This is going to be a tutorial on the XCS classifier system. I will try to explain it so that everyone gets a good idea how it works. I'll spend most of the time on this. I will assume no prior knowledge of classifier systems or XCS. Toward the end I will indicate some important directions of current research.

The talk will not be about classifier systems in general. I remain inspired by Holland's basic framework. But, as some here know, I believe certain key aspects must be changed in order for the framework to be successful. Some of the changes are embodied in XCS, which appears to work much better than previous classifier systems. I think people at a tutorial would like to learn about things that work better, so I will focus on XCS. ♣

However, it is important to know about the traditional system. I would recommend Holland's chapter in the 1986 Machine Learning book, as well as the chapter in Dave Goldberg's book. I would also recommend papers by Lashon Booker, John Holmes, Rick Riolo, Rob Smith, and Tom Westerdale among others. Plus my own papers prior to 1995. At the end, if people are interested, I will discuss the relationship between XCS and earlier classifier system work. Tim Kovacs has investigated this and has some important insights. [BREAK]

I will do this talk in a sort of question-answer format, posing and then answering what would seem to be the next major question. But, if you have another question as I go along, and it is urgent enough, please interrupt and ask it. I will also deliberately pause for questions. ♣

Okay, What is XCS? First of all, it is a learning machine—a learning program within a computer. Here, learning means that behavior improves with time, through interaction with the environment.

Many learning programs start with *a priori* information about the problems that will be faced. Such information is often called "domain knowledge". If the program still goes on to improve its behavior, this is not exactly cheating. But there can be confusion between what is given in advance and what is actually learned.

To understand learning per se, and eventually to create really powerful learning machines, I have felt it important to start with *minimum* a priori information, so that as much of the machine's knowledge as possible results from adaptation to the environment. Then whatever we *do* have to put in, in order to get to higher levels, we will know is probably essential and cannot be left out.

Another aspect of XCS is that its learning is "on-line". This means that it has to learn as it goes along. It does not have the luxury of collecting a lot of experience in some sort of temporary storage and then processing it at leisure for the implications. Instead, XCS must extract the implication of each experience as it occurs, because the raw data cannot be saved.

This criterion is at odds with the way humans sometimes learn, since we sometimes sit down and study large amounts of collected data free of pressure for immediate performance. However, it seems likely that the simpler the animal, the less this is possible, and that they learn primarily "on-line". Since simple animals can still be amazingly competent, it seems essential to understand how *our machines* can also learn on-line.

A third aspect of XCS is that as it learns it attempts to capture regularities of the environment. This means to detect and lump together situations that call for the same behavior—even though the situations may appear different. A machine with even a small number of sensors will encounter an enormous number of sensory states in any reasonably complicated environment.

A sensory state means a particular vector of values on the sensors. In order to avoid an explosive demand on memory, the learning machine must be able to *group* states having the *same* implication for its behavior. This is what I call the problem of generalization. XCS is able to generalize quite powerfully, if the environment allows it. ♣

What does XCS learn? Always, it learns to get reinforcements. More precisely, it learns to act so as to maximize a summation of current and future reinforcements. As the diagram indicates, XCS receives inputs—sensory state vectors—from the environment and emits actions that affect the environment and may result in payoffs.

The payoff is always a scalar. A payoff associated with need satisfaction could be represented by a high positive number. A payoff associated with pain could be represented by a large negative number, etc.

This is the framework of *reinforcement learning*, which seems to me to be the appropriate framework for developing learning machines that can function autonomously. Often, we don't know what a machine should do in order to achieve a goal that we set for it. We do not "see" the environment the way its sensors do, and we cannot predict how its effectors will affect the environment.

But we do often know the end results we want and can attach reinforcement values to them. We can say, "I want the machine to find as much *dust* as possible, so I will give it a small payoff every time it finds some", etc. This is usually much easier than telling it—as a teacher might—just what to do to find dust. ♣

All right, what are XCS's inputs and outputs? Now we are going to simplify a lot. Most of the work so far with XCS has been done with binary input vectors and discrete actions. Thus what I called the sensory input vector is a string of bits. To make it a little more organic, you can think of each bit as the result of thresholding the continuous valued output of some sensor. Despite the resulting loss of information, binary sensor values are adequate in many behavioral situations.

There is nothing in principle that prevents XCS from using continuous sensor values directly, i.e., using real input vectors. I will describe an approach to this later on.

