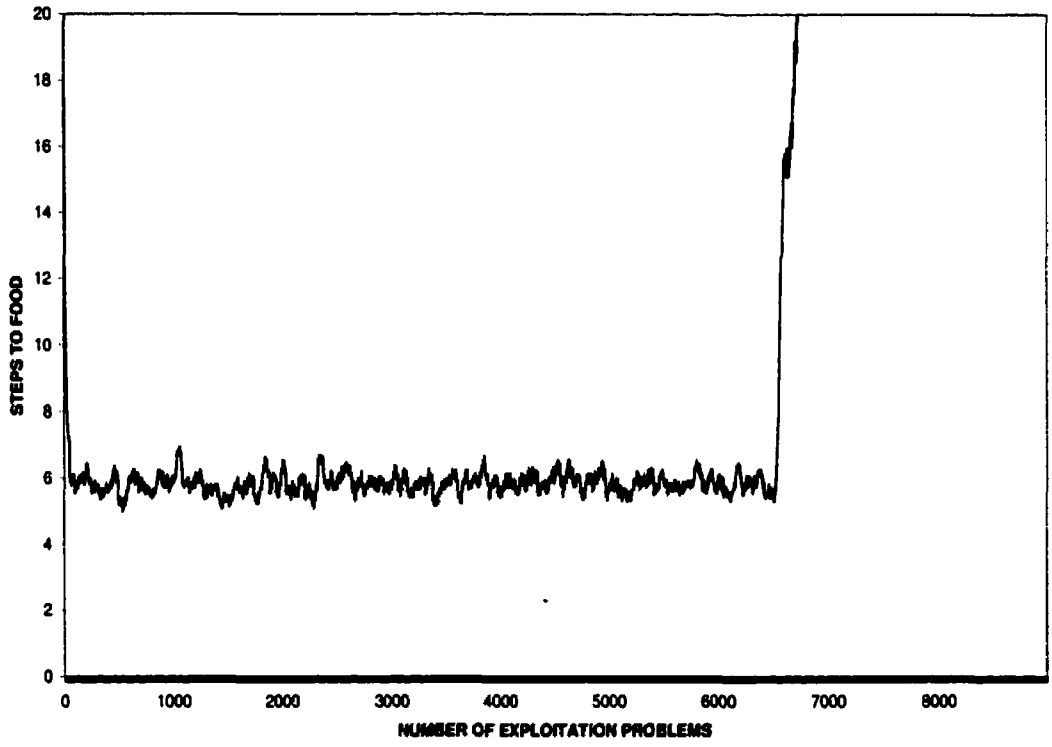

Figure 10: XCS in Woods101

XCS's performance during the test period (starting at 6500 exploitation problems) is somewhat surprising. The test period, when the system always exploits existing knowledge, is normally an opportunity for performance to improve and stabilize. Instead, the number of steps taken to reach the goal increases dramatically. By the end of the test period, the average number of steps is 278!

Inspection of the classifier population before and after the onset of the test period suggests the reason for the dramatic change. As has been mentioned previously, XCS has no means of distinguishing between the aliasing positions. During the test period, the action with the highest system prediction is always selected. This means that, at any given moment, the system is able to reach the food from only one of the two aliasing

positions. Should the animat find itself at the other aliasing position, it will waver back and forth until reinforcement alters the system predictions to make it possible to move past the aliasing position to the food. Now, however, the animat will no longer be capable of reaching the food from the first aliasing position – until another lengthy period of reinforcement tips the balance the other way, etc.

If XCS had been allowed to execute normally, with the genetic algorithm and generalization enabled, its solution would be more robust. With a much more varied pool of classifiers to draw upon, it could maintain solutions for both aliasing positions concurrently. The poor performance during the test period is problematic, making the experiment's usefulness as a baseline dubious.

The optimal average number of steps to the food is 2.90. Since XCS's average number of steps is never better than 4.99, and frequently exceeds 6.00, there is obviously room for improvement.

The results for HXCS in Woods101 are shown in Figure 11. HXCS can take different kinds of action: a planning step, involving the posting of an internal message to the list; and a reactive step, involving the execution of an external action. The performance for the planning and reactive levels are both graphed, along with their cumulative totals. At the end of the test period, the average number of planning steps is 2.91, and the average number of reactive steps is 4.22, for a total of 7.13 steps. The appropriate weighting for the planning and reactive steps is highly context dependent. If the goal is to minimize the number of decisions, internal or external, then it makes sense to add the two values together. On the other hand, if the executing a motor action is

significantly more costly than posting an internal message (e.g., the animat is a physical robot, or part of a simulation where moves have an energetic cost), the number of planning steps may be mostly ignored.

Curiously, during the test period, the number of reactive steps decreases while the number of planning steps *increases*. Examination of the classifiers before and after the onset of training reveals that the latter phenomenon is caused by the strengthening of planning rules matching positions *e* and *j* – cells one and two steps, respectively, from the food. Positions *d* and *f* are also two steps from the food, but the classifiers that match them are not accurate because of confusion caused by the aliasing. The classifier populations for the best and worst experiments are listed in Tables 3 and 4.

During the ascent phase, the reactive and planning populations compete to execute their decisions. The test period allows the prediction of one of the planning classifiers matching *e* and one matching *j* to converge to 1000. This is desirable, since once a planning classifier matching one of these positions has posted its message, no more planning decisions need be executed before the goal is reached; the classifier should therefore reliably receive the full reward of 1000. Once this convergence has occurred, however, these planning classifiers are able to compete with the reactive classifiers corresponding to the same positions. If the animat begins at *e* or *j*, one of the planning classifier matching this position should post its message. However, if the animat begins at another position and moves to *e* or *j*, these planning classifiers should not post any messages; the internal messages are useful only as a means of distinguishing the aliasing positions, which have been encountered already. Thus, during testing, the number of

planning steps is inflated because planning classifiers matching *e* and *j* post their messages even when they do not aid performance.

Table 3: Planning and reactive classifiers for HXCS with best performance in Woods101

Cond.	Msg.	Pred.	Error
a|i	0	145.26	43.18
a|i	1	357.91	0.00
b	0	504.10	0.00
b	1	158.72	29.30
c	0	108.56	14.19
c	1	1000.00	0.00
d|f	0	710.00	0.00
d|f	1	186.21	86.80
e	0	259.03	127.32
e	1	1000.00	0.00
g	0	504.10	0.00
g	1	208.80	109.47
h	0	504.10	0.00
h	1	186.35	14.37
j	0	382.84	301.61
j	1	1000.00	0.00

Cond.	Action	Pred.	Error	Cond.	Action	Pred.	Error
#, a|i	000	215.63	4.21	1, d|f	100	196.90	65.68
#, a|i	001	143.62	29.02	1, d|f	101	199.39	100.64
#, a|i	010	145.06	29.36	1, d|f	110	196.73	130.21
#, a|i	011	142.01	28.57	1, d|f	111	189.62	62.84
#, a|i	100	147.18	28.16	#, e	000	504.10	0.00
#, a|i	101	140.28	28.05	#, e	001	504.10	0.00
#, a|i	110	142.94	28.73	#, e	010	374.18	30.21
#, a|i	111	142.46	28.94	#, e	011	504.10	0.00
#, b	000	185.31	55.83	#, e	100	710.00	0.00
#, b	001	301.17	20.15	#, e	101	504.10	0.00
#, b	010	188.01	45.84	#, e	110	350.91	29.95
#, b	011	187.36	44.97	#, e	111	504.10	0.00
#, b	100	185.57	6.34	#, g	000	190.53	46.67
#, b	101	185.03	46.50	#, g	001	184.30	46.26
#, b	110	186.08	45.41	#, g	010	194.88	46.79
#, b	111	186.95	45.82	#, g	011	182.22	45.88
#, c	000	156.45	47.07	#, g	100	197.64	41.38
#, c	001	172.74	47.30	#, g	101	187.02	46.99
#, c	010	504.10	0.00	#, g	110	304.44	22.43

