A Learning Classifier System adapted for Hold'em Poker

Colin Reveley

Birkbeck College University Of London

September 2002

Contents

Introduction

This paper stems from the observation that in a game of poker, if our opponents follow a strategy and do not play randomly, there must be some set of rules such that if we follow those rules we will maximise our profits. Perhaps a Genetic Algorithm would be able to discover such a set of rules. This report looks into how feasible such an approach might be, and after some analysis, makes an attempt at an implementation.

Section 1: Poker

1.1 Hold'em Poker

The basic principle of the game is simply that cards are dealt to each player and they then place bets on whose cards are the strongest. The number of and fashion in which cards are dealt varies from game to game, as does the amount it is necessary or permitted to bet. After all the cards have been dealt, the player whose cards are ranked highest receives the sum of the bets, and all others lose their contributions.

Many variations are possible on this theme. The variant I have chosen is Hold'em poker, since it is the most commonly used competitively and has been studied by AI researchers already. From now on, I will describe Hold'em poker specifically.

The pot:

Is the pool of bets. The victor of each hand takes the entire pot, although it may be split between two players in the event of a tie.

A Game:

Is a complete game of poker, although many such games will normally be played in succession. I will refer to such a succession of games as a poker "session". The winner of the session may be considered as he who has the highest net gain at the time play ceases. A game consists of four rounds. Each round consists of a number of cards being dealt, and bets being placed by each player (often referred to as a "round of betting") on the new state of affairs presented by the new card/s. Semantically, a bet may be considered an expression of the level of confidence that we wish to express in the value of our cards. Poker strategy revolves around this idea. If I say "I bet ten", I am directly implying that I believe my cards will most likely beat yours, and if you bet more than that you are making a similar counter-claim.

The button:

After a game, the "button" is moved one place to the left. In a human game the "button" would be the position of the dealer, but in our computer games the button is nominal and is there to determine the sequence in which the players place bets. After the button is moved, player two now becomes player one and so on. This is done because the order in which players bet may be significant. There is some advantage

in being the last to bet, because then you have more information about your opponents intentions.

Betting Round:

Play proceeds around the table with players placing money in the pot when it is their turn . At any given time the total amount of money a player has contributed to the current round may be described as his “current round bet”, or in poker terminology simply his “bet”. Each player, when it is his turn, may elect not to place any more money in the pot. In this case, he “folds” and may not continue playing. He loses his contributions to the pot so far. If he does not fold, he places enough money in the pot so that no player's round bet exceeds his (though his may exceed theirs). The round stops when the round bets of all players that have not folded are equal.

Types of card:

A "community card" is a card that belongs to all players' hands simultaneously. They may all use such a card in constructing their hand, and in a human game these cards are placed face up in the centre of the table. The only other type of card in Hold'em is a hole card. Each player has two of these, and they are private: Opponents do not know what they are.

A Hand:

Confusingly, a game is often referred to as a "hand" of poker. This is confusing because "hand" is also the name of the particular set of cards that a player holds. I will use the term "hand" to refer to the cards a player holds, and "game" to refer to hand in the other sense.

All poker hands consist of five cards, even if more cards than that are dealt to a player. If a player gets seven cards, as she would in Hold'em, the best five card poker hand is made from those seven cards. The other two cards are now completely irrelevant, so you cannot say "we both have an ace high flush, but my sixth card is a ten and his is a nine, so I win". In that instance, it would be a draw. I do not describe the hand types here, since my analysis of the game does not require it.

Actions:

There are really only three possible actions in poker. However, poker terminology describes five actions. The additional two are each subsets of one of the original three.

The three actions are :

Call: match the current amount required to stay in the game

Bet: bet more than the current amount required to stay in the game, thereby increasing it.

Fold: decline to match the current bet, and thus drop out of play, losing the money you have put in the pot so far.

The other actions mentioned in the terminology are:

Check: which means to call a bet of zero

Raise: which means to place a bet that is higher than the current amount required to stay in the pot, when that amount is more than zero. One does not usually "raise" when there is nothing in the pot, but otherwise "raise" means the same as "bet".

The word "bet" is also often used to describe the quantity of money that a player is required to contribute to the pot to stay in the same: "the bet is ten" for example. This is, again, rather confusing. However, this double meaning is ubiquitous in poker and I will hope that the sense in which the word is used can be inferred from context. Poker terminology was not designed with scientific study in mind. In any case, the two meanings of the word "bet" are not unrelated.

The four rounds in Hold'em each have their own characteristics. It should be clear that each one is followed by a round of betting. The rounds are, in order:

The pre-flop: each player is dealt two hole cards, face down

The flop: three community cards are dealt face up

Fourth street: one more community card is dealt face up

Fifth street: another community card face up

Showdown: hands are compared, the pot going to the highest ranking hand.

If there is only one player left in the game at any point (because all the others have folded) she takes the pot.

On each pre-flop, the player to the left of the button (the button being the "dealer") must put in five dollars. This is known as the "small blind". The player to his left must put in ten dollars. This is the "big blind".

All bets and raises until fourth street are in increments of ten dollars.

Bets on fourth and fifth street are in increments of twenty dollars.

If no one raises the player who put in the big blind, he may raise himself.

Checking is permitted, as is checking and subsequently raising when another person bets.

1.2 Human Poker Strategy

Poker is a strategically rich game. A full analysis of strategy in Hold'em is beyond the scope of this project. However, some awareness of common strategies will inform us on what properties a learning system would need to have in order to successfully discover and use those strategies. I have based this section on Sklansky, 1976.

Slow playing and Check raising

Slow playing and check raising are similar strategies, although one strategy works at the round level, while the other spans rounds. Slow-playing means that we bet low or not at all for an entire round, hoping to entice players into remaining in the game. Since the aim of the strategy is to keep players in the game and thus increase the total amount in the pot, we must be confident that we have the best hand, and that our hand will remain the best for the duration of the game. A variant of this strategy is where we try to induce a bluff on the part of some opponent.

Check-raising is a similar strategy where, with a strong hand, we check and hope that someone places a bet subsequently. We then raise that bet, and hopefully increase the total amount in the pot that round.

Semi Bluffing

This strategy is used when we do not think our hand is the strongest at present - but has a good chance of becoming so as more cards are dealt. We bet high in this strategy: if all opponents fold, we take the pot. If they do not fold, we place our hopes in the possibility for improvement in the hand as more cards are dealt.