Outputs are also discrete, though not necessarily two-valued. In some problems, XCS will learn to give a yes-no decision, and the outputs will therefore be 1 and 0. In other problems, the outputs will be actions in a physical (or simulated physical) environment, such as move a predetermined distance forward, or turn left through a predetermined angle.

For discrete actions to be effective, their size should be large enough so that strings of such actions actually accomplish something in a useful time, and small enough so that fine maneuvers can be carried out if necessary. A better solution is for actions to be continuous, such as "head 34 degrees left until you reach the door", but this is in the future for XCS. ♣

Now I'm going to start talking about what's inside of XCS. First, I will describe the system *after* it has learned something, and show how it arrives at a decision, given some input.

XCS's knowledge is contained in a *set* of condition-action rules called *classifiers*. Each classifier consists of a condition part, an action part, and a prediction part. The classifier "says": if my condition matches the sensory input and my action is taken by the system, I predict that the payoff will be as follows.

The example classifier says: if the first two bits of the input are 0 and 1 and the fourth bit is also 1, I predict a payoff of 943.2 if the system takes action 1. (The # symbol means I don't care what the value of this component is.) This rule matches a total of eight possible input vectors, so it is making a statement that applies to all eight of those inputs. It is expressing a *generalization* in the sense discussed at the beginning. It is saying that the environment seems to have a regularity such that if three of the input bits are set as just described and the action is action 1, the payoff will be 943.2 regardless of the settings of the other three bits.

Whether this classifier is *correct* is another matter. We can certainly doubt that the payoff will be exactly 943.2 for all eight matching input vectors. In fact, 943.2 may be an average over a widely different set of actual payoffs for those eight cases. Thus, while the classifier asserts a generalization, the generalization's *accuracy* may be high or it may be low. So the classifier may be very useful to the system as a predictor, or it may be of little use.

There are many classifiers within the system at any one time, perhaps several hundred in the types of problems I have studied. If you examine one of the classifiers, you will find that it makes a payoff prediction, as above, with respect to some *subset* of the input space in combination with one of the possible actions. After XCS has learned for a while, it will contain classifiers that cover all parts of the input and action space that it has experienced, plus often much more that it has not experienced.

There is an important difference between XCS and learning systems based on artificial neural networks. In XCS, the knowledge about subsets of the problem space is encapsulated in individual classifiers. That is, given an input, information about payoff for that input (in combination with a particular action) will be contained in just a few individual classifiers, maybe just one. Conversely, given a classifier, it makes an assertion with respect to a definite subspace of the input, and says nothing about other parts of the input space.

In contrast, the nature of network-based systems is that payoff information, for any input, is distributed over the whole network, and is in general extracted from the network by adding up contributions from all parts of it. This is the nature of the connectionist or "parallel distributed processing" (PDP) approach associated with workers like Rumelhart and others.

While it is formally possible to relate XCS's rule-based approach to the PDP approach, they seem distinct enough to merit pursuing quite separately on their own terms, at least until the potentials of both are more fully understood. XCS is one of very few rule-based approaches that are truly adaptive. I will say more about the differences between XCS and PDP later on. ♣

Now I will begin to explain how XCS works by restricting attention to the so-called performance cycle, i.e., what happens when XCS simply makes a decision in the presence of an input. Learning steps will be left out for the moment.

Also for simplicity, we will assume that XCS is in a "one-step" problem environment. In a one-step problem, a single input is put to the system, it chooses an action, and a reward is returned. Then another one-step problem occurs, with no connection to the first. This allows us, for the moment, to avoid the complication of a sequential problem in which many steps may occur before there is a reward.

Here is what you need of XCS for the performance cycle. Assume that a population of classifiers [P] is already present in XCS (I'll get to its origin shortly).

In the diagram, XCS receives an input 0011. The input is compared with the conditions of all the classifiers in the system's current population [P]. Classifiers that match are placed in the match set [M]. The other classifiers play no further role in this problem. The contents of the match set embody the totality of XCS's current knowledge about what to do with this input. Formation of the match set is a sort of recognition step. The classifiers in [M] can be said to recognize this input.

Notice that of the four classifiers in [M], two have action 01 and two have action 11. Consider the two with action 01. Their predictions are quite different: 43 and 27. Which prediction should XCS use for action 01? Perhaps it should combine them somehow?

We need a notion of the reliability of a classifier's prediction. If we had that, we could choose the more reliable prediction, or we could blend the predictions in accordance with the reliabilities. In fact, XCS blends them. Notice that there are two parameters associated with each classifier besides its prediction: ϵ and F. ϵ is an estimate of the *error* in the prediction, and F, *fitness*, is an inverse function of ϵ.

