This will be an introduction to learning classifier systems. But I will focus mostly on XCS, which contains many of the ideas that are currently being pursued in the field. I will try to explain XCS so that everyone gets a good idea how it works. I will assume no prior knowledge of classifier systems or XCS. Toward the end I will indicate some important directions for future research.

Holland's basic classifier system framework certainly remains inspiring. But 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 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 first book. I would also recommend papers by Lashon Booker, Larry Bull, Stephanie Forrest, John Holmes, Tim Kovacs, Rick Riolo, and Rob Smith, 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 systems. [BREAK]

I will do this tutorial 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 behavior that improves with time, via 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 think it's 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 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 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 large 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 be the right 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 that 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? We are going to simplify a lot. Much of the work with classifier systems 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. Or vectors of integers. I will describe approaches to these 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, but small enough so that fine maneuvers can be carried out if necessary. A better solution would be actions that evolve to have the optimum value, such as "head 34 degrees left". A promising approach has recently been proposed, which I will describe later. ♣

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

XCS's knowledge is contained in a large *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 often 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 (PDP) or "neural network" approach originated by workers like Rumelhart and others.

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

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.

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 show 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 is called 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 that 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 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, 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? We 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, usually 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 $\kappa_j\prime$, 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 one classifier that is more general than both parents, and another 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.

Each classifier keeps an estimate of the number of classifiers in its action sets. The probability of deletion is made proportional to this estimate. Then classifiers in action sets that are 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. One can also factor in the age or *experience* of a classifier, so that inexperienced classifiers are not prematurely deleted. [BREAK] ♣

Let's look at some results on a classical one-step problem, the Boolean multiplexer. This problem is used 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. Problems as large as the 70-multiplexer, an enormous problem, have been solved. ♣

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 always chose the action—the decision—with the maximum prediction. Also, updates and the GA did *not* occur—that is, no learning occurred on test problems.

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 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 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, in the sense of being minimal in number while still covering every instance. A hypothesis has been made 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. These residual classifiers already have no effect on performance, since performance is 100% at this point.

I should note in passing that the classifiers' actual fitnesses were multiplied by 1000 for this list. The actual errors were divided by 1000. To me, these normalizations are helpful. I hope they aren't too confusing to you. ♣

Finally, it is fun to try to observe the creation, or at least the arrival, of one of these accurate maximally general classifiers. This slide 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. It's number 9.

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.e., to generalize. ♣

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 in XCS depends not only on fitness, but on reproductive opportunity. A more general classifier will occur in more action sets, and therefore 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. ♣

We can use these results to get a rough estimate of XCS's *learning complexity*. Each larger multiplexer was about five times harder than the previous one. When cases differ by a constant factor, a power function relationship is suggested. You can write D (for difficulty) equal to a constant c times g to some power p. If you take g as the number of maximal generalizations—equal to four times the number of terms in the formula—then choosing $p = 2.3$ and $c = 3.2$ gives a curve that fits the three multiplexer cases. Thus D is polynomial in g.

What is D with respect to l, the string length? Notice that l approaches 2^k for large k. At the same time, $g = 4 \cdot 2^k$, so that l is proportional to g. This means that D is also polynomial in l. The same polynomial relationship has recently been found to hold for the 37- and 70-multiplexers as well.

A tentative conclusion 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 learning techniques do not have the property. For instance, tile-coding, nearest neighbor, and standard neural networks. [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 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 applies to predictions made by *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 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 go quickly 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 the 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] ♣

So far I have described environments that produce binary inputs. What if the input variables are not binary? Suppose they are real or integer valued. It turns out XCS can be adapted rather easily to these cases. I will describe one scheme.

The classifier condition is changed from the 1,0,# notation to a concatenation of *interval predicates*, as shown. The interval for a given variable is denoted by an upper and a lower limit for that variable. A match occurs if every input component is between the corresponding limits. The classifier condition thus consists of $2n$ numbers, where n is the number of input variables.