Pot Odds

A good player notes how much money is in the pot. He then weighs this against his chances of victory. Say the ratio of the amount of

money we are required to put into the pot to the amount of money that is in the pot is 6 to 1. Say also that we have 5 to 1 odds of winning in the showdown. The correct move is clearly to put the money in the pot, since overall we will make a net gain.

Bluffing

Bluffing is a fundamental poker strategy. Bluffing is like semi-bluffing, except that we do not expect our hand to improve enough to win in a showdown. Bluffing is usually used in later rounds of play to try to drive out the remaining players and take the pot. Certainly, if the pot odds warrant it, some bluffing is essential. But, to quote from Sklansky: "If you are playing with the same players every day, you should occasionally bluff even when the odds don't seem to justify it. This makes it more difficult to read your hands in the future".

Raising

Raising simply means increasing the current bet. There are four strategic reasons why one might want to do this:

i) To increase the size of the pot

ii)To drive players out, and thus take the pot.

iii) To bluff

iv) To get information.

1.3 Summary

Sklansky covers the possible strategies in considerable detail. It does not seem to me necessary to do so here, since it is unlikely that the system I propose will be capable of such nuances. Suffice to say that all of the above strategies are played contingently upon factors such as the player's position relative to the dealer (i.e. betting order), the money in the pot, which rounds the players are in, and, hopefully, with some regard as to the way specific players have played in the past. We must also note the variety of possible strategies in poker, and the way that use of these basic strategies becomes nuanced by particular opponents, which means that an approach to the development of a poker player which utilizes a fixed algorithm can only take us so far. Certainly a skilled human player would be able discern the workings of such an algorithm and adapt her play accordingly. Adaption of some kind will be required in an artificial poker player capable of competing at a high level. An adaptive rule based approach, a classifier system, looks like a

fruitful avenue for research.

1.4 An Anatomy of the Environment

In order to analyse what properties a classifier system (CS) might need to compete successfully, some preliminary analysis of the environment, (poker), and what a system might need to succeed in that environment is necessary. While a CS is supposed to be a domain independent learning system, it does not learn equally well in all types of environment. Different approaches to classifier system design have their own strengths and weaknesses for particular types of environment; these will be considered in later sections.

I have identified five attributes of the environment that make it difficult for an CS to learn. See also the above section on human poker strategy. These issues will be explored more fully in my discussion of alternative classifier system architectures below.

a) A human player would consider the cards in her hand carefully. This would be difficult for a CS because of the large number of possible hands. Some method of conveying information about hand value succinctly, must be found, and presented to the system. I used the methods devised by the University of Alberta for this purpose, see the section below on the Alberta algorithm.

b) Semi-bluffing requires a notion not only of current hand value, but future hand value as more cards are dealt. Some hands have a greater potential for improvement than others. For instance, we might have two cards of the same suit in the hole, and two on the board. If there are two more cards to come, we have a high chance of gaining a flush, which is a formidable hand. As with a), some method of assessing hand potential needs to be discovered and also an appropriate method of presenting that information to the classifier system needs to be found. Again, I used the techniques devised at Alberta for this.

c) Reward is not Boolean, but is rather there is a range of possible rewards bounded at one end by the maximum possible pot size, and at the other end by the maximum amount it is possible to lose in one round. In fact reward varies in two ways:

i) the size of the pot (reward) varies according to player behaviour. I call this "type one" noise.

ii) whether or not the system wins the game (and thus takes the pot) or loses (and thus loses its stake) varies. I call this "type two" noise.

While it is true that we want the CS to learn how much reward to expect, there is an element of luck in how much reward will be attained. This element of luck constitutes "noise" in the reward. Point ii) is particularly important, because large variation in reward will occur as a result, even when classifiers are well adapted to the smaller variations in reward that occur due to point i). We need to find a system that is robust in the face of these noisy rewards.

d) Some strategies involve some concept of deception, e.g. slow-playing, check raising, semi-bluffing. To implement each of these strategies, the poker playing agent must be given the capacity for intention. Crucially, each set of state variables needs to be evaluated separately in the context of each possible intention: The environment of the CS consists of private information, such as a measure of how strong a hand it has, and public game state information. In the absence of an intended strategy on the part of the CS, the environment can be considered a partially observable Markov environment. It is partially observable because we don't know what cards the opponent has, and that has an impact on what events will occur in the future. This is the root cause of the noise mentioned above: if we had complete knowledge of the environment, correct actions would not be so difficult to determine! Nevertheless, the environment remains Markov. Once deception is introduced, there is an ambiguity: each state can be considered in the presence of a decision to, e.g., semi-bluff on the part of the agent, and in the absence of such a decision. Thus, the environment becomes non-Markov. Some method of disambiguating these ambiguous states must be found.

Additionally, information needs to be made available from which the agent might base its intention. This information must take the form of a history over a number of rounds of play - if the agent bluffs every round, this will also be picked up on by an opponent, but if the agent places its decision to bluff in the context of a number of rounds, it may vary its tactics in such a way that the opponent cannot predict the agent's likely strategy. At the least in the case of bluffing, a history should be maintained between rounds.

e) In poker, a strategy of "death by a thousand cuts" is often favoured by strong human players. That is to say, consistently winning small amounts in a large number of games is preferable to attempting to win large amounts in a small number of games. This ties in with point b) ii) because we want the system to maximise average reward over the total number of hands played and to consider how much reward will be lost in the case of a loss as well as how much will be won in case of victory. I present an alteration to the XCS classifier system that allows it both to filter the noise from the win/lose noise in the reward, while simultaneously implementing the basis for a "death by a thousand cuts" strategy.

Section 2: Classifier Systems

The system I chose to base my work on was the XCS classifier system. In order to justify this choice and as a basis for further discussion, what follows is a detailed account of the workings of a standard XCS system. This is not a complete system intended to play poker; I propose alterations to the system later in this report after I have examined the issues further.

2.1 Basic XCS Architecture

This explanation of the basic architecture of XCS is based primarily on (Wilson95 and Wilson98)

Environment

The environment is represented by n boolean state variables. Where a boolean variable is insufficient to represent some aspect of the environment, a collection of variables may be used. For example, in a maze learning task, where the maze is implemented ad a grid of cells, we may have four kinds of obstacle, two kinds of reward, and empty space. each kind of obstacle may be represented by a set of booleans, e.g. 000 = obstacle type 1, 001 = obstacle type 2, 111 = reward type 1.