I will describe the calculation of ϵ and F shortly. For now, just notice that F is large when ϵ is small. XCS uses F as the measure of classifier reliability, so that reliability in effect goes up as error goes down. Or up as the classifier's accuracy goes up.

The net prediction for action 01 is simply calculated by taking a weighted average of the two individual predictions, where the weights are the respective values of F. I don't have an equation for that, but you know this is just the F's times the p's, divided by the sum of the F's. The result is placed in action 01's position in the Prediction Array. It is what I call the *system prediction* for that action.

The system prediction is a quantity distinct from the prediction of any individual classifier. Notice that here the system prediction, 42.5, is very close to the prediction of the more accurate classifier, as it should be.

The system prediction for action 11 is similarly calculated. No system predictions for actions 00 and 10 are computed, since [M] contains no classifiers with those actions.

How should XCS now decide between actions 01 and 11? Well, you say, obviously it should choose action 01. Yes, it should, if its aim is to get the highest reward it can. Suppose it does do that. Then action 01 is sent to the environment—meaning the system tells its effectors to do the thing assigned to action 01. And the environment returns some reward value.

Finally, the two classifiers which advocated action 01 are placed in the *action set* [A]. So endeth the performance cycle. Let's now begin to ask about learning. ♣

We assumed that [P] was already full of classifiers. Let's still assume that, but inquire just how the classifiers acquire their predictions. Consider the action set [A] from the problem just discussed. Each of its classifiers made a prediction about what reward to expect, and now we have in hand an actual reward. Let's adjust the predictions accordingly.

The update expression says "replace the current p_j by p_j plus α times the difference between the current reward and p_j". (The left arrow means "replace by" or re-assign). The value of α is often about 0.2, so this step reduces the difference between p_j and R by 20%. If R is always the same and the update occurs infinitely many times, p_j will become equal to R. p_j will predict the reward exactly.

The interesting aspect of this update procedure, though, is that it achieves a "recency-weighted" estimate of R. It is a sort of exponential moving average of R, such that recent values of R have a greater weight. This is expressed in the equation shown. Recency weighting allows XCS to track an environment in which the reward values for given inputs are slowly changing. Faster tracking results from larger values of α. However, α should not be too large, or the noise-suppression advantages of averaging will be lost.

Okay, this is how predictions of classifiers in [A] are updated. But the classifiers in [A] were those which gave the highest system prediction. How do other classifiers in the match set get updated? Will they ever be in [A]? The answer is: they must sometimes be. I.e., XCS must sometimes choose apparently *sub*-optimal actions, in order to be sure it has sufficiently updated all classifiers. It must do that to be sure that the apparently optimal classifiers are in fact optimal!

This is an example of the famous—or infamous—*explore/exploit* dilemma. The system would like to choose the best action all the time in order to maximize its return. But it can't *determine* the best action without sampling other actions. So there is no way it can ever be *certain* that its return is maximal. There are many approaches to the explore/exploit dilemma, and none is perfect. For this talk, however, let's assume that—some fixed percentage of the time—the system chooses a *random* action from those in the prediction array. I will call this "exploration". The rest of the time it will pick the apparently best, highest predicting action. This will be called "exploitation". ♣

Okay, those are the predictions. Where do the classifiers themselves come from? I usually start with an *empty* population. So there is nothing to match the first input. To get started, and for any unmatched input afterwards, XCS creates a classifier by "covering". This occurs as shown. The created rule matches the input, has a random action, and is assigned a low initial prediction.

Notice that the new rule has a certain *number of #'s* in random positions. They give the rule an initial generality that will allow it to be tested in several distinct input situations.

Covering is only necessary initially and the number of classifiers so created is very small compared with the size of the input space. The vast majority of new rules are derived from existing rules. ♣

How are new rules derived? First we need to examine a classifier's other two principal parameters, the error and fitness. They are also updated whenever a classifier is in the action set. The error update is like that for prediction, except the quantity being averaged is not R, but the *absolute difference* between R and the current prediction p_j. This is a simple measure of the classifier's current error.

Now look at the equation for *accuracy*. It and the next one are very important in XCS. The classifier's accuracy, κ_j, is a negative power-function of its current error estimate. The power, n, is quite large, around 5. Accuracy is thus very steeply inverse to error. However, κ_j is not allowed to have an infinity. Any classifier with error less than or equal to ϵ_0 has a high but finite value for accuracy, as shown.