#, c	011	155.93	46.58	#, g	111	184.99	46.00
#, c	100	173.48	24.62	#, h	000	189.13	44.39
#, c	101	168.70	47.74	#, h	001	195.19	41.58
#, c	110	173.55	47.44	#, h	010	187.33	37.67
#, c	111	155.73	46.47	#, h	011	189.81	37.62
0, d | f	000	210.26	64.48	#, h	100	177.78	6.31
0, d | f	001	218.00	65.66	#, h	101	179.22	46.28
0, d | f	010	215.62	131.16	#, h	110	177.74	46.65
0, d | f	011	222.32	169.98	#, h	111	307.79	20.79
0, d | f	100	220.46	69.74	#, j	000	504.10	0.00
0, d | f	101	218.21	169.03	#, j	001	378.38	28.74
0, d | f	110	424.79	79.18	#, j	010	710.00	0.00
0, d | f	111	216.56	66.59	#, j	011	710.00	0.00
1, d | f	000	208.18	65.08	#, j	100	1000.00	0.00
1, d | f	001	204.94	65.41	#, j	101	710.00	0.00
1, d | f	010	208.88	114.62	#, j	110	710.00	0.00
1, d | f	011	710.00	0.00	#, j	111	373.36	23.42

Table 4: Planning and reactive classifiers for HXCS with worst performance in Woods101

Cond.	Msg.	Pred.	Error
a | i	0	28.89	11.63
a | i	1	375.24	76.55
b	0	536.61	151.39
b	1	57.48	29.65
c	0	394.78	43.19
c	1	55.48	16.32
d | f	0	119.81	44.29
d | f	1	493.64	242.81
e	0	1000.00	0.00
e	1	123.48	37.09
g	0	125.10	58.89
g	1	332.29	51.60
h	0	368.91	121.65
h	1	83.13	28.00
j	0	1000.00	0.00
j	1	212.94	159.67

Cond.	Action	Pred.	Error	Cond.	Action	Pred.	Error
#, a | i	000	202.78	14.05	1, d | f	100	163.43	56.65
#, a | i	001	115.18	31.66	1, d | f	101	303.31	189.96
#, a | i	010	114.44	31.59	1, d | f	110	163.41	75.63
#, a | i	011	116.05	31.78	1, d | f	111	164.20	56.78
#, a | i	100	116.26	31.96	#, e	000	504.10	0.00

#, a | i	101	113.38	31.71	#, e	001	504.10	0.00
#, a | i	110	115.49	31.70	#, e	010	392.27	37.55
#, a | i	111	115.83	31.83	#, e	011	504.10	0.00
#, b	000	152.21	31.57	#, e	100	710.00	0.00
#, b	001	302.15	53.85	#, e	101	504.10	0.00
#, b	010	151.49	46.29	#, e	110	397.12	19.54
#, b	011	150.05	46.14	#, e	111	504.10	0.00
#, b	100	157.11	22.09	#, g	000	154.83	47.79
#, b	101	147.27	44.86	#, g	001	140.57	45.33
#, b	110	143.61	43.69	#, g	010	143.34	45.76
#, b	111	148.56	44.96	#, g	011	142.76	45.77
#, c	000	163.45	49.06	#, g	100	152.96	38.67
#, c	001	162.36	49.32	#, g	101	143.12	45.63
#, c	010	162.33	41.38	#, g	110	205.93	40.07
#, c	011	163.68	49.04	#, g	111	141.83	45.55
#, c	100	197.38	18.61	#, h	000	144.08	47.27
#, c	101	163.60	49.18	#, h	001	142.80	46.04
#, c	110	163.12	49.55	#, h	010	143.43	46.05
#, c	111	159.85	50.33	#, h	011	142.12	45.42
0, d | f	000	207.69	68.05	#, h	100	141.37	37.14
0, d | f	001	213.59	71.34	#, h	101	139.57	45.82
0, d | f	010	497.19	30.30	#, h	110	141.57	45.57
0, d | f	011	218.59	166.78	#, h	111	287.27	55.59
0, d | f	100	221.60	69.18	#, j	000	504.10	0.00
0, d | f	101	217.31	186.90	#, j	001	409.01	26.22
0, d | f	110	216.27	126.74	#, j	010	710.00	0.00
0, d | f	111	218.99	70.16	#, j	011	710.00	0.00
1, d | f	000	161.25	57.35	#, j	100	1000.00	0.00
1, d | f	001	165.79	57.51	#, j	101	710.00	0.00
1, d | f	010	165.88	116.03	#, j	110	710.00	0.00
1, d | f	011	171.58	138.26	#, j	111	397.69	23.92

It is worth noting that only one of the two planning classifiers *e* and one of the two matching *j* predict a payoff of 1000. Logically, the classifiers for both messages, 0 and 1, should share the same prediction, since the internal state is irrelevant once the animat has passed through the aliasing positions.

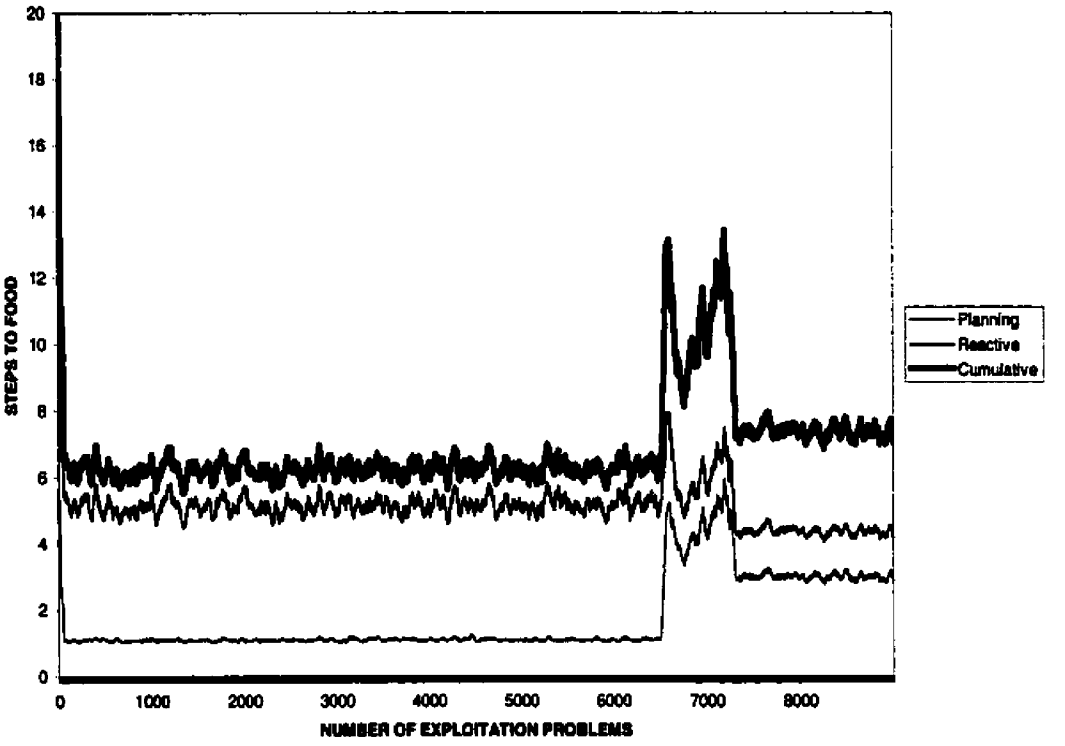

Figure 11: Performance of HXCS in Woods101.