Crossover may occur either between or within the predicates. Mutation changes an allele by a bounded random amount, as shown. Covering produces a classifier with a condition having interval predicates that match the components of the current input. ♣

This non-binary scheme is very suitable for *data mining* problems in which inputs are expressed as strings of integers or reals. Here you see the classic Wisconsin Breast Cancer dataset. It is available, along with many others, from UC Irvine. The instances look a little odd at first. But each consists of a number identifying that clinical case, followed by nine numbers which are attribute values between 1 and 10, and finally the case's outcome, 2 for benign and 4 for malignant. The meanings of the nine attributes are shown. The 699 total cases are divided into 458 benign and 241 malignant.

XCS, modified for integer inputs, is called XCSI. At the bottom of the slide you see results of a 10-fold cross-validation test after ten instances of XCSI learned from the dataset. 10-fold cross-validation is a standard method of evaluating the performance of a learning system on a given dataset. It attempts to estimate the performance of the system on further examples drawn from the same universe of examples.

Basically, you let XCSI learn from 90% of the dataset and test it on the remaining 10%. Then you have it learn again on another 90% and test it on the remaining 10%. You do this ten times and average the test results. As you can see, the test results were somewhat varied, with an average of about 95.5%. This and higher values produced after further work with XCSI are very competitive with the best results from other learning systems, such as decision trees or neural networks. It appears that XCSI-like systems are excellent tools for this kind of problem. ♣

XCSI not only shows high performance, but it readily allows human users to "see the knowledge" that it has learned and that it uses to make decisions. This is because the system consists of discrete ***classifiers*** and is not a network. ***Knowledge*** consists of generalizations that are as broad, while still being accurate, as the domain permits. In terms of XCSI, this means evolving the population until maximally general classifier have been found. The graph shows this process with the Wisconsin dataset.

Look carefully at the upper left-hand corner. You see the performance curve quickly reach 100%, so most of the graph is spent ***after*** performance is perfect. The rising curve is generality—the percentage of don't-cares in the population. The falling curve shows the population size. The size falls a lot after 100% performance is reached and is still falling at two million problems. Increasingly general classifiers are still being found. They are still accurate, of course, since performance remains at 100%.

If you examine the population at two million problems you find classifiers like the four shown at the bottom. It is easy to translate the actual classifier notation to English language statements–in fact, a program does it. These classifiers were drawn from the highest numerosity classifiers, which tend to be those with the best generalizations, as we saw earlier with the multiplexer. In fact, there is another program that culls the population and finds a very small subset that nonetheless covers all dataset instances. For the Wisconsin dataset, the subset is 23 classifiers, of which just 10 cover 90% of the instances. Thus XCSI is a data-miner offering both high performance *and* high visibility of knowledge. [BREAK] ♣

XCSI extends classifier syntax to include inputs with integer or real components. However, a *condition* is really a predicate—i.e. a truth function—and the syntax so far considered is just one subclass of possible predicates. For example, suppose, as seen in the slide, you needed a condition that was true for $x > y$. Then the so-called *conjunctive* syntax of standard classifiers would be awkward. You would need many classifiers, not just one, to capture this relation.

However, there is no reason you can't have classifiers with any required condition syntax! In fact, using conditions consisting of Lisp S-expressions of appropriate elementary functions, the system can evolve an almost unlimited variety of predicates. This means that essentially any generalization or regularity in the environment can be represented.

Initial experiments have borne out the power of S-expression conditions. As in genetic programming, however, there can be a "bloat" problem which reduces the transparency of evolved classifiers. More research is needed so that you can have both performance ***and*** transparency ***and*** completely general syntax. This line of research—toward classifier systems that can efficiently capture any required generalizations—is very important, in my opinion. [BREAK] ♣

Up to now, I've talked about so-called "Markov" environments. The way we use the term, an environment is Markov if knowing the current input is sufficient to allow a system to choose the optimal action. Standard condition-action classifiers in XCS basically assume that the environment is Markov.

Unfortunately, it turns out that most environments do not have the Markov property—they are ***non-Markov***. The best action to take at a given point may depend on both the current input and some number of prior inputs. Here is a simple example of a non-Markov environment, due to Andew McCallum. The arrows indicate two states that look identical to a system that can only see the adjacent cells. Yet the optimal actions are different. The two states are said to be "aliased"—another piece of terminology.