2.1.2 Performance Behaviour and XCS Components

XCS contains a population, [P], of classifiers. Each classifier has a condition and an action. The condition consists of n characters in a ternary alphabet {1,0,#}. It attends to n boolean state variables which may take either value in the alphabet {1,0}. For a classifier to match the set of n state variables, each character {0..n} must either be the same character as the state variable (i.e. state variable 1 = 1, character at position 1 in the condition = 1) or a #. The # indicts that the classifier "does not care" about the value of that state variable. The action of the classifier is represented by an integer. The number of possible values varies according to how many actions are appropriate to the environment. In poker, there are three actions: fold, check/call, bet/raise. Therefore, classifier actions are integers between 0 and 2.

Each classifier has three associated values. These are: the prediction (expected reward)p , the prediction error e(discussed below) and the fitness for the GA, F(discussed below).

When an input is presented to the system, a match set [M] is formed from [P] consisting of those classifiers whose conditions match the input in the manner described above. If no classifier matches, a matching classifier is generated. This is termed "covering". The system then forms a system prediction P(a[i]) for each action a[0..n] where there are n actions, and i is the ith action. An array of n elements, the prediction array, is formed mapping the actions to the system predictions. The system prediction for action a[i] is a fitness weighted average (i.e. weighted by F) of the predictions of each of the classifiers in [M] which advocate action a[i]. The prediction array contains one entry for each action a[0..n] that is possible. If no classifiers in [M] advocate an action a[i], then the entry for that action in the predictionn array is 0.

The system then chooses an action to perform from the prediction array. There are two ways in which the system may choose an action. The first is deterministic. In that case, the system chooses the action which has the highest system prediction, i.e. that action which has the largest value in the prediction array. Deterministic action selection is used when we want the system to give us its guess as to the best action to perform under a given state . This is described more fully below. The alternative is probabilistic action selection, where an action is usually selected completely at random from the prediction array. Probabilistic action selection is used when we want XCS to"experiment" with different actions in order that the learning component of the system might gain information upon which to update p, e and F for classifiers advocating an action. When to choose between deterministic and probabilistic action selection is an example of an "explore or exploit" dilemma. In my experiments, an epoch was explored or exploited with a probability of 0.5.

After an action has been selected, an Action set [A] of classifiers in [M] whose actions are the same as the action selected from the prediction array is formed. The action is then executed in the environment. In a single step problem, a reward is always passed back after an action has been executed, while in multistep problems a reward is passed back only at the end of an epoch.

2.1.3 Reinforcement Behaviour

Thus far, I have described the performance behaviour of XCS. The performance behaviour is the subset of behaviour that does not involve learning, rather the execution of rules present in the system, perhaps with the aim of maximising reward, or for gaining information about the environment. The reinforcement component of the system consists in updating the prediction p , prediction error e, and the fitness F of

classifiers in the action set [A] in single step problems, and in updating p e and F in the previous step's action set, [A]-1 for multistep problems. Multistep problems are those in which reward does not come to the system each time an action is executed, but rather after a sequence of actions has been performed. In poker, at least four actions (one for each round of betting) must be performed before payoff is received. It is therefore a multi-step problem. More actions may need to be performed than the minimum four in the case that, for example, the CS's opponent raises our bet. In multi step problems, The technique used to update p is Q-learning, and a brief digression to explain that technique is therefore necessary.

In Q-learning, an agent learns a value function [S] X [A] -> [V] where [S] is the set of states, [A] is the set of actions and [V] is the set of values for the function. Consider a maze, implemented as a grid of cells, in which the agent's task is to find some goal, g, from a random starting point in as few a number of steps as possible. Clearly, if the agent is one cell south of g, then a move north will result in the attainment of g and must have the highest value possible in [V], since no other move possible in [A] can reach g in fewer steps. Let us take that value to be 1, and determine that 1 is the highest value possible in [V]. Let us now say that the agent has found itself not one cell, but two cells to the south of the goal. In that case, we value a move north as a constant value ,q, times the value of the cell that is one step from the goal. Say that q is 0.9. Then, the action/state pair for a move north when the agent is two steps south of the goal will map to 0.9 * 1 = 0.9 under the value function for the maze. Similarly a move to the north from a position three steps to the south of the goal will be valued at 0.9 * 0.9 * 1. We want the classifier system to learn the value function as described.

In order to update the classifiers using q-learning, we must know what the value of the succeeding position that the classifier will find itself in will be. Therefore, for multistep problems, updates are done on the previous step's action set, [A]-1. A classifier j's prediction (that classifier being in [A]-1) p[j] is updated by taking the maximum system prediction in the prediction array for the current step, reducing it by a discount factor (i.e. q above), to get a value P. In terms of the above definition of Q-learning, P is that member of [V] mapped to by the classifier's action and condition (which can be considered members of [A] and [S] in the above description). The prediction p[j] then updated using the value P. In terms of the q-learning example above, if the CS finds itself three steps south of the goal, then P is the value of the cell directly to the north. The widrow-hoff delta rule is used in the update to minimise the error. So the prediction p[j] for classifier j in [A]-1 is updated

p[j] <- p[j] + (Learning rate * (P - p[j])).

In a single step problem, these updates are done in [A], rather than [A]-1. Clearly in that case, P is not determined using Q-learning, but rather is simply the current reward presented to the system. This is also the case if a multi-step problem in fact takes just one step.

Before the prediction is updated however, we must calculate values for e and for F. e is calculated not on the basis of the new prediction, but the old (since, of course, we're trying to determine in the case of e what the error in the classifier's prediction was). So e is a measure of the discrepancy between a classifier's prediction, p[j] and the reward actually gained when the action was taken, P. The Widrow-Hoff technique is used again to adjust e towards the absolute difference P-p[j] for classifier j. so

e[j] <- e[j] + (learning-rate(P-p[j] -e[j]))

N.B Widrow-Hoff is used to update p & e only after 1/learning-rate adjustments have been made to a classifier. Otherwise, the new value is just the average of the new value and the old value. The purpose of this is to allow the values to converge more quickly to a "true" average early on in a training session, when the values are usually random.

To update F, the accuracy of a classifier needs to be computed, using the error. Let accuracy be termed k. k[j] is a negative power function of the current value of e[j]. The power used is 5, so

k[j] <- e[j]^-5

next, the relative accuracy of the classifier is calculated. This is done by dividing the value k[j] by the sum of the accuracies of all the classifiers in [A]-1. Let relative accuracy be termed k'. Relative accuracy is used to determine a new value for F. Relative accuracy is used because we want to calculate a classifier's accuracy not in an absolute sense, but relative to other classifiers which match the same state variables, and advocate the same action. This is in keeping with XCS's stated goal of developing environmental niches - i.e. allowing a classifier to have a high fitness, even where the action it advocates in the circumstances to which it relates results in low payoff. So,

F[j] <- F[j] + learning_rate*(k'[j] - F[j])

Just as with the calculations for p and e, this procedure is used only when a classifier has been updated 1/learning_rate times. Otherwise, F[j] is the average of the current value for k'[j] and the previous value for k'[j].