There is yet another problem. At the beginning of a problem, when the animat is randomly moved to an open cell, its start position may be one of the aliasing positions, *d* or *f*. Since the animat has not yet had encountered any cues that could disambiguate the aliasing, it cannot possibly make an optimal decision reliably. In fact, though it has been mentioned that the optimal number of reactive steps is 2.90, this is not entirely true. If the animat always travelled the shortest distance between its start position and the food, the average number of steps would be 2.70. But because an animat that begins a problem at an aliasing cell cannot know the correct direction, an extra step is required for it to orient itself when it starts at *d* or *f*.

This limitation means that the planning classifiers matching *d* and *f* will never be able to distinguish the aliasing states, and this handicap affects the reactive classifiers that depend on them either directly or indirectly. Looking at the sample classifier populations, one notes that most of the reactive classifiers matching positions at or preceding *d* or *f* are inaccurate.

But there are two distinct scenarios involving the aliasing states: (1) the animat has started at an aliasing state, and (2) the animat is passing *through* an aliasing state. In the former scenario, the animat cannot reliably make the correct decision, but in the latter it can because it will have encountered cues that indicate whether it is travelling along the left or right corridor. If distinct classifiers were used for each scenario, the inevitable uncertainty of scenario (1) need not affect the accuracy of classifiers used in scenario (2).

One solution is to add an extra bit to the detector messages that is on if the animat has not yet moved from its start position and off otherwise. (Alternatively, the bit could be part of the animat's internal state.) This "start position bit" is not useful save for classifiers matching *d* and *f*. The revised planning and reactive classifier conditions are shown in Table 5. The planning and reactive populations now contain 18 and 88 classifiers, respectively.

Table 5: Planning and reactive classifier conditions for Woods101 with "start position" detector bit.

Planning classifier conditions	Reactive classifier conditions
#, a | i	#, #, a | i
#, b	#, #, b
#, c	#, #, c
0, d | f	0, 0, d | f
1, d | f	0, 1, d | f
#, e	1, 0, d | f
#, g	1, 1, d | f
#, h	#, #, e
#, j	#, #, g
	#, #, h
	#, #, j

The experimental results for HXCS with the start position detector bit are shown in Figure 12. The performance is significantly better than for HXCS without the extra bit; by the end of the test period, the average number of planning and reactive steps are 1.83 and 3.54, respectively, for a total of 5.37 steps. As before, the number of planning steps increases during the test period. In one of the ten experiments, optimal performance was attained: 1.24 planning steps and 2.72 reactive steps (actually, 2.72 is *better* than the optimum of 2.90; this is obviously impossible, and is probably due to sampling error). The final population for this experiment is listed in Table 6.

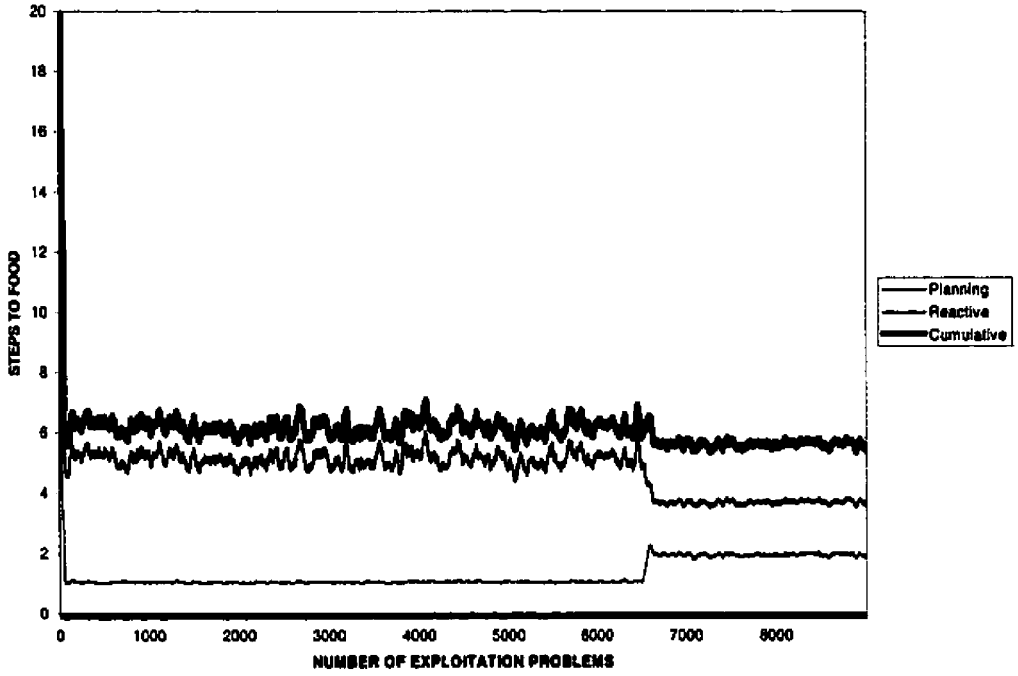

Figure 12: Performance of HXCS with start position bit in Woods101.

Table 6: Planning and reactive classifiers for HXCS with start position bit with best performance

Condition	Message	Prediction	Error
#, a | i	0	710.00	0.00
#, a | i	1	114.90	45.24
#, b	0	117.89	29.26
#, b	1	1000.00	0.00
#, c	0	140.74	96.51
#, c	1	1000.00	0.00
0, d | f	0	205.92	94.20
0, d | f	1	158.11	56.93
1, d | f	0	218.23	163.86
1, d | f	1	873.17	155.79
#, e	0	1000.00	0.00
#, e	1	130.04	43.89
#, g	0	1000.00	0.00
#, g	1	135.53	34.93
#, h	0	1000.00	0.00
#, h	1	161.90	34.72
#, j	0	1000.00	0.00
#, j	1	283.22	91.97