What additional information does the system need in order to take the right actions in these two states? Well, if it could see better—i.e. see a little deeper into the environment—it could tell the states apart and therefore take different actions appropriately. This would be a sensor-based solution. However, let us assume we can't change the system's hardware. What would we then do?

The options basically all use some form of temporary memory to disambiguate the two states. The history window option simply remembers some number of previous inputs. The classifier condition would be extended to, say, attempt to match both the current input and the prior one. This would be sufficient in this example to distinguish the two states. The initial non-Markov problem would in effect be converted to a Markov one, if you regard the present and prior inputs as forming the "current input".

You can see, however, that history window approaches are inefficient unless you know precisely how much window length is required. For if the window is too long, the system has to remember more than necessary, and the number of required classifiers grows rapidly. On the other hand, if the window length is too small, the problem will remain unsolved, though performance may improve somewhat.

Another approach—actually the one used by McCallum—is to remember all past states and statistically look for correlations between them and the right move. That is, try to identify events in the past that can tell you what move to make now. This method can be more efficient than the history window, but it can still require an explosive amount of temporary memory on which to base sufficient statistics.

Finally, the concept that has been tried with XCS is—appropriately—more Darwinian. It is to *evolve* internal symbols, or signals, that the system can use to tell the aliased states apart. This amounts to the creation of *adaptive internal state*. It is a simplified version of Holland's original message list concept. ♣

Consider a classifier whose condition concatenates an environmental condition, as usual, with an internal condition. And let the action be a combination of an external action and an internal action. Also, let the system have an *internal register* R. The internal action modifies R. For the classifier to match, its environmental condition must match the input, and its internal condition must match R. The system's *internal state* consists of the current contents of R.

Now, consider a non-Markov problem like McCallum's maze. One can imagine, at least, that R might get set to, say, 0 when the system enters the left-hand aliased state. And, conveniently, that R would get set to 1 when the system enters the right-hand state. If this happened, a classifier matching the input, looking for 0 in R, and advocating the action, "move south-east", would reliably receive a high payoff. Similarly, a different classifier that matched the input, looked for a 1 in R, and advocated moving south-west would also be reliably reinforced. These two classifiers, plus those required to set the register properly, would be sufficient to disambiguate the aliased states.

The hypothesis is that, given this addition to classifier syntax, and without keeping histories, the system will simply *evolve* exactly the classifiers needed! It seems that this is indeed the case, or at least that it is true enough to merit considerable further investigation. In the next few slides I will show several non-Markov environments in which high performance has been reached by this method. ♣

Here is Woods101 again. The graph shows steps to goal—i.e. performance—in two regimes, learning and test. During learning, up to 6500 problems, the system alternates between explore problems and exploit problems. Then, at 6500 problems, the regime is switched to test and all problems are pure exploit. Notice that performance goes to optimal or very nearly so. The graph averages 10 runs. When the populations for individual runs are examined, classifiers that set and read the register appropriately are indeed found.

In both regimes, the graph plots performance on exploit problems. Interestingly, it is not until all exploration is turned off that the exploit performance goes to optimal. Before that it is not far from optimal. But the presence of explore problems—which explore changes in the register settings—is enough to mess up the intervening exploit problems somewhat.

I should mention that exploration during learning consisted only of trying different *external* actions randomly. Internal actions were selected deterministically. This meant they were only explored by the action of the genetic algorithm. To get the system to solve non-Markov environments, it was necessary to restrict exploration of internal actions in this way. The implications are interesting, but I have to leave them out because of time. ♣

Woods101.5 is more complicated. It has four aliased states, in which the input forms the cross pattern at right. With four states, one would expect a 2-bit internal register to be sufficient. The results for that case are shown in the upper curve. When the test phase is switched on, performance improves and flattens, but the level is not optimal. However, when the register size is increased to 4 bits, optimal performance is reached, as shown. Two bits may not work because the coding is too "tight". I.e., the right classifiers have to be found for four separate locales. Once they are found for three, the last one's coding must, independently, be picked just right or it will be confused with one of the other places. However, with 4 bits of register, there are 16 possible codings, and finding a consistent set would be easier. ♣

Finally, Woods102 is a maze with two separate 4-member groups of aliased states, as shown. So a 4-bit register would theoretically be sufficient. Again, however, optimal performance was only reached when the register size was increased to 8 bits. Even so, this is a pretty tricky non-Markov maze, and getting a solution still seems rather amazing. Adaptive internal state appears to work, and further investigation is important, in my opinion. [BREAK] ♣

Learning classifier systems have a number of exciting future directions that I haven't discussed. I will briefly mention six.