2.1.4 Rule Discovery

Rule discovery is, of course, by means of a genetic algorithm. Unlike other classifier systems, the GA is not run on the population as a whole, [P], but rather on action sets [A]-1 in multi-step problems and [A] in single step problems. So, the GA is in effect searching for rules with high accuracy in prediction of payoff P, within the set of classifiers whose condition and action are in a particular subset of [the set of states] X [the set of actions].

2.2 Analysis and comparison of classifier systems

Space limitations prevent a full description of other classifier systems. However, my choice of XCS was informed by an understanding of two other architectures commonly used. These are the "Michigan" system and the "Pitt" system. The use of a Pitt system was considered with particular care, because it has been used successfully in the past on a poker task, though the task in fact was "draw" rather than "hold'em" poker.

2.2.1 Michigan Systems

A standard Michigan system differs from XCS in four respects which are relevant to the present problem.

a) Classifier fitness is not based on accuracy, but on strength, which is in the present case directly related to how much money was won when a classifier's action was executed. Strength is a function of how much reward was gained when the rule fired, but it is not a simple function. Credit (strength) is apportioned to classifiers through the bucket brigade algorithm. This may be conceived of as an "information economy" where classifiers buy the right to execute their actions with payments based on strength, and receive payments based on the amount of reward that was received. The bucket brigade is a complex and subtle algorithm; this is not the place for a full explanation of it

b) A Michigan system contains a message list. In XCS, the match set is formed from classifiers which match the present values of the state variables. In a Michigan system, the values of the state variables are posted to the message list. The actions of Michigan classifiers may be external actions, but they may also be further messages to be posted to the message list. Classifiers, when matched, bid for the right to post messages to the message list. A rule may match a message posted on the message list by another classifier. It is therefore possible for a classifier to attend only to internal messages. This allows an internal representation of the problem domain to be formed - something not possible in XCS as it stands. In XCS, the simpler Q-learning algorithm

is substituted for the bucket brigade.

d) In a Michigan system, the genetic algorithm, when run, is run on the entire population of classifiers, whereas in XCS it is run only on those classifiers in [A] for single step problems, or [A]-1 for multi-step problems.

2.2.2 Assessment of Michigan systems

Fitness based on accuracy, as in XCS, has the advantage that a rule may have a high fitness, while still gaining a small reward - provided it is accurate in its prediction of that reward. This is important in poker. In poker, not all environmental states have the same potential for reward. The first way in which reward may vary is that the size of the pot may vary, see above. This is important because of the panmatic application of the genetic algorithm in a Michigan system. Under that regime, classifiers with low strength would get fewer opportunities to mate, and might be driven to extinction - despite possibly representing an optimal solution to the problem posed by a particular set of state variables. Consider a situation in which, in round two, four opponents remain in the game rather than just one. In the former case, the reward may be increased by a maximum of 600, while in the latter only by 150. Rules which evolved to deal with the latter situation would be unlikely to survive, since they can never get a reward greater than 150 though they may nevertheless be near optimal for that situation. The primary consequence of this is that a "gap" may develop in the CS's covering of a problem domain. Where such gaps exist, the CS will need to develop completely new rules which have not been the subject of a genetic search. Such rules will be random, and therefore unlikely to provide an appropriate response to the situation, resulting in low performance for that subset of the problem domain. In the case of poker, I would expect this problem to have a quite significant impact on performance as a whole, since the maximum reward possible in different situations may vary widely.

There is a further, related problem. A rule may activate in several problem instances which are clearly distinct in terms of how they need to be addressed, but are identical or similar in respect of the particular subset of the input space to which that classifier attends. Such a classifier may behave appropriately in some situations,but inappropriately in others. This is the problem of "over-general" classifiers. It may be that such a classifier gains high rewards in the situations where it behaves appropriately, and thus guarantees that its action will be executed even in the situation in which it performs poorly. If enough classifiers exhibit this property, then overall system performance will be adversely affected.

2.2.3 Pitt Systems

Both XCS and Michigan system work with populations of rules. Where rules need to work together toward a common goal, an algorithm is used to link rules together, and to apportion credit between them. The system is only as effective as that algorithm permits. This is often referred to as "the credit assignment problem". Pitt systems are designed to side-step this problem.

Rather than running the GA within sets of rules, and relying on a separate algorithm to co-ordinate those rules, a Pitt system runs its GA on sets of rules. That is to say, each chromosome is a set of rules, rather than a single rule. That set of rules is a complete program for, in this case, playing poker. The Pitt system maintains a population of sets of rules. The fitness of a set of rules is its winnings when it is followed. Crossover occurs between chromosomes.

When the performance of a chromosome is being assessed, it is presented with an instance of the problem, in this case a particular game of poker. The Pitt system does have a message list: Individual rules within a chromosome match and post messages much as they would in a Michigan system. An internal representation of the problem can thus be produced by a Pitt system.

Each chromosome in the population is presented with the same instance of the problem. The fitness of a chromosome is proportional to, in this case, its winnings.

2.2.4 Assessment of Pitt Systems

Pitt systems have been shown by Smith(1981) to make effective poker players, Although Smith used a different variant of poker. However I chose not to pursue this direction of research for two reasons

i) It has already been done.

ii) A Pitt system would be a great deal slower than a Michigan or XCS system

Ultimately, I would like to produce a system that could adapt to human play in real time. There is no hope of a Pitt system ever achieving this goal. In fact my research has determined that an XCS system cannot achieved this goal either - but that was not known at the start. Even aside from the issue of speed, each chromosome needs to be presented with the same problem instance, and this would be impossible against human opponents.

2.2.5 Assessment of XCS

The problems with Michigan systems, at least in terms of poker, are I believe at least partially addressed by XCS. The combination of accuracy as a measure of fitness, and a niche GA make XCS particularly suitable for use in poker, because of the above mentioned variance in reward (i.e. variance in pot size) We want the system to choose the most profitable action possible in the particular situation in which it finds itself. In XCS, a classifier which predicts a low reward may have greater fitness than one which predicts a high reward. Also, running the GA in the action set has the advantage that competition is between classifiers which relate to the same environmental niche, or more formally between classifiers who attend to the same subset of the power set of the state variables. Crossover does not occur between classifiers whose role is unrelated. Over generality of the type possible in a Michigan system does not occur, because an over general classifier will be pushed out of the population in favour of a less general, more accurate classifier which will have higher fitness.