Condition	Action	Prediction	Error	Condition	Action	Prediction	Error
#, #, a | i	000	357.91	0.00	1, 0, d | f	100	413.64	24.52
#, #, a | i	001	200.95	2.36	1, 0, d | f	101	490.10	229.12
#, #, a | i	010	203.18	2.30	1, 0, d | f	110	369.22	111.75
#, #, a | i	011	201.46	2.15	1, 0, d | f	111	420.47	18.47
#, #, a | i	100	200.51	3.20	1, 1, d | f	000	366.39	55.33
#, #, a | i	101	200.29	1.97	1, 1, d | f	001	358.95	33.73
#, #, a | i	110	202.05	2.36	1, 1, d | f	010	391.42	111.44
#, #, a | i	111	202.20	3.21	1, 1, d | f	011	392.12	98.24
#, #, b	000	287.36	8.52	1, 1, d | f	100	379.89	37.22
#, #, b	001	504.10	0.00	1, 1, d | f	101	556.02	189.14
#, #, b	010	284.62	6.09	1, 1, d | f	110	365.38	142.18
#, #, b	011	287.29	5.85	1, 1, d | f	111	351.87	37.85
#, #, b	100	201.43	3.44	#, #, e	000	504.10	0.00
#, #, b	101	288.83	5.23	#, #, e	001	504.10	0.00
#, #, b	110	284.65	6.24	#, #, e	010	402.77	26.74
#, #, b	111	283.75	8.13	#, #, e	011	504.10	0.00
#, #, c	000	287.05	6.95	#, #, e	100	710.00	0.00
#, #, c	001	287.12	7.41	#, #, e	101	504.10	0.00
#, #, c	010	504.10	0.00	#, #, e	110	405.81	38.14
#, #, c	011	284.56	10.67	#, #, e	111	504.10	0.00
#, #, c	100	285.70	7.12	#, #, g	000	279.02	6.22
#, #, c	101	284.73	8.97	#, #, g	001	282.16	6.13
#, #, c	110	286.22	6.82	#, #, g	010	278.78	11.51
#, #, c	111	285.57	5.74	#, #, g	011	285.37	6.41
0, 0, d | f	000	372.18	58.96	#, #, g	100	280.77	6.97
0, 0, d | f	001	382.15	26.74	#, #, g	101	281.70	7.74
0, 0, d | f	010	369.73	123.46	#, #, g	110	504.10	0.00
0, 0, d | f	011	377.18	190.43	#, #, g	111	283.72	8.91
0, 0, d | f	100	353.11	55.19	#, #, h	000	281.00	7.24
0, 0, d | f	101	710.00	0.00	#, #, h	001	286.01	5.27
0, 0, d | f	110	363.46	120.45	#, #, h	010	281.10	10.08
0, 0, d | f	111	365.06	61.02	#, #, h	011	281.71	8.47
0, 1, d | f	000	357.32	39.37	#, #, h	100	199.58	3.58
0, 1, d | f	001	369.21	43.70	#, #, h	101	279.63	6.55
0, 1, d | f	010	434.06	112.72	#, #, h	110	282.76	6.73
0, 1, d | f	011	522.06	238.25	#, #, h	111	504.10	0.00
0, 1, d | f	100	374.27	36.86	#, #, j	000	504.10	0.00
0, 1, d | f	101	489.62	234.20	#, #, j	001	413.46	29.96
0, 1, d | f	110	374.82	138.20	#, #, j	010	710.00	0.00
0, 1, d | f	111	404.62	52.64	#, #, j	011	710.00	0.00
1, 0, d | f	000	411.35	33.33	#, #, j	100	1000.00	0.00
1, 0, d | f	001	401.52	25.22	#, #, j	101	710.00	0.00
1, 0, d | f	010	466.15	72.79	#, #, j	110	710.00	0.00

1, 0, d | f	011	710.00	0.00	#, #, j	111	414.70	41.68

The number of *reactive* steps for the best experiment is optimal, but what of the number of planning steps? The optimal number of planning steps has not yet been mentioned. In part, this is because it is difficult to specify. If the animat begins at *a* or *i*, it cannot yet know which corridor it is travelling. Once it moves forward a step and reaches *b* or *h*, it is possible to post an internal message that identifies the corridor. So, from *a* or *i*, two planning steps are needed. Positions *b*, *c*, *g*, and *h* may all be used to identify the corridor, so only one planning step is needed. If the animat starts at one of the aliasing states, *d* and *f*, it cannot reliably make the correct decision, so as soon as it moves a (reactive) step, another, more accurate planning classifier can post a message. Thus, from *d* or *f*, two planning steps are required. Positions *e* or *j* are encountered after the aliasing positions, so the internal messages serve no useful function; only one planning step is required for an animat that starts there. The average number of planning steps for the optimum is therefore 1.4.

Or perhaps not. When the animat starts at *d* or *f*, it will make a good decision half the time, so no extra planning step is required. In this case, the optimal solution uses 1.3 planning steps on average. But the animat will also guess the "correct" internal message half the time at *a* or *i*, so the average number of planning steps for an optimal policy is actually 1.2. The number of planning steps for the best experiment is 1.24, which is very close to this ideal value.

Note that the optimum number of planning steps is not a minimum like the optimum number of reactive steps. Unlike a reactive step, a planning step is abstract and

is not subject to physical constraints. Thus, it is always possible for a problem to be solved with only one planning step, since the lifetime of an internal message can span any number of reactive steps.

An example of how the classifiers in Table 6 are used to solve a problem is useful both to illustrate the algorithm and to better interpret the classifier parameters. The following example is drawn from an actual execution trace. The system is in exploit mode, so the action with the action (or message) with the highest system prediction is always chosen.

1. The animat begins at cell *a*. Phase = "descent". The message list is empty. The start position bit is on. The initial match set includes two classifiers: (1) #, *a* | *i* : 0 and (2) #, *a* | *i* :1. Their system predictions are 710.00 and 114.90, respectively. (Since there is only one classifier per combination of input and message, the system prediction equals that classifier's prediction.) Classifier (1) has the highest system prediction, so its message (0) is posted to the list
2. No external action has been executed yet, so the animat is still at *a*. Since phase = "descent" and a planning step has already been taken, the match set includes reactive classifiers matching both *a*, the internal message of 0, and the start position bit (which is still on). The list of matches and their predictions are:

   (1) #, #, a | i : 000; P = 710.00

   (2) #, #, a | i : 001; P = 199.14

   (3) #, #, a | i : 010; P = 194.88

   (4) #, #, a | i : 011; P = 197.52

(5) #, #, a | i : 100; P = 196.19

(6) #, #, a | i : 101; P = 198.15

(7) #, #, a | i : 110; P = 200.60

(8) #, #, a | i : 111; P = 197.53

Classifier (1) has the highest prediction, which means its action has the highest system prediction, so action 000 ("move one step to the north") is executed. The animat moves to cell *b*, phase = "ascent," and the start position bit is turned off.

3. Because phase = "ascent," actions are chosen from both the planning and reactive populations. The match set for the planning population includes: (1) #, b : 0 (P = 1000.00 and (2) #, b : 1 (P = 265.68). Classifier (2) has the highest prediction, so its message is chosen at the planning level. At the reactive level, the candidates are:

(1) #, #, b : 000; P = 271.53

(2) #, #, b : 001; P = 504.10

(3) #, #, b : 010; P = 273.25

(4) #, #, b : 011; P = 277.44

(5) #, #, b : 100; P = 202.95

(6) #, #, b : 101; P = 275.22

(7) #, #, b : 110; P = 272.99

(8) #, #, b : 111; P = 274.45

Classifier (2) has the largest prediction, so its action wins at the reactive level. Next, the maximum system predictions for the planning and reactive levels are compared. Since 1000.00 is greater than 504.10, the planning level wins, and the internal

message of 0 is posted to the list, replacing the previous message (also 0). The classifier that posted the original message (#, *a* | *i* : 0), has its prediction updated with the discounted maximum system prediction – that is, 1000.00 * 0.71 (the discount factor) = 710.00. Since the prediction had already converged this value, there is no change. The same is true for the reactive classifier responsible for last motor action (#, #, a | i : 000). The phase is changed to "descent."