The next step is to compute κ_j', termed *relative* accuracy. It is just κ_j divided by the sum of the accuracies of all classifiers in the current action set. This is important, because what we really want to know is how the classifiers in [A] *compare* in terms of accuracy, and not their absolute accuracies per se.

Finally, the classifier's fitness F_j is computed by updating its current F_j using the value of κ_j'. Thus the fitness of a classifier is an estimate of its accuracy with respect to the accuracies of other classifiers in the action sets in which it occurs.

Now, let's make some *new* classifiers! With some probability—not always—we run a *genetic algorithm* in the action set. In XCS, the GA's population consists of just the classifiers in the current action set, not the population [P] as a whole. The steps are as shown. Two classifiers are selected with probability proportional to their fitnesses and copied. The copies will be the offspring.

Often, the offspring are crossed, for example as shown, where the vertical line is a randomly selected crossover point. You can see that the result of exchanging parts at the crossover point is the pair of classifiers on the right. As a last step, mutation occurs at individual positions with a low probability like 0.02. Then the resulting classifiers are inserted into the population.

Notice what is happening here. In the first place, the more accurate classifiers in [A] tend to reproduce. And, through crossover, their parts are often recombined. In this example, the results of crossing are a classifier that is more general than both parents, and a classifier that is more specific than both. This is not always the case, but the process tends on balance to *search* along the generality-specificity dimension, using pieces of existing higher accuracy classifiers.

A classifier that is more specific can never be less accurate, as a moment's reflection will show. Since the GA often produces a more specific offspring, it is clear that the population will tend, over time, toward having classifiers with greater accuracy, i.e., greater ability to predict the consequences of actions. ♣

Here is the previous overall diagram, adding the updating and GA components. The updates occur on every action set. The GA occurs less often, at a rate set to allow sufficient updating for the fitness values to be reasonably stable. ♣

Let's not forget the parents. What happens to them? They stay in [P], where in effect they enter into competition with their offspring. But this means that the population has enlarged by two. We do not want an indefinitely increasing population, so two classifiers must be deleted from [P].

There are a number of ways to do it, gracefully. Deletion in fact provides an opportunity to keep the system's resources balanced. Here, balance means that approximately the same number of classifiers are devoted to each action set "niche". This is achieved by letting the probability that classifier C_j will be deleted from [P] be proportional to the average size of the action sets in which it occurs.

All that is required is to add to the action set updates, an update of an estimate, kept by each classifier, of the number of classifiers in its action sets! The probability of deletion is made proportional to this estimate. Then classifiers in action sets larger than average will tend to be deleted more often, and the sizes will come down. Members of small action sets will be less likely to be deleted. As a result, action sets will tend to be about the same size. Methods for preferentially eliminating very low fitness classifiers can be added to this balancing based on action set size. The best deletion technique so far, in my opinion, is due to Tim Kovacs. [BREAK] ♣

Let's look at some results on a classical one-step problem, the Boolean multiplexer. I've used this problem a lot because it is difficult and non-linear, and because the multiplexers form a family of functions from which complexity estimates may be derived. I'll use the Boolean 6-multiplexer as the primary example.

Let's first define the function. The "6" means the input vector is six bits long. It goes into the function box and out comes an answer, 1 or 0. In the example shown, the correct answer is 0. We can get it two ways. You can get the right answer by thinking of the first two bits as an address into the remaining four bits. Thus the address bits, 10, address data bit 2 as shown by the arrow, and that is the answer. The other way is to process the input through the Boolean formula. For this input, none of the terms is true, so the result is 0.

The formula in bold says that there is a multiplexer function for integer values of k greater than 0. So $k = 2$ gives the 6-multiplexer. $k = 3$ gives the 11-multiplexer. $k = 4$ gives the 20-multiplexer, whose formula is shown. ♣

These are results for the 6-multiplexer. In this experiment, random inputs were presented. If XCS's decision was correct, the reward was 1000; if incorrect, 0. Learning problems alternated with test problems. In a learning problem, XCS functioned as described, but in the action selection step, it chose a random action. Thus every learning problem was done in exploration. Updates, GA, and everything else occurred as described. On test problems, XCS did the same, except it always chose the action—decision— with the maximum prediction.

The upper curve plots the fraction of correct test decisions, averaged over the preceding 50 test problems. It reaches 1.0 within 2,000 problems. The dashed curve shows the fall in the *system error*. This is the absolute difference between the reward and the system prediction for the action chosen on test, divided by 1000. I need to take a moment to explain the third curve.