XCS is not without its problems, however. It has been observed that XCS tends towards classifiers that are maximally general. This is, again, the result of the GA being run in the action set. Assuming that the regime in XCS does produce correct estimates of classifier accuracy, it may be observed that no classifier, a, may be less accurate than a classifier, b, if it is more specific, since it is predicting payoff over a smaller number of situations. A corollary of that is that more general classifiers will occur in more match sets and more action sets. A more general classifier therefore has more opportunities to reproduce. Thus, there is pressure in the system towards generality. There is also opposing pressure toward accuracy. XCS theorists claim, therefore, that XCS searches effectively along the line between accurate classifiers and general ones, discovering classifiers which are "optimally general".

Normally, this would be a desirable property. However ,this tendency towards generality re-introduces the problem of over-general classifiers to XCS with regard to the present problem domain because of variance in reward of the second type described in part 1, or "noise"

What happens to XCS in the face of noise in the reward is subject to controversy at present. However, most seem agreed on these facts. If the noise in the reward (i.e. the amount by which reward can vary due to random factors) is below 0.5, then XCS should be able to handle that. This is because *relative* accuracy can still be computed. Consider that the average level of accuracy of a classifier in an XCS system where the is no GA will be 0.5. If the noise level is below 0.5, then a classifier can still attain higher fitness by

being more accurate. Thus if a GA is applied under those circumstances, the population will fill with classifiers who are more accurate than 0.5. If however the noise level is at or greater than 0.5, then no classifier can attain a greater than average accuracy. In these circumstances, the pressure toward generalisation mentioned above, whereby more general classifiers have more chances to mate, will cause the population to fill with over-general classifiers. The GA will effectively choose a random member of the Action set each time it selects a classifier for mating - so those classifiers which appear in more action sets [A] will predominate in [P]. I consider that "type two" noise in poker may well preclude an accuracy of greater than 0.5 on the part of classifiers in the system.

2.3 Alterations to XCS

There are three major alterations that I propose to make to XCS in order to make it capable of performing well at the poker task. Two of these are alterations that have been implemented elsewhere - using real values for input, and implementing a bit register - and the other has been conceived of by me for the present work, with very significant help from Tim Kovacs.

a) To solve the problem of "type two" noise, I/Kovacs propose the following solution:

i) giving each classifier not one, but two predictions, one in case of a win and one in case of a loss. Predictions for a loss will be negative. When predictions/error/fitness are updated for a classifier, the win prediction is updated if the system has just won, and the loss in the case of a loss.

ii) Having each classifier keep a record of the number of times the system won when the classifier was in the action set, and the number of times the system lost when that classifier was in the action set

iii) Rather than storing only the win prediction in the prediction array, do the following: Take the ratio between the number of wins the classifier has recorded, and the number of losses it has recorded. Multiply each prediction by its part of the ratio. So the win prediction is multiplied by the win part of the ratio, and the loss prediction is multiplied by the loss part. Then, sum the two resultant numbers and take that number to be the classifier's "completed prediction". If the system's aim is to maximise reward - if it is operating in "exploit" mode - it will follow the action with the highest completed prediction.

This solution removes the large amount of type two noise described above from the reward, leaving only type one noise which I hope will

not be too detrimental to performance. This solution has a rather nice side effect: The system is now explicitly averaging reward in the case of a win and a loss. This is what a human player would attempt to do. Rather than always following the classifier which predicts the highest reward, we follow the one whose predictions, when viewed in the light of its "win:lose" ratio means that it wins the most money over a large number of games, since that will be the one with highest "completed prediction" as specified. This will implement the basis for a "death by a thousand cuts" strategy as described above.

b) Much of the information that we need to present to the LCS will be in the form of real numbers. For example, the amount of money in the pot, the number of players and so forth. A scheme therefore needs to be in place to allow the LCS to accept real numbers as input. Wilson's XCSR implements one such scheme. For performance reasons, however I chose to implement my own scheme. Each real number is encoded in a spread of bits. Say, for example, that we want to encode a number between 0 and 1. We might choose a spread of ten bits. If the first bit is set, then the number being represented is between 0 and 0.1. If the second is set, the number is between 0.1 and 0.2. Using more bits yields greater precision. Classifier conditions that deal with these spreads no longer employ the '#' symbol. Generalisation is still possible however; the condition may match more than one part of the spread by having more than one bit set.

c) In order to give the system intentions as described above, I plan the following:

i) To add to LCS a crude "message list". In fact, this will be a simple bit register. Each classifier is extended with an internal action, and an internal condition. The internal condition reads/matches the contents of the bit register. The internal action alters the contents of the bit register. At the beginning of each game, the bit register will be intitialised to all zeros (how many bits are necessary in the bit register is a matter for experimentation; one would imagine that one bit would be necessary for each strategy that we are hoping the classifier will discover - e.g. semi-bluffing, bluffing, slow playing, because each possible set of state variables can be viewed differently in the light of each of those strategies). Internal actions will be composed of symbols in a ternary alphabet, {0,1,#}. Let us say that the bit register has only one bit. Classifiers' internal action will thus also be one bit long. If the action is '1' then '1' is written to the register. If the action is '0', then that will be written to the bit register. If the action is '#' then the bit register will remain unaltered. The internal condition matches the bit register in precisely the same way that the condition matches the state variables. The bit register may be extended to account for the other strategies mentioned in part x. One would assume that one bit would be

required for each strategy, but see ox for more details on the implementation of this scheme in XCS. The implications of this scheme for the wider XCS system will be discussed further in the section on implementation and results. The essence of this system is that rules may now explicitly pass information to each other, from round to round, within a game of poker

ii) In order to keep track of the frequency which the system has bluffed a "bluff counter" will be implemented and maintained between games. Ideally, this would use information from the internal bit register described in point i) above. However, it should be clear that before the classifier system has been in operation, the bits in the bit register are meaningless. We hope that the LCS itself will invest the bits with meaning through its operation, and develop an internal language. But even once that language has been developed, it will be difficult to read and will probably differ from experiment to experiment. In one experiment the second bit may represent a slow-play, while in another it may represent a semi-bluff. Therefore, an external function would be required to determine when a bluff has been made, and increment the bluff counter. The LCS will be presented not with the bluff count itself, but rather with the number of games that have passed since a bluff was last recorded. Classifier conditions will be extended to match that number using the scheme described below. Classifiers will then be able to express the following:

If game state <i> and <ten> rounds have passed since a bluff then <bet>

I am not yet certain how to determine when a "bluff" has been played, but perhaps it could simply be a high bet with cards whose value is below some threshold value. A better system than this would link the knowledge that the system kept between games (the bluff counter) to the internal bit register described above. Perhaps a count of the number of times each bit in the bit register was set could be used. That way, the system would be aware of how often each strategy represented by a particular bit had been played, rather than just how often a bluff had been made. The main problem with that is that if we implemented the count as a bit spread as described in b) above, and we used ten bits to represent each number, this would extend classifiers' length by sixty bits if the bit register were six bits long. Learning would be slowed.