4. The competition is limited to the reactive population. The match set contains:

   (1) #, #, *b* : 000; P = 271.53

   (2) #, #, *b* : 001; P = 504.10

   (3) #, #, *b* : 010; P = 273.25

   (4) #, #, *b* : 011; P = 277.44

   (5) #, #, *b* : 100; P = 202.95

   (6) #, #, *b* : 101; P = 275.22

   (7) #, #, *b* : 110; P = 272.99

   (8) #, #, *b* : 111; P = 274.45

   Action 001 is executed, moving the animat one step north-east to cell *d*. The phase is changed to "ascent".

5. The competition includes both the planning and reactive populations. The candidates at the planning level are (1) 0, d | f : 0 (P = 380.89) and (2) 0, d | f : 1 (P = 372.76). The match set for the reactive level includes those rules that are satisfied by the current state (*d* or *f*), internal message (1), and start position bit (off = 0):

   (1) 0, 1, d | f : 000; P = 381.67

(2) 0, 1, d | f : 001; P = 347.60

(3) 0, 1, d | f : 010; P = 374.58

(4) 0, 1, d | f : 011; P = 710.00

(5) 0, 1, d | f : 100; P = 352.38

(6) 0, 1, d | f : 101; P = 368.32

(7) 0, 1, d | f : 110; P = 364.44

(8) 0, 1, d | f : 111; P = 355.16

Classifier (4) has the highest prediction, so action 011 is executed, moving the animat south-east to cell *j*. Because a motor action was executed, the phase remains "ascent." The classifier that executed the previous motor action (#, #, *b* : 001) is reinforced with (4)'s discounted payoff (0.71 * 710.00 = 504.10), with no effect since the target classifier's prediction has already converged to this value.

6. The match set at the planning level includes: (1) #, *j*: 0 (P = 574.05), and (2) #, *j* : 1 (P = 1000.00). Classifier (1)'s action wins. The match set at the reactive level is:

(1) #, #, *j*: 000; P = 504.10

(2) #, #, *j*: 001; P = 413.67

(3) #, #, *j*: 010; P = 710.00

(4) #, #, *j*: 011; P = 710.00

(5) #, #, *j*: 100; P = 1000.00

(6) #, #, *j*: 101; P = 710.00

(7) #, #, *j*: 110; P = 710.00

(8) #, #, *j*: 111; P = 409.28

Thus, action 100 ("move south") wins at the reactive level. The maximum system prediction for the planning and reactive levels are both 1000.00. With the current implementation, ties are resolved in favour of the planning level. So the internal message is changed to 1. The phase becomes "descent."

7. The match set includes the same reactive classifiers as in the previous step. Action 100 is executed, moving the animat to the food and ending the problem. The planning classifier, #, *j* : 1, that posted the current message, and the reactive classifier, #, #, *j* : 100, that executed the last motor action, are both reinforced with the external reward of 1000.

In the above example, the classifiers with the highest predictions tend to also have the lowest prediction errors. An accurate classifier's prediction reflects the number of steps (planning or reactive) that fall between the activation of its message or action and the receipt of an external reward. So, a reactive classifier whose action immediately brings the animat to the food, or a planning classifier whose message is still on the list when food is found, has a prediction equal to the external reward of 1000. If the classifier is separated from the reward by one step, its prediction is $1000 * 0.71 = 710$. If the distance is two steps, the prediction converges to $1000 * 0.71^2 = 504.1$, etc. These values apply only to accurate classifiers.

In examining the planning classifiers for states *b* and *c*, one notices that the rules for message 0 are accurate with predictions of 1000, while the rules for message 1 has low predictions and poor accuracy. The reverse is true for the planning classifiers matching *g* and *h*. HXCS has learned to associate 0 with the west corridor and 1 with the

east corridor. This information is used by the reactive classifiers matching *d* and *f* to disambiguate the aliasing.

Classifier "#, #, a | i : 000," has a prediction of 710, indicating that after its action is executed, the animat is one step from the reward. In fact, the animat is three (reactive) steps from the food. However, on the step that follows its motor action, the planning classifier "#, b : 0" posts its message, and its prediction of 1000 is used to reinforce the reactive classifier whose action enabled it to become active. One could say that "#, #, a | i : 000" is only one planning step from the reward.

Except that the example described above suggests otherwise. There was an additional planning step, occurring on step 6, where the planning classifier "#, *j* : 1" posted its message. The extra step serves no useful purpose. Furthermore, if this extra step occurs consistently (which it would, during the test period), the planning classifier that posted the previous message (#, *b* : 0) should have a prediction of 710 rather than 1000. The reactive classifier that preceded #, *b* : 0 would have a prediction of 504.1 rather than 710, etc. According to the performance log, the system executed an average of 1.20 planning steps per problem; if the extra planning step is always executed, this number would have to be higher.

The classifier parameters for the HXCS that achieved optimal performance were saved to a file, and were later reloaded in order to create the example. Floating point numbers are notoriously fickle. The predictions for the best reactive and planning candidates in step 6 are both reported as 1000.0. If these values were misleading, such that the best reactive classifier's prediction was actually larger than the best planning

classifier's prediction by an infinitesimal amount, the redundant planning step would be avoided. When these values were saved to a text file, this miniscule difference would be lost due to rounding; after reloading these classifier parameters, the system would behave differently. An informal experiment proved that this can indeed happen.

Obviously, it is undesirable for a learning algorithm to be dependent on such fragile serendipity. The planning classifiers matching positions *e* and *j* should predict a payoff of 1000, because once one of them posts its message, there is no reason to post any subsequent messages before the food is obtained; the message will still be on the list when the goal is reached, so the classifiers are updated with the full reward. But if the animat is passing *through e* or *j*, no message should be posted, since internal messages are useless after the animat has passed one of the aliasing positions (unless it backtracks). This kind of behaviour is unlikely to occur using the current implementation (it occurs only when the classifier predictions are infinitesimally different in just the right way); if planning classifiers for *e* and *j* have predictions of 1000, it is likely that they will post their messages (which will have the maximum possible system prediction) even when they serve no useful purpose.

One solution is to bias action selection (Booker, 1990) so as to discourage useless planning steps. During the ascent phase, when the planning and reactive levels are competing for the next action, the planning level could be prevented from winning if the maximum system prediction is not greater than that of the current message. The assumption is that if the proposed message leads to the same payoff as the current one, there is no reason to post it. The results for HXCS with the bias are shown in Figure 13.

The bias is not enabled until the onset of the test period; if enabled from the beginning, it could cripple exploration.