I said that offspring classifiers are added to the population. Well, not exactly. Given a new offspring, the population is first searched to see if a classifier with the same condition and action is already present. If so, the existing classifier's *numerosity* parameter is incremented by one, and the new offspring is discarded. If not, the new offspring is added with its own numerosity set to 1.

As a result of this creation of so-called *macroclassifiers*, each member of the population is unique. Said another way, what would otherwise be n structurally identical classifiers are represented in the population by a single macroclassifier. Macroclassifiers make the system faster, plus they make it easier to "see the system's knowledge". But, where appropriate, all system *operations* take place as though the macroclassifier consisted of its constituent "micro"-classifiers; i.e., they take the numerosity into account.

The third curve shows the population size in these macroclassifiers. You can see that it initially rises rapidly from zero but then begins a gradual fall to 77 by 5,000 problems. What this indicates is that XCS is finding *general* classifiers to replace specific ones, so that the whole problem space can be handled by fewer classifiers. Let's look inside after 5,000 problems and see the classifiers that have actually been evolved. ♣

This is a listing of the population in descending order of numerosity. Notice first that the error estimates of all classifiers except the bottom two are zero. Thus accurate classifiers have been found. But note the first sixteen classifiers. Their address bits are specified, together with precisely the bit indexed by the address bits.

These classifiers are not only accurate, but are *maximally general*, in the sense that if you change any specified bit to #, the classifier will become highly inaccurate. Thus XCS has evolved classifiers that are both accurate *and* maximally general. The 16 classifiers correspond directly to the terms of the Boolean formula.

In fact, they constitute an *optimal cover* of the problem space. Tim Kovacs has studied the development of classifiers in XCS, and has made the hypothesis that XCS always drives toward an optimal cover.

Well, what about the other classifiers in the list? They are present because the system's search of the classifier space, of its model, continues on. *New* classifiers, not maximally general and sometimes inaccurate, are still present. However, note the developing abrupt fall in numerosity between numbers 15 and 16. Eventually it will be very sharp with even fewer classifiers beyond 15 and at lower numerosities and fitnesses (the fitness values in this listing are multiplied by 1000). These residual classifiers already have no effect on performance, as seen from the graph. ♣

Finally, it is fun to try to observe the creation, or at least the arrival, of one of these accurate maximally general classifiers. This shows some action sets for the particular input 101001 and action 0, when that input happened to arrive at problem numbers given on the left. On problem 247, the action set has three matching classifiers, including the completely general one, and all have huge errors. At problem 1135, all of these are gone, and the classifier we want has appeared, with zero error and a fitness already dominating the others.

At 1333, our favorite dominates a much smaller action set and its numerosity is growing. At 2410 it has just one companion, with fitness zero. And at 2725 it is joined by a couple of more-specific versions of itself. They are equally accurate but have much lower numerosities and fitnesses. Why is that?

We now should address the question of why XCS drives not only toward accurate classifiers, but ones that are also maximally general, as seen in the previous population listing. If fitness is based on accuracy, shouldn't XCS drive toward more specific classifiers, not more general ones? The answer is at the heart of XCS's ability to detect and represent the regularities in its environment. ♣

I've written the explanation out since it is so important. Let's go over it in conjunction with the two example classifiers shown. "Consider..."

The essence is that reproductive success depends not only on fitness, but on reproductive opportunity. A more general classifier will occur in more action sets, and so have more reproductive opportunities. By reproducing more, it will attain a greater numerosity. The greater numerosity will mean that more of the fitness update, which always sums to a constant, one, will be "steered" toward it and less toward its less general competitors. Gradually, if all are equally accurate, the more general classifier will drive the others out of the population. I.e., they will disappear.

The system will keep searching for yet more general versions of an accurate classifier until the point is reached where adding a # anywhere results in a loss of accuracy. Then the process will stop: any more general classifier will have little chance of survival.

This generalization mechanism is responsible for the gradual ascendancy of the 16 highest numerosity classifiers shown in the 6-multiplexer listing. XCS has in effect detected and represented the terms of the Boolean formula. For the multiplexer problem, these are the environmental regularities. [BREAK] ♣

Scale-up is an essential property of a learning system. As problems get larger, we want the system's memory or learning effort to grow much less rapidly than the size of the problem domain. In general, problem domains grow exponentially with the number of variables describing them. The worst case would be a system that must also grow exponentially. This would be a system that treated each input state individually, say using a gigantic table.