Section :3 Alberta Algorithm

Ultimately, the aim is to produce a classifier system based poker player that can learn against human opponents. However, learning requires a very large number of trials, and in order to get meaningful

results from experiments, learning needs to be assessed against some constant measure. After much research, I chose to use the algorithm "Loki" from the University of Alberta's poker research group. In its complete form, this algorithm possesses all of the attributes required to test the theories described above, and different parts of the whole algorithm can be used independently to test particular parts of the classifier system based player. Additionally, parts of the Alberta program were used in the development of the classifier system based player, in determining hand value. I present here in considerable detail the program as defined by the Alberta group in Dennis Papps 1998 Msc thesis.

STRUCTURE OF PROGRAM.

Introduction:

The program has :

A Hand evaluator.
A betting decision maker (bettor).
An opponent modeller.

The hand evaluator attempts to give a probability of how likely it is that the program has the winning hand. The betting decision maker takes this probability, together with knowledge of the game state, and decides on one of the three actions. In order to determine the probability of victory, the program enumerates over all the possible hands its opponent, or opponents, may have. It counts all those that would win, lose and tie against it. If 50% win/tie and 50% lose/tie, then a probability for victory of 0.5 is determined. However, Loki does not consider each of the opponent hands to be equally likely. Based on the betting behaviour of the opponent, it accords each hand a probability value. It is these values which are summed.

a) HAND EVALUATOR

The hand evaluator is responsible for determining the probability of victory. It has a behaviour for the prelim, and a behaviour for the three post flop rounds.

On the prelim, there are 52 - choose - 2 possible two card holdings a player might have = 1,321. Loki keeps three tables of INCOME RATES. Each table has one cell for each two card holding possible. The entry in that cell is the income rate for that holding in that table. During the prelim, Loki passes the income rate to the bettor. Each table contains the income rates for a different number of opponents. There is a table of income rates for one opponents, a table for two/three and for four or more.

The income rates in the tables were calculated off line by running 1,000,000 simulation trials for each two card holding. In a simulation run, the cards were dealt to a player A in a game with n opponents. During a simulation, each player in the game always calls to the end, unless it is the first to bet, in which case it bets. The upshot for this is that each player always bets ten in the first two rounds, and twenty in the last two. No player folds or raises the bet beyond ten. A's average winnings for a two card holding over the 1,000,000 simulations against n opponents is the income rate for that holding against that number of opponents.

After the flop, and in subsequent rounds, the hand evaluator calculates two measures.

i) How likely it is that our hand is currently the strongest held. (HAND STRENGTH)

ii) A measure of how our hands strength will change in the ensuing rounds . (HAND POTENTIAL). Hand potential may be subdivided into positive potential and negative potential (PPOT, NPOT). PPOT is the probability that we are behind at the moment, but will end up ahead. NPOT is the probability that we are ahead at the moment but will end up behind.

Hand evaluation is performed when it is loki's turn to bet, so that opponents' behaviour can be factored into it's calculations. (see below, OPPONENT MODELLING)

There are 52 - choose - 2 possible two card holdings an opponent might have = 1,321. This number can be reduced to 50 - choose - 2 in the pre-flop (because we know two of the cards) = 1225. After the flop, there are 47 choose 2 = 1,081. After the turn there are 46 choose 2 = 1,035, on the river there are 45 choose 2 = 990.

We look at our cards, and derive an integer, our HAND RANK. I have a set of routines that will produce such a number in a standard way. Essentially, the routines take every possible five card hand and rank them in ascending order, with the best poker hand receiving the highest integer, and the lowest receiving the lowest integer. Some hands have the same value as others. These hands all receive the same integer. The hand rank is thus an absolute measure: A hand with a higher rank will always beat one with a lower rank.

We then calculate how many possible hands will win lose or tie against us. we divide the number of hands which are ahead by the total number of hands and this gives us a number n such that $1 >= n > 0$. We refer to this number as HAND STRENGTH. Factoring in the hands that tie with us also by giving them half points, hand strength is defined as:

hs = ahead + (tied /2) / (ahead + tied + behind)

There is a complication though. In the introduction I mentioned that the Hand evaluator uses data from the opponent modeller in its calculations. It does so when calculating Hand strength in the following manner.

For each opponent, Loki maintains a weight array. How the weights are calculated is described below. Suffice to say that at the beginning of a hand the weight array is either full of ones, representing the fact that all two card holdings are equally likely, or it has some pre-determined set of values in it. The weights represent how likely it is that an opponent has the cards indexed. The full array of 1,321 is maintained even though there are fewer entries than that needed in later rounds. The unused entries are simply of 0 probability. The weights in the array are recalculated after each opponent action, because our estimation of what hand they hold is contingent upon their behaviour.

Rather than simply adding up the number of hands that win and dividing by the total, we add up the weights for the hands that win against us, and divide by the sum of the weights that win, lose or tie. Some pseudo code representing this algorithms is presented below.

What about the fact that there are multiple opponents?. In order to get a measure of our strength against all the opponents, we combine all the opponent arrays into a single array containing the average values across all opponents. Then, we raise the hand strength to the power of the number of players at the table (who are still in). The right approach would probably be to consider all possible permutations of opponent cards, but this would be a very large number. The averaging technique described above was shown to produce acceptably accurate results.

```
double HandStrength(cards ourCards, cards communalCards, double[] weights)
{
   int ourRank, oppRank; // our rank and opponents rank
   Cards oppCards; //opponents cards
   Double ahead, behind, tied, handStrength;

   ahead = behind = tied =0;

   ourRank = Rank(ourCards, communalCards);

   for each case(oppCards)
      {
         oppRank = Rank(ourCards, communalCards);
         if(oppRank > ourRank) ahead += weights[oppCards];
         if(oppRank == ourRank) tied += weights[oppCards];
         if(oppRank < ourRank) behind += weights[oppCards];
      }
```

```
    return ahead + (tied /2) / (ahead + tied + behind); // ties count for half
}
```

remembering that the weight array used contains the average values across all of the opponent arrays, we can raise the value returned by this function to the power of 5 for five opponents, six for six etc.