Performance is abysmal. By the end of the test period, the average number of reactive steps is 357.75! Ironically, the number of planning steps, which the bias was supposed to decrease, does not change noticeably. By examining an execution trace, and by inspecting the classifiers, a partial interpretation of the disaster is possible. In each experiment, one of the two planning classifiers that matches *d* or *f* and a start position bit of zero (i.e., the animat did not begin the episode at *d* or *f*) has a prediction of 1000. For an optimal policy, the planning classifiers matching *d* or *f* should *never* post their message unless the animat begins at one of these cells (i.e., the start position bit is 1). Because a planning classifier for one of the aliasing positions has the maximum possible prediction, its message will always be posted. The internal message is used to disambiguate an aliasing state, but because the *same* message is always posted upon reaching such a state, no disambiguation is possible; the same motor action will be chosen regardless of whether the animat is at *d* or *f*, until the choice is altered by a series of reinforcements. In other words, HXCS with the bias behaves similarly to XCS, which is not exactly a sign of progress.

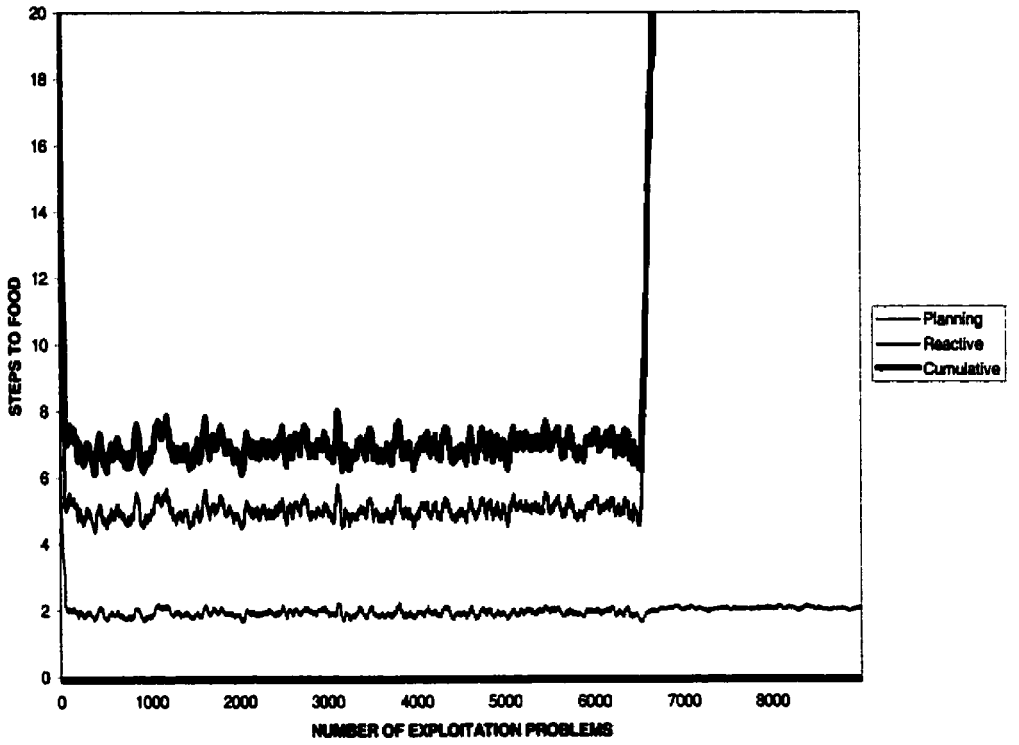

Figure 13: HXCS with start position bit and bias in Woods101. The bias is enabled at the onset of the test period (after 6500 exploitation problems).

Perhaps it is too extreme to always require a message's system prediction to be larger than the system prediction of the current message. By adding a little randomness, a planning classifier for *d* and *f* could be prevented from obtaining a stranglehold on the system's behaviour. The results in Figure 14 are for a bias that is applied 80% of the time. As before, the bias is disabled during the learning period. The softening of the bias does indeed prevent the terrible performance of the previous experiment, but performance is worse than for HXCS without any bias. By the end of the test period, the number of planning steps is 2.02 and the number of reactive steps is 3.65. Results for a bias that is applied 90% of the time (not shown) were slightly worse.

So, while a bias is good for enforcing the desired behaviour on a system that has already learned the correct classifier parameters, it appears that it tends to interfere with the learning of these parameters.

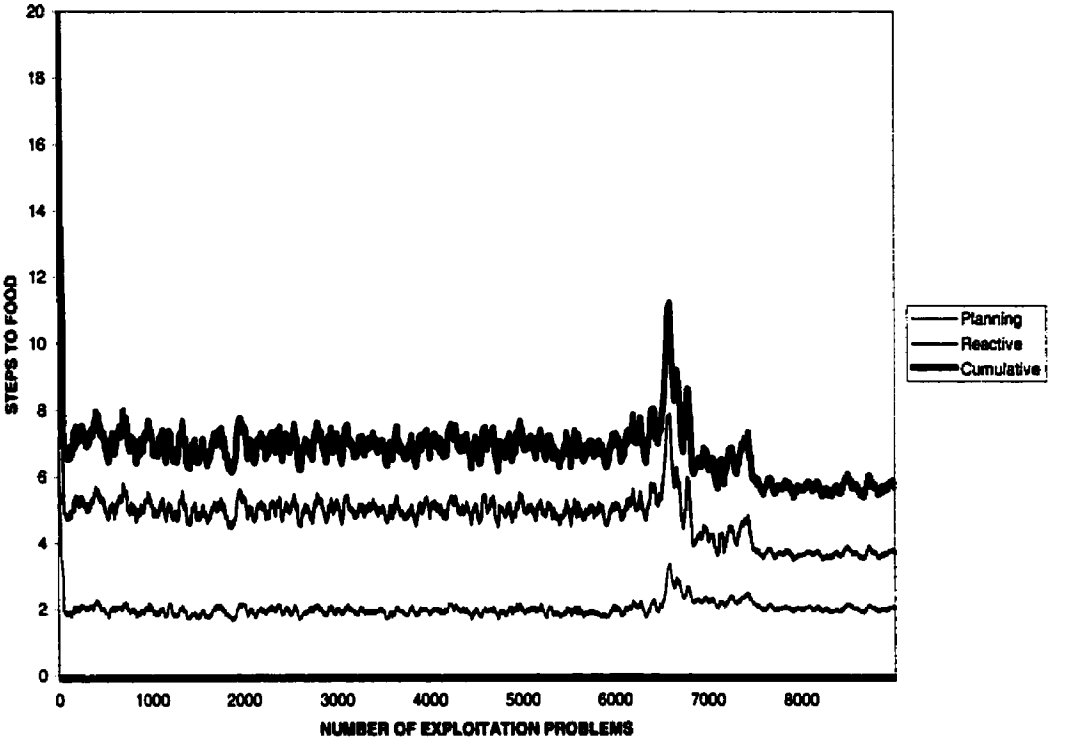

Figure 14: HXCS with start position bit and bias that is applied 80% of the time. The bias is enabled at the onset of the test period, after 6500 exploitation problems.

9. Discussion

Interpretation of Results

HXCS was able to achieve optimal performance in Woods101 with the addition of the start position bit. Optimality was attained, however, in only a fraction of experiments. Even worse, optimality depended on infinitesimal differences between ostensibly identical prediction values.