Intuitively, if the problem domain contains regularities, we would like the learning system to grow only as fast as the number of regularities. The multiplexer family of functions permits a test of XCS's capability in this regard. The three graphs show results for the 6-, 11-, and 20-multiplexers. Let us look at learning effort, as measured by the number of inputs required to reach 100% performance.

For the three tasks, the 100% point is reached at approximately 2,000, 10,000, and 50,000 problems, respectively. Thus each differs from the previous by a factor of five. Examination of the Boolean formulas shows that the number of terms doubles in going from one task to the next, i.e., a factor of two. At the same time, the input domain size goes from 2^6 to 2^{11} to 2^{20}, i.e., it grows exponentially. In fact, the *20-multiplexer* domain is so large that XCS has seen only about 5% of it by the time XCS reaches 100% performance.

My tentative conclusion from these relationships is that XCS's learning effort—in effect, its learning complexity—is much more closely tied to the number of *regularities or generalizations* in the input domain, than it is to the size of the domain itself. This is a very desirable property, if true. Several popular network-based or network-like approaches do not have the property. [BREAK] ♣

Let us go on to sequential or multi-step problems. They have the *new* complication that reward does not necessarily arrive on every step. Sometimes there is no reward on a step, so what should the system do, or learn?

Theoretical treatment of this issue is by now quite vast, and forms much of the subject called *reinforcement learning*. I will show one basic approach through a fairly simple example and some appeal to intuition.

Consider this portion of a grid-world. The system wants to be able to reach food, F, from any starting point, and it cannot pass through cells containing an O. One widely used reinforcement learning approach, called *Q-learning*, is to learn a *value function* of the states and actions. Then, in a state, the system chooses the action with the highest value.

How would this work out? Suppose we are in the state, or cell, just below the F. If we move North, we will get an external, a real, reward. So it makes sense to make the value function, say, 1, for that action in that state. What if we are one step away from that state, say under the O, and we move to the East? Well, we could then go North and get the reward, so maybe the East move should be valued the same, 1. That does not seem satisfactory, since the state under the O is two steps from the F. Let us instead value the East move at γ times the value of the best move in the state under the F, i.e., γ times 1, where γ is a constant somewhat less than 1, like 0.9.

Now let's go back to the state under the F, and consider a move to the West. How should it be valued? Using the rule just mentioned, we should value it at γ times the value of the best move from the resulting state, thus γ times γ, or γ^2. That's nice, because it reflects the fact that the minimal path if you start by moving West is three steps long. Continuing this way, we can fill in the action-values for every move in every state, and they will all reflect similarly the minimal distance to food.

Notice that we can fill in all the action-values based only on local updates. At any one time, we only need to remember the values of the current and succeeding states. By trying the moves and doing the updates, the action-values will gradually become reliable.

What has been *proved* for Q-learning is that if the environment is Markov and the updates are done sufficiently often, the action-value estimates will converge to values such that taking the action with the maximum value in every state will always result in the shortest path to the goal. If external reward occurs in *more* than one state, a similar, more general result holds stating that the above procedure will result in action-values such that following their maxima will result in an optimal flow of (discounted) future reward.

Now, to put this in XCS terms, the expression below shows the update procedure for updating predictions p_j in multi-step problems. The prediction is updated based on the maximum system prediction in the succeeding state plus any external reward, r_{imm}, in the current state. While this procedure is based on the Q-learning model, it is new because it is based on predictions of *rules*, which may be general in some degree, and not on predictions tied to individual *states*. Like all reinforcement learning procedures involving generalization, there are no proofs that the procedure results in an optimal policy. But empirically it works quite well. ♣

Here, quickly, is the full XCS diagram, with the multi-step parts added in. You see max, discount, summation, and time delay boxes required for the update expression on the last slide. For computational reasons, the update is actually done retrospectively, but the effect is identical to that expression. [BREAK] ♣

Now I will rather quickly go over some multi-step results. While XCS has by now been tried in quite a number of environments, the one here, called Woods2, is good to talk about because it has a surprising number of regularities that XCS captures in its generalizations.

The system, an animat represented by an asterisk, is placed randomly in an open cell of Woods2 and then, under control of XCS, moves until it bumps into food. There are two kinds of food, which look different to the animat, and there are two kinds of impenetrable rocks, which also look different. The actual coding of the sensory input vector is shown at the right. The sense vector is 24 bits long.