First, it appears possible to generalize the classifier architecture in an interesting way that could give more powerful generalizations in some problems. It would also allow for continuous actions instead of discrete ones. Notice that in the classifier shown, the prediction is not a fixed scalar, but a *function* of the input and the action. This is the essential notion of a generalized classifier.

Even with traditional classifiers, you can think of the population as forming a *mapping* from inputs and actions to predictions. With traditional classifiers having scalar predictions, the mapping can never be better than piecewise constant. But the environment itself may well call for a smoother representation. That is possible with generalized classifiers if, for instance, $p(x, a)$ is a linear function. This results in the evolution of a piecewise-*linear* approximation of the payoff function, more accurate than any piecewise-constant one.

How can you get continuous actions from this? It's a bit involved, but the essence is, given a match set, maximize each classifier's payoff with respect to a. Then pick the best maximizing action. Any action value within the *action restriction* $r(a)$ is possible. This frees a system of generalized classifiers from the constraint of a finite set of discrete actions. It works because the payoff is a continuous function.

The next direction is that of anticipatory classifier systems. Here the big difference is that the system predicts, not a payoff (although it might do that too), but the *next state*. Why is this interesting? Because if your classifiers can predict next states, it should be possible to *plan*, i.e., to follow out a sequence of actions to see the consequences without actually taking those actions. There is by now quite a bit of good work on anticipatory classifier systems. Two main variants exist. In one, the entire next state is predicted by each classifier. I.e., the prediction is a *vector* predicting the next input (if that action is taken). In the other variant, each *component* of the next state is predicted by a separate classifier system, and the results combined. This would seem less efficient, but in fact the single-component predictor may have higher generalizing power so the individual classifier systems are smaller.

A third direction is continuation of the non-Markov research so as to create a *hierarchical* classifier system. In such a system some actions would be elementary ones, but others would be *calls* to sequences of classifiers that would perform *sequences* of actions. This would implement the idea of behavioral hierarchies in a classifier system. Obviously, most actual behavior is in fact hierarchical. All of this appears to be within the scope of classifier systems. Control would be along the lines of the register idea. The register setting would symbolize an *intention*, or the name of a subroutine or behavioral module. It is easy to imagine very sophisticated systems like this. With enough work, we could make them a reality.
[BREAK] ♣

Yet another direction is research on the fundamental theory of XCS. This has been started in recent papers, including some in this conference. We now understand much better how accurate classifiers are evolved and why they tend to be maximally general. This work is leading toward definite ideas about the *learning complexity* of XCS. The basic hypothesis is, as discussed earlier, that XCS learns in times that depend on the complexity of the *problem itself*—the complexity of the *target function*. The times do not depend on the size of the problem space, as they do for many well-known learning paradigms. Confirming this hypothesis is very important to the future of classifier systems, in my opinion.

The fifth direction is toward basic nuts and bolts improvements in the internal mechanisms of XCS. It appears that the accuracy measure can be improved. Tournament selection and uniform crossover appear to be good ideas, for justifiable reasons. Problems for XCS on long paths need attention, and some ideas have been proposed.

Finally, it is important to continue research aimed at comparing accuracy-based systems such as XCS with the traditional strength-based classifier systems. Obviously I have tilted toward the former in this presentation. But there is good work being done on the virtues of strength-based systems. ♣

Now, 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 or radial-basis-function 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" turns out very important as XCS is applied to data inference and other areas where understandability to human users is vital.

Sixth, 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.

Finally, XCS has a powerful generalization capability. This is probably the deepest aspect of XCS. It permits the scale-up and transparency just noted. Can it be extended to any problem domain? I think so, if the classifier condition *syntax* is chosen to reflect the structure of the domain.

Thanks for this opportunity to speak to you.