For illustration, consider an extremely simple woods consisting of two empty cells (labelled *a* and *b*) and a food cell, F:

a	*b*	F

Suppose the HXCS for this environment has two planning classifiers, P_a, matching *a*, and P_b, matching *b*. Similarly, it has two reactive classifiers, R_a and R_b, each advocating a move to the right. The discount factor is 0.71, and the reward for obtaining food is 1000. If the animat starts at *a*, P_a will post its message. No other planning steps (message postings) should occur before the animat reaches F, since the internal messages do not aid performance (this includes P_a's message, but the algorithm requires that every problem begin with a planning step). Therefore, under an optimal policy, P_a's message will still be active when the reward is received for reaching the food, and P_a's prediction will equal 1000, the undiscounted reward. The same is true of position *b* and P_b. R_b's prediction will also be 1000, since its action leads directly to the food. R_a is reinforced by R_b, so its prediction equals R_b's discounted prediction, $1000 * 0.71 = 710$.

But the large system prediction for P_b's message means that P_b has a high probability of posting its message once the animat reaches *b*, even if it started at *a*, resulting in a useless planning step. Because of the extra step, P_a's prediction will reflect the *discounted* reward (P_b's prediction multiplied by the discount factor) and become less than 1000 (the accuracy of its prediction could also suffer, since P_b may win the competition and post its message only part of the time).

The useless planning step is consistently avoided, however, if P_b's prediction actually converges to a value marginally less than R_b's prediction. When the animat moves from *a* to *b*, the phase becomes "ascent" resulting in a competition between the reactive and planning levels for the next decision. If R_b's prediction is greater than P_b's prediction, R_b's action will always be executed in preference to the posting of P_b's message (assuming action selection is deterministic). Thus, there is no redundant planning step.

Future Work

The experiments reported for HXCS are extremely limited in scope. The classifier populations are pre-initialized with an appropriate set of rules, and never change (excepting the parameters of individual rules). The ability of HXCS to discover good classifiers needs to be tested, by initializing the system with an empty or random set of rules and enabling the genetic algorithm.

To date, HXCS has been tested in only one environment. Woods101 tests HXCS's ability to use internal messages to learn an optimal policy for a non-Markov environment. One benefit of hierarchical reinforcement is the shortening of rule chains,

so that classifier parameters may converge to their true values more quickly; the rule chains in Woods101 are too short for this advantage to be properly evaluated. Future experiments could use woods where the average distance to the goal is much greater.

Also, it is intended that HXCS evolve hierarchically organized behavioural modules that explicitly convert a goal into tasks and sub-tasks. This kind of structure both facilitates analysis of what a system has learned, and provides the system with reusable building blocks that improve learning. To assess HXCS's ability to learn behavioural modules, it is best to use a problem with an obviously hierarchical solution. For example, the navigational task of reaching a landmark can be decomposed into sub-tasks involving landmarks encountered en route (as per experiments with the MonaLysa architecture). The learning of grammatical syntax, where behavioural modules correspond to non-terminals, is another possibility.

HXCS has been tested with one planning level, but it may theoretically accommodate an arbitrary number of levels. How practical is it to co-evolve multiple planning levels plus the reactive level? Also, how does one know in advance how many levels are needed for a given problem? One possibility is to start with one (or zero) planning levels and add more as needed, incrementally extending the hierarchy's depth. This approach depends on a triggering function that specifies when a new level is required. For example, the function might check whether the system error is above a threshold for an extended period.

Also, what shaping policy is best for training the various levels? Two policies were tested with ALECSYS: (1) a *holistic* policy, where all modules were trained

concurrently, and (2) a *modular* policy, where modules were trained one at a time and then frozen, moving bottom-up. The same policies could be applied to HXCS's levels.

Experimental results suggested that HXCS needs some kind of bias to eliminate redundant planning steps and reliably achieve optimal performance. Unfortunately, one failed attempt to devise such a function has shown that a bias may interfere with exploration and actually decrease performance. More study of potential biases is required.

Lanzi's experiments with XCSM showed that performance was improved when the number of register bits was greater than the minimum required to disambiguate the aliasing states. The same could be true of HXCS and the length of its internal messages. Redundant message bits could provide an alternative to the start position bit extension. If the messages posted when the animat started at an aliasing position were different from those posted when it passed through such a position, accurate reactive classifiers matching the latter could be distinguished from inaccurate classifiers matching the former.

Obstacles to hierarchical learning

Lanzi's study of XCSM, another variation of XCS with internal state, yielded some interesting observations applicable to HXCS. In XCSM, each classifier has, in additional to the conventional external condition and external action, an internal condition matching the memory register and an external action that modifies this register. Although XCSM was able to learn an optimal solution in Woods101, performance deteriorated in more difficult woods. It was discovered that XCSM's policy for exploring

actions was responsible for the problem. During exploration, both the internal and external actions were chosen at random. Internal actions were explored at the same rate as external actions, making it difficult for XCSM to co-evolve a "language" (of register settings) and its "interpretation." Put another way, because the register bits do not have a stable "meaning," classifiers receive variable payoff within the same context, making them inaccurate (Lanzi and Wilson, 2000). According to hierarchy theory, the upper levels of a hierarchy should be more stable than the lower levels (Ahl and Allen, 1996). The solution was to modify action selection during exploration; the internal action is chosen first, deterministically, followed by the external action, chosen randomly as before. Internal actions were therefore never explored by the reinforcement component, only the genetic algorithm. The internal language being explored relatively slowly, it was now easier to learn a useful interpretation. This modified version of XCSM was called XCSMH (Lanzi, 1998; Lanzi and Wilson, 2000).

HXCS also uses a language (of internal messages posted by the planning level) that is interpreted (by the reactive level). If planning and reactive niches are explored at the same rate, classifiers will tend to be inaccurate. A reactive classifier treats a message the same way regardless of which planning classifiers posted it; to the reactive classifier, the context is the same, yet the payoffs will often be different. Unfortunately, the solution developed for XCSM is not directly applicable to HXCS. In XCSM, the reactive classifiers are extended with internal parts; in HXCS, the reactive and planning levels are distinct sets of classifiers. If planning messages are always explored deterministically, many planning classifiers will never be evaluated. One solution is to select messages

deterministically most of the time, but inject enough randomness to ensure that all planning classifiers gain experience.

In the original implementation of XCS, actions are selected completely at random during exploration; this policy is called "pure explore" (Wilson, 1995). Lanzi used a modified strategy: actions are selected at random with probability p_s; the rest of the time, actions are selected deterministically, as is done during exploitation. Pure explore is equivalent to $p_s = 1.0$. Exploration at the planning level of HXCS could be slowed by setting p_s at the planning level (p_s^{planning}) to a value significantly smaller than that of p_s at the reactive level (p_s^{reactive}). In fact, such experiments have already been run (e.g., $p_s^{\text{planning}} = 0.3$, $p_s^{\text{reactive}} = 1.0$, and $p_s^{\text{planning}} = 0.1$, $p_s^{\text{reactive}} = 0.3$), but there was no evidence of improved performance.

This lack of improvement may be partly due to the two-way coupling between the planning and reactive levels. The reactive level depends on accurate planning classifiers, but the reverse is also true. The more frequently actions at the reactive level are chosen randomly, the longer the animat's average path will be. During such undirected meandering, there is an increased likelihood of additional planning steps. Because the number of planning steps that follow the posting of a message is variable, the planning classifiers will be inaccurate.

The close coupling between the planning and reactive levels makes it difficult to learn stable hierarchies. Randomness and inaccuracy at the planning level affect the reactive level (because reactive classifiers interpret internal messages, and sometimes receive payoff directly from the planning level), and the same is true of the reactive

level's influence on the planning level (because the number of planning steps is unpredictable).