The left-hand graph shows performance, in average steps to food, versus the number of explore problems so far. An explore problem is a problem in which the animat starts at a random position, moves randomly, updating and doing the GA as it goes, all as previously described, and finally arrives at a food. *Performance* measures the number of steps to food on interleaved test problems, in which the animat always chooses the best move. You see that performance rather quickly comes down to the optimum. The three curves are for three different XCS regimes.

The graph on the right shows population size in terms of macroclassifiers. The message is that for the dashed regime, the number of classifiers condenses, via generalization, to a value less than 100. Since there are 560 distinct state-action pairs in Woods2, this indicates XCS's ability to detect and represent regularities in Woods2. ♣

In particular, this slide shows two of the generalizations found. Actually this data is from Woods1, which has just 2 bits per object instead of 3, but the results are similar. The first classifier matches in all positions marked "3". It says, in effect, I don't care about anything else, but if there is a blank cell to the West, then the action-value of moving North is 504. Since 504 equals 1000 times γ^2 in this case, the classifier in effect predicts a distance to food of three steps. XCS has discovered this truth about all states marked 3 and expressed it in a single classifier. Similarly, the other classifier expresses a regularity about all states with a non-blank object to the West. [BREAK]

I'd like to pause now and show a couple of videos. The first shows XCS after having learned Woods2. You will see the animat start in a random state, then proceed as in a test problem to food— nearly always by the shortest path. [SHOW VIDEO]

Now I'll show a brief video from another environment, Maze5. The video shows XCS learning from scratch. Again the animat will start at a random position. During a learning problem, its steps will be random with probability 0.5. It will choose the apparent best move the rest of the time. In between learning problems will be a test problem, in which all steps are exploit steps. The display is much faster in this video. At the beginning, the movements are very random looking. But that changes... [SHOW VIDEO]

Now I will describe a few current research directions. ♣

It is important to move from discrete inputs and actions, and also time—meaning finite "time-steps"—to continuous spaces. Inputs are of course often continuous, so that the sense vector becomes a real vector. The first expression suggests a way of doing classifier conditions for real vector inputs.

For each input component, there are two values, a "center", and a "spread". The center plus or minus the spread constitutes a sort of receptive field. For each component there is a receptive field and if the input components *all* fall within the corresponding receptive fields, the condition is satisfied, or matches. If there are n input components, the condition consists of $2n$ numbers.

The GA would act on these $2n$ numbers. Most often, the GA works with bitstrings. Real number strings are a somewhat different world. The reason is that the sampling implicitly performed by the GA is most effective on bitstrings, that is, on numbers expressed using the smallest possible practical base, 2. For real numbers, instead of converting everything to base 2, there has grown up a set of genetic operators including blending recombination and hill-climbing-like mutation. There is also a distinct evolutionary approach, originally from Germany, known as the *Evolutionsstrategie*.

For continuous *actions*, note that it is quite natural to express the classifier as shown in the second expression. It says, if the input is within these ranges, then if the action, a continuous quantity like a turn angle, is within this range, then the payoff prediction is p. This just reflects the fact that it is the space of inputs and actions, not just inputs, that we are concerned with. Of course, this classifier will have some finite error—the actual payoff received over its domain will vary—but the error may be acceptable.

With such classifiers, it may be possible to compute continuous actions by some form of combination of the classifiers that match. Combinations are analogously computed in the field of fuzzy logic, and in fact there are some fuzzy classifier systems in existence, though they do not define fitness as in XCS.

Finally, continuous time. Suppose you want to make mid-course corrections in your action very frequently, in order to achieve fine control. On a discrete basis, this could enormously expand the number of states, and it would take forever to learn the action-values. This problem is just beginning to be addressed in the reinforcement learning literature. I can't offer much at this point except the hint shown.

I'd like to go back now to the case of continuous, or *real* inputs. I'll show a few slides from some recent work I've done on that problem. [DISCUSS REAL SLIDES] ♣

Up to now I have talked about *Markov* environments. But *non*-Markov environments are by far the predominant kind. McCallum's Maze is a very simple example. The two states marked with * look identical to an animat that sees only one cell away. Yet the optimal actions in each are different. How can the system learn what to do?

In a Markov environment, the optimal action is always determinable from the current sensory input. No past information need be remembered. In a non-Markov environment, this is no longer the case. For the animat in those two positions, current input is insufficient to decide what to do. Markov environments are rare in the real world. Non-Markov environments are the rule. So it is vital to understand how to learn them.

One approach, the "history window", would concatenate sufficient previous inputs to the current input vector to make the environment, in effect, Markov. Classifier conditions would be extended accordingly. In McCallum's Maze, only the current and one previous input vector would be necessary. But other environments need larger history windows to resolve their so-called aliased states. In general, the space of input histories grows exponentially. Furthermore, the window information is only *needed* at aliased states. For all other states it's excess baggage.