Calculation of PPOT and NPOT: Considering the situation on the flop, we have (in the array of averages) 1,081 possible opponent hands for which we have weights. There are two cards to come, so there are 990 two card additions to the board cards possible in the next two rounds. Therefore we have 1081 * 990 = 1070190 possible situations to consider. For each sub case in the weight array, we calculate how many of the 990 possible situations to come in which the opponent hold the indexed cards result in us being ahead, tied and behind. So the positive potential is the number of hands where we are behind but end up ahead after the next two cards divided by the total number of hands where we are behind. Negative potential is calculated in like fashion. Factoring in the ties, the calculation for PPOT is:

Positive potential = (total[behind, ahead] + (total[behind, tied] /2) + total[tied, ahead]) / total[behind, sum] + (total[tied, sum] /2)

Here is pseudo code for the positive potential and the negative potential algorithm, looking two cards ahead. The algorithm used on the turn only needs to look ahead one card.

```
HandPotential(Cards ourCards, Cards boardCards, double[] weights)
{
    double[3][3] HandPotential;
    double[][] HPTotal;
    double posPotential, negPotential;
    int ourRank5, ourRank7, oppRank, index;
    Cards additionalBoardCards, oppCards, currentBoard;

    ourank5 = Rank(ourCards, boardCards);

    for each case (oppCards)
        {
            oppRank = Rank(oppCards, boardCards);
            if(ourRank5>oppRank) index = ahead;
            if(ourRank5==oppRank) index = tied;
            if(ourRank5 < oppRank) index = ahead;

            HPTotal[index] = weights[oppCards];

            for each case(additionalBoardCards)
```

```
{
currentBoard = boardCards + additionalBoardCards;
            ourrank7 = Rank(ourCards, currentBoard);
            oppRank = Rank(oppCards, currentBoard);
           if(ourrank7>oppRank) HandPotential[index][ahead] += weights[oppCards];
            if(ourrank7==oppRank) HandPotential[index][tied] += weights[oppCards];
            if(ourrank7<oppRank) HandPotential[index][behind] += weights[oppCards];
        }
}

   posPotential = (HandPotential[behind][ahead] +  (HandPotential[behind][tied]/2)
            + (HandPotential[tied][ahead] /2)) / (HPTotal[behind] + (HPTotal[tied] /2));
   negPotential = (HandPotential[ahead][behind] + (HandPotential[ahead][tied] /2)
            + (HandPotential[tied][behind]) / (HPTotal[ahead] + (HPTotal[tied] /2));

return (posPotential, negPotential);
}
```

This algorithm is just pseudocode - the real version would take account of the fact that there are only 47 - choose - 4 = 178,365 possible values of oppRank in the inner loop.

To extrapolate this to many players, all combinations of cards for player one would have to be compared to all possible combinations of cards for player 2 and so on for more players. This adds an extra level of nested iterations for each player, just as in the hand strength calculations. This is not doable in real time, so we just use the calculation made with the array of averages as we do in the hand strength calculations.

b) OPPONENT MODELLING

Our strategy is essentially to update the weights in the weight array for each opponent in accordance with their betting behaviour during the hand, thus giving the hand evaluation algorithm a more realistic view of what we are up against. Statistics on behaviour frequencies are kept by the opponent modeller between hands, and these determine how we will interpret the betting behaviour in this hand. The interhand frequencies are the model of the opponent. The weight array is a model of the opponents hand, based on interhand frequencies, and the behaviour we currently observe. If we know the player bets a lot, then we can infer that he bets with lower value hands than do others, and adjust the weight table accordingly, i.e. weaken the probabilities for strong hands, and boost those for weaker hands.

The ACTION FREQUENCIES show how often a player has made a given action in a given game CONTEXT. The context is defined by a set of variables like the players pot-odds, the number of players in the game, the amount to call, the amount bet last time. Experimentation

with different variables and different algorithms to interpret those variables and update the weight table appropriately is a direction for my research, though not the primary one (described below).
I'll begin with variables and algorithms chosen by the Alberta group (described below), which are reasonably simple in principle, although there is much "devilry in the details".

At the beginning of each hand, the weight array is set to 1 for each entry (although, by setting a parameter for the program it can be filled with the income rates for each two card combination as described above). As the weights are updated, each entry represents the probability that the opponent would have played in the manner observed if they held the cards in that entry. To get a spread of the probabilities that each set of two cards is held, we can divide the entry by the sum of all the currently valid entries (valid entries change depending on what round we are in as described above - unused entries get a 0 value in the weight array).

If we assume that all players play the same (e.g. if they have a hand in the top 15% they raise) then we can do generic opponent modelling. We take their behaviour and the game context and update the weight table, without reference to their behaviour in past hands. However, not all players play the same way. Opponents are therefore modelled in the following way, using player specific information.

The context is defined by 1,2,3 + bets to call and one of the four game rounds = 12 contexts, at least for now. It might be feasible to add more later.

For each opponent, we determine the context and update the array of action frequencies. Each context is represented by Loki as a row in a two dimensional array. The actions are represented by the columns. The value for an action in a context is incremented each time we observe it.

actionFrequency[context][action] += 1;

To find out how often a player makes a certain action in a certain context, we do

actionFrequency[context][action] / actionFrequency[context](total)

At the beginning of the game, we don't have many observed betting actions. Therefore, a set of default values are used, and mixed with the observed frequencies until a threshold number of observations have been made

actionFrequency'[context][action] = (actionFrequency[context][action] * actionFrequency[context](total)/20)) +
(defaultFrequeny[context][action] * (20 -

actionFrequency[context](total) / 20))

when the threshold is 20 and we have observed < 20 actions in this context. Otherwise, we just use actionFrequency[context][action]

Now, to adjust the weights for an opponent we calculate a threshold hand strength2 for an observed context/action based on the action frequencies we have recorded.

Say that we've seen a player P play 100 games on the flop. He folded ten times, bet seventy times and raised twenty times. Let us say that this player has just raised. We want to re-calculate the weight array based on this action. So we compute the frequency of raising, which is 0.20. This makes the threshold hand value 1 - 0.20 = 0.80. So in order to raise, our player requires a hand value of 0.80 or higher. So, we get a ranked distribution of all the possible two card holdings by calculating the hand strength for each one (or use pre-determined income rate values for the pre-flop).