Conclusion

Hierarchical system have the potential to scale much better than non-hierarchical systems. A hierarchical CS can break complex tasks into more manageable subtasks, and may use either internal messages or action sequence "chunks" to achieve optimal performance in non-Markov environments. HXCS does not require any special domain knowledge, hierarchical relationships are learned rather than pre-designed, and it is infinitely extensible. In practice, however, it has only been tested in a simple, non-Markov environment. It is capable of learning an optimal policy in Woods101, but does not do so consistently, partly due to the tight coupling between levels. Behavioural biases and new exploration strategies may allow this limitation to be overcome.

References

Ahl, V., and Allen, T. F. H., Hierarchy Theory: A Vision, Vocabulary, and Epistemology, Columbia University Press, New York, 1996, 206 pp.

Booker, L. B., "Classifier Systems that Learn Internal World Models," *Machine Learning*, 3, pp. 161-192, 1988.

Booker, L. B., "Instinct as an Inductive Bias for Learning Behavioral Sequences," *From Animals to Animats 1. Proceedings of the First International Conference on Simulation of Adaptive Behavior (SAB90)*, Meyer and Wilson, Eds: The MIT Press/Bradford Books, pp. 230-237, 1990.

Booker, L. B., Intelligent behavior as an adaptation to the task environment, Ph.D. Dissertation (Computer and Communication Sciences), The University of Michigan, 1982, 236 pp.

Booker, L. B., Riolo, R. L., and Holland, J. H., "Learning and Representation in Classifier Systems," *Artificial Intelligence and Neural Networks*, Honavar and Uhr, Eds: Academic Press, pp. 581-613, 1994.

Butz, M. V. and Wilson, S. W., An Algorithmic Description of XCS, IlliGAL Report No. 2000017, 17 pp., 2000.

Cliff, D. and Ross, S., Adding Temporary Memory to ZCS, Technical Report CSRP347, School of Cognitive and Computing Sciences, University of Sussex, 1995.

Donnart, J. Y., and J. A. Meyer, "A Hierarchical Classifier System Implementing a Motivationally Autonomous Animat," *From Animals to Animats 3: Proceedings of the 3rd Int. Conf. on Simulation of Adaptive Behaviour*, Cliff et al., Eds: The MIT Press/Braford Books, pp. 144-153, 1994.

Donnart, J. Y., and J. A. Meyer, "Learning Reactive and Planning Rules in a Motivationally Autonomous Animat," *IEEE Transactions on Systems, Man and Cybernetics - Part B: Cybernetics*, 26(3), pp. 381-395, 1996.

Dorigo, M. and Bersini, H., "A Comparison of Q-Learning and Classifier Systems," *From Animals to Animats 3: Proceedings of the 3rd Int. Conf. on Simulation of Adaptive Behaviour*, Cliff et al., Eds: The MIT Press/Braford Books, pp. 248-255, 1994.

Dorigo, M., "Alecysys and the Autonomous: Learning to Control a Real Robot by Distributed Classifier Systems," *Machine Learning*, 19, pp. 209-240, 1995.

Goldberg, D. E., Genetic Algorithms in Search, Optimization & Machine Learning, Addison-Wesley, 1989, 241 pp.

Grefenstette, J. J., "Credit Assignment in Rule Discovery Systems Based on Genetic Algorithms," *Machine Learning*, 3, pp. 225-245, 1988.

Kovacs, T., "XCS Classifier System Reliably Evolves Accurate, Complete, and Minimal Representations for Boolean Functions", *Soft Computing in Engineering Design and Manufacturing* (WSC2), Row, Chaudhry, and Pant, Eds: Springer-Verlag London, 1996.

Lanzi, P. L. and Wilson, S. W., "Toward optimal classifier system performance in non-Markov environments," *Evolutionary Computation*, 8(4), pp. 393-418, 2000.

Lanzi, P. L., "A Study of the Generalization Capabilities of XCS," *Proceedings of the 7th International Conference on Genetic Algorithms (ICGA97)*, Bäck, Ed: Morgan Kaufmann, pp. 418-425.

Lanzi, P. L., "An analysis of the memory mechanism of XCSM," *Genetic Programming 1998: Proceedings of the Third Annual Conference*, Koza et al., Eds: Morgan Kaufmann, pp. 643-651, 1998.

Lanzi, P. L., "Adding Memory to XCS," *Proceedings of the IEEE Conference on Evolutionary Computation (ICEC98)*, IEEE Press, 1998.

Martin, R. C., Designing Object-Oriented C++ Applications Using the Booch Method, Prentice-Hall, Inc., 1995, 528 pp.

Riolo, R. L., "Bucket Brigade Performance: I. Long Sequences of Classifiers," *Proceedings of the 2nd Int. Conf. on Genetic Algorithms*, Grefenstette, Ed: Lawrence Erlbaum Associates, pp. 184-195, 1987.

Riolo, R. L., "Lookahead Planning and Latent Learning in a Classifier System," *From Animals to Animats 1. Proceedings of the First International Conference on Simulation of Adaptive Behavior (SAB90)*, Meyer and Wilson, Eds: The MIT Press/Bradford Books, pp. 316-326, 1990.

Shu, L., and J. Schaeffer, "HCS: Adding Hierarchies to Classifier Systems," *Proceedings of the 4th Int. Conf. on Genetic Algorithms*, Belew and Booker, Eds: Morgan Kaufmann, pp. 339-345, 1991.

Stolzmann, W., Learning Classifier Systems using the Cognitive Mechanism of Anticipatory Behavioral Control - Detailed Version, Report No. 4, Osnabrück: FMD, AG Mathesis, 19 pp., 1996.

Watkins, C., and P. Dayan, "Technical Note: Q-learning," *Machine Learning*, 8, pp. 279-292, 1992.

Weiss, G., "Hierarchical Chunking in Classifier Systems," *Proceedings of the Twelfth National Conference on Artificial Intelligence*, AAAI Press/The MIT Press, Vol. 2, 1994, 1335-1340.

Wilson, S. W., State of XCS Classifier System Research, Technical Report No. 99.1.1, Prediction Dynamics, 1999, 21 pp.

Wilson, S. W., "Explore/exploit strategies in autonomy," *From animals to animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, Maes, Mataric, Pollack, Meyer, and Wilson, Eds.: The MIT Press/Bradford Books, pp. 325-332, 1996.

Wilson, S. W., "Classifier Fitness Based on Accuracy," *Evolutionary Computation*, 3(2), pp. 149-175, 1995.

Wilson, S. W., "Generalization in XCS," unpublished contribution to the ICML '96 Workshop on Evolutionary Computing and Machine Learning., 1996.

Wilson, S. W., "Hierarchical Credit Allocation in a Classifier System," *Genetic algorithms and simulated annealing*, Davies, Ed.: Pitman, pp. 104-115, 1987.

Wilson, S. W., "ZCS: A Zeroth Order Classifier System," *Evolutionary Computation*, 2(1), pp. 1-18, 1994.

Wilson, S. W., and D. E. Goldberg, "A critical review of classifier systems," *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, Davis, Ed.: Morgan Kaufmann, pp. 244-255, 1989.