In light of these problems, some systems, including neural networks, attempt to detect *correlations* between the current best action and past input events. The so-called "curse of dimensionality" is still not avoided, however.

The checked approach, "adaptive internal state", looks to me more promising than these, and is directly suitable for XCS. Observe that many decisions based on the past require only one or a few bits of information: Have I had lunch? If so, turn left toward class; if not, turn right toward the cafeteria. Such information could be stored in a short internal bit-register.

Consider extending the classifier format as shown at the bottom. The condition is extended to include an *internal condition* and the action is extended to include an *internal action*. The overall action performed by the system is then a combination of the internal action, in which the system writes into the register, and an external action, which is a physical action in the environment. Furthermore, to enter the match set, a classifier's condition must match both the environmental input and the contents of the register.

The hypothesis is that in non-Markov environments, this extension of XCS will evolve classifiers that write and read the internal register in such a way as to do the right thing in the aliased states!

Dave Cliff and Suzi Ross first tried the memory-register approach with the classifier system ZCS. They got good, though not optimal, performances in non-Markov environments such as McCallum's Maze. Recently, Pier Luca Lanzi extensively investigated using the register with XCS. I will discuss some slides from his work. [DISCUSS MEMORY SLIDES] ♣

Next, while we have seen that XCS is powerful at detecting and representing conjunctive generalizations—ANDs of variables— it is restricted with respect to other kinds of regularities, since it cannot straightforwardly express them. Consider the simple example shown here. It is a perfectly reasonable environmental regularity. Yet it would take a whole set of ordinary classifiers to express it. However, with a condition which said simply $(> x\ y)$—this is Lisp's way of saying true if x is greater than y—you could say it in one classifier.

Why not express classifier conditions as Lisp expressions (which are called *s-expressions*)? The whole energetic field called *genetic programming* has devoted itself to the evolution of such expressions. So the time is ripe to see if a classifier system can be built using s-classifiers! I actually thought of this five years ago, and hoped someone expert in genetic programming would try it.

Pier Luca Lanzi, who is expert in both XCS and genetic programming, took it up about a year ago. He's had fascinating results, and will give a talk on this topic in the conference on Saturday.

To be sure that you don't get the impression that XCS is only good in simulation, I should mention that I have done some preliminary work with XCS in both the Khepera robot simulator and the Khepera robot itself. This is really the subject of a full talk, so I will just show a video in which Khepera has learned to move around a simple environment without bumping into the walls. To teach this, reinforcements were only given if, when the robot was *not* close to a wall, it went straight ahead. In contrast, the behavior near a wall was not directly reinforced, but was learned as discussed earlier for sequential problems. [SHOW VIDEO] ♣

Finally, let me make this summary, which suggests some key differences between XCS and other reinforcement learning systems.

The big difference is that XCS is rule-based, not network-based. Under that heading it seems important that XCS's structure, the classifiers, is created as needed; this differentiates it from things like back-prop networks in which sufficient structure must be present in advance. In things like radial basis function and nearest neighbor approaches, structure can be created as needed, but that structure tends to be fixed and is not further adapted, as are XCS's classifiers.

Second, from comparisons—for instance on the multiplexer—the learning speed of XCS is at least as fast as for network approaches. I think this is because a classifier is already a non-linear structure, so that non-linear problems are more quickly adapted to.

Third, from the multiplexer results, it is likely that the learning complexity is significantly better than for networks. Many kinds of networks are known not to scale up well. They grow with the problem space, not the complexity of the problem function. The same is true of radial basis function approaches.

Fourth, classifiers have this neat ability to keep lots of statistics about themselves, such as their error, etc. It is very awkward to do this with networks—you end up needing a separate network for each type of statistic! I think that as we explore, we will find many more statistics that are useful in classifier systems.

Fifth, since classifiers are rules, the knowledge they embody is reasonably "transparent". In contrast to systems like networks where knowledge is distributed over the elements, knowledge in XCS is represented relatively clearly and compactly. This ability to "see the knowledge" will turn out very important as XCS is applied to data inference and other areas where understandability to human users is vital.

Finally, the fact that classifiers are rules, and can be manipulated like rules, may turn out very important when we want our systems to do things like reason. This is a bow to standard AI. But maybe adaptive classifiers are at just the right midpoint between symbolic and connectionist approaches to intelligence.