Now we have a mean hand value required for the action (0.80) and the hand values of all the possible two card holdings. We choose a spread value. For each possible two card holding, we compare that holding hand value with our average hand value (0.80). If that holding's value is 0.80, then our re-weighting factor is 0.5. If it is more than the spread value below 0.80, the reweighing factor is 0.01. If it is more than the spread value above 0.80 the relighting value is 1. Linear interpolation is used to determine the relighting value if the holding's value is between 0.80 and the spread value above or below it. Here is some pseudo code:

```
void reWeight(double mean, double spread, double[]weights,
gameState gs)
{
  double reweight;
  double handStrength;
  Cards hand;

  for each case(hand)
  {
     handStrength = getHandStrength(hand);
     reweight = (handStrength - mean + spread) / 2 * spread;
     weights[hand] = weights[hand] * reweight;
  }
}
```

There are further subtleties depending on round of play. Also, we don't let the weight for any hand go below 0.01. the spread value is variable. Alberta used spread = 0.4 * (1 - mean). If we hold, say a pair of aces, we must be careful not rule out those cards on our opponents behalf, since she is not aware of the fact that hold these

cards. Like I say, there is much detail surrounding the reweighing system, but I hope I've conveyed the general idea. When building prototype systems, I'll explore the issues involved with re-weighting.

Section 4 Conclusion and Implementation Notes

Implementation was not completed in time. There are two linked reasons why this is so. First, the work was more demanding that it appeared to be on paper (some six weeks was spent on implementation; the PPOT calculation and the XCS with internal memory meant that programs had extremely long run times). Second, my methodology could have been improved. To take the second point first: I have extensive experience in building web based database applications. Bugs are relatively easy to locate and correct in such applications, but I have found that this is not true of the systems involved in this project. Because time was limited, I tended to try to write large blocks of code and test only after implementation was completed. Of course, with classifier systems as with the Alberta programs, bugs are difficult to track. The system simply "does not work" and scanning through the code looking for the single bug that is causing the problem is often fruitless. The workings of a classifier system appear straight-forward, but the fact that every part of the system depends on every other part of the system means bugs frequently crop up.

The correct approach., I have discovered, is not to wait until all the pieces are in place, and test the operation of the whole system, but to custom write unit tests for each function and be sure that that function is behaving precisely as it should. While this is time consuming, it is repaid since complete re-writes (of which there were quite a few during coding) will not be necessary. It might also have been wiser to write my own version of XCS from the ground up, rather than adapting open source code on the web. This would have given me more certainty about the operation.

Also, The XCS implementation I used (XCSJava 1.0 by martin Butz) did not take advantage of the OO Java programming paradigm. For example, Butz used only one class to represent the notion of a "set". This class had different constructors for each type of set (Population, Match set, Action set). A better approach would have been a hierarchy of classes, with an abstract "Set" class at the top, and derived classes for the different types of set. I suspect that Butz was used to programming in a functional style using the C language. Indeed he has written a version of his system in C, and it is very similar

It is worth bearing in mind that I viewed this project primarily in

terms of coming to grips with classifier systems theory. I hope I have succeeded to some extent in this goal. Many ideas about how to get a classifier system to learn the poker task were mooted and eventually dismissed. Again, my research methodology could have been improved - I tended to read too widely before I had fully understood the implications of what I had already read.

I am continuing to work on the implementation, and will complete it after this report has been submitted.

The following have been successfully implemented:

1) Alteration of XCS to accept real numbers using a bit spread
2) Loki's expert betting system.

Additionally, the following were constructed but did not function as expected:

1) The internal memory system for XCS
2) The remainder of the Alberta program, i.e. the Opponent modelling system.

and the following were not completed at all:

1) coupling between XCS and Alberta to allow them to play against each other
2) Alterations to XCS to implement the "two predictions" method of noise compensation.

Once all this is completed, I plan to experiment with different state variables. Poker games will be implemented using Alberta's framework. Certainly, the CS will be passed HS and PPOT as used in Loki, and also The pot size, and a history of opponent actions during its current game (it is hoped, of course, that the rules will implicitly contain all the knowledge relating to past games that the CS might need). This can easily be implemented with a simple wrapper for Alberta's GameInfo object (see appendix) , with specific information from the players' particular PlayerInfo objects. How well the system performs with different numbers of opponents will also be examined.

Full source code is presented in an appendix, including those parts of the system which did not function. Martin Butz's code is also appended so that it is clear which parts are my work, and which his. The Alberta group provided some poker AI utilities. It should be clear from my source code where that framework has been used.

Since implementation was not completed, no experiments were done to demonstrate that the system I have presented would in fact be able to learn the poker task. Nevertheless, I think my arguments are sound.

After completion, future work will involve research to determine whether a system can be built which is capable of adapting to human strategy on a human time scale. I propose the following:

Record each state and action that the opponent makes in a number of games - perhaps fifty or so. Then simulate the required number of games for the CS to learn the problem in the following way:

When the simulated human opponent acts, we find the action in the recorded game state that is the nearest to the situation found in the simulation, and have the opponent perform that action. Initially, this will be quite an inaccurate simulation - but after fifty games have been played, the opponent will have been observed in a larger number of states, and simulation of their behaviour might be more accurate.

The primary issue here is the function for finding "similar" states and actions. In fact, what we consider to be similar about two game situations involves making judgements about correct strategy - precisely what we want to avoid. However, different functions could be experimented with, and the performance of the system could be assessed empirically with a number of different functions.

Bibliography

This bibliography contains only those sources needed for the preparation of this paper. In fact, I examined a great deal more material , but most of that was discarded – for example Martin Butz's ACS classifier system. Everything I read was exciting, but it seems pointless to include work That is not directly relevant to the points made in this report.

Optimal Classifier System Performance In Non-Markov Environments,
Pier Luca Lanzi, Stewart Wilson 1999

Classifiers that approximate Functions
Stewart Wilson 2002

An Extension to the XCS Classifier System for Stochatic Environments
Pier Luca Lanzi , Marco Collumbetti 1999

A tutorial Survey of Reinforcement Learning
S. Sathiya Keerthi

State of XCS Classifier System research
Stewart Wilson 1999

What Makes a Problem Hard For XCS?
Tim Kovacs 2001

Classifier Fitness Based on Accuracy
Stewart Wilson 1995

Generalisation in the XCS Classifier System
Stewart Wilson 1998

Dealing With Imperfect Information in Poker
Dennis Papp Msc, University of Alberta.

A learning System Based on Gen etic Adaptive Algorithms
S.F. Smith, Phd thesis U. of Pittsburgh 1980

Genetic Algorithms In Search Optimisation and Machine Learning
D. Goldberg