OVERVIEW OF XCS EFFICIENCY
(TRADE-OFF BETWEEN ACCURACY AND GENERALITY)

Jackie Jacobs
Abstract

Since the appearance of Learning Classifier Systems, there has been a great deal of literature on how to improve the learning capabilities of systems and their convergence to an optimal, general and accurate population. The Butz-Wilson X-Classifier System is a type of LCS which is known for its excellence in this domain. This paper attempts to introduce the reader to the fields of Learning Classifier Systems, genetic algorithms, particular environments and problems such as aliasing which persist in the domain. It discusses just how well XCS can deal with problems, from the suggestion that it still faces certain issues, such as an over generality and inaccuracy of dominating classifiers in some cases. The experimental work tends to imply an imperfection of the subsumption algorithm used in the XCS, which can encourage poor learning, particularly in aliased situations.
Acknowledgements

Many thanks to everyone who has helped me during the year, especially of course, Dr. Alwyn Barry.
Contents

All values represented in square brackets in this document refer to the Web resources.

Chapter 1: Introduction ................................................................. 1

Chapter 2: Genetic and Evolutionary Algorithms ............................. 4

  2.1 Definition ................................................................................. 4

  2.2 Genetic Operations ................................................................... 5

     2.2.1 Reproduction .................................................................... 5

     2.2.2 Crossover ......................................................................... 7

     2.2.3 Mutation .......................................................................... 8

     2.2.4 Deletion ........................................................................... 8

  2.3 The Genetic Algorithm Cycle ................................................... 9

Chapter 3: Classifier Systems .......................................................... 12

  3.1 Definition ................................................................................. 12

  3.2 Example of a Classifier System ................................................. 13

  3.3 Limits of a Classical Classifier System ..................................... 15

Chapter 4: Learning Classifier Systems ......................................... 16

  4.1 From a Classifier System to LCS .............................................. 17

  4.2 Limits of LCS .......................................................................... 17

  4.3 Concluding Towards XCS ...................................................... 20

Chapter 5: The Butz-Wilson X-Classifier System ............................. 22

  5.1 The XCS Model ...................................................................... 22

  5.2 How the XCS Operates ........................................................... 23

     5.2.1 Iteration mode ................................................................. 24

     5.2.2 Match Set Creation ......................................................... 24

     5.2.3 Prediction Array Creation and Action Selection ............... 25
Chapter 1: Introduction

This project was originally going to investigate the use of simulated chemicals within a learning classifier system, in the context of machine learning. This would be used to solve certain problems faced by a particular type of learning classifier system, called XCS (devised by Wilson, 1995). In the end the project investigated another way of dealing with one of the aliasing problems called the consecutive state problem, in a finite state world. This involved exploring the world of machine learning and classifier systems.

Machine learning is an essential sub-discipline of artificial intelligence. Any kind of intelligence generally requires some sort of memory, updated by learning. One way that Collins English dictionary defines learning is as gaining “by experience, example or practice”. In the context of machine learning, a program carries out experiments and remembers their outcome for future use. It improves its performance based on previous results, so as to simulate a human-like intelligence. The goal of machine learning is to exploit knowledge descriptions easily translatable into natural language. The success of machine learning appears on both the methodological and the application front. Machine Learning has enabled improvement in a number of traditional areas of computing, such as data analysis and data mining, but it has also opened up completely new domains of research. A recent success of machine learning is the feasibility of learning algorithms with understandable and analysable results. Applications which profit from the growth of machine learning as a study are information extraction from text, design of “softbots” for the internet, medicine, molecular biology, telecommunications, banking and commerce. Machine learning has allowed advances in classification and pattern recognition, diagnostic reasoning and user profiling. It is also intimately linked to automated knowledge acquisition used in the domains of digital libraries, knowledge organisation, structuring and integration (knowledge management), the design of agent systems (exploiting and sharing knowledge of a domain or task), web services and smart interfaces with semantic web access and intelligent integration and retrieval.
Evolutionary programming is a form of machine learning whose methods are inspired from natural mechanisms. This type of computation maintains a population of structures which evolve, by learning to survive, in a shared environment, according to specific rules. A learning classifier system (or LCS) is a form of evolutionary programming devised by Holland in 1975, which uses induction rules in order to learn to solve a given problem. An LCS consists of a set of classifiers, or rules consisting of a condition, an action and a strength record, which propose actions to be carried out in the programming environment. Not only does the system classify the original inputs in one way, but it improves its operation as it is running, in order to classify them in a more efficient way. It does this using a genetic algorithm and other low-level operations to maintain its population of classifiers and reach optimal response to the environment. In order to know which classifiers are the ‘strongest’, it reinforces communication with the environment. The system interacts with the environment both on input and output. The environment gives the system an input message and the system selects an action to be carried out, via its classifiers. An action is returned to the environment as output, and the system observes and records the result of carrying out the action in the environment, for future use. The method for registering results is a reward system: when a reward is reached (that is, once the system has achieved its goal) the classifiers which have helped in this achievement are rewarded, using a prediction system for all states in the environment. The higher the prediction, the ‘closer’ the state is from reaching a reward.

The Butz-Wilson X-Classifier System (or XCS) carries out reinforcement learning. It represents an improvement on classical learning classifier systems, as it solves certain issues faced by these. This type of Learning Classifier System has proved to be very successful, although some problems remain. One of these is the Aliasing Problem. This arises when two states in the program environment have the same input, yet differ in prediction, or in ‘distance’ from a reward. This makes it impossible for the classifier system to accurately assign a single prediction value to each classifier condition. To overcome these problems, only
few solutions exist so far. Lanzi used a memory mechanism which tries to disambiguate between aliased states, but this remains a very complicated device. Barry has suggested the idea of simulating chemicals which would have the effect of skipping certain steps in the system, to avoid checking the environment for input in the case of aliased states. This could also possibly be done without using the chemicals. This project investigates the possibility of carrying out the same action many times, over the chain of aliased states, using the Butz-Wilson XCS adapted to a finite state world environment.

This document starts by laying the bases of the field before going into the detail of the experiments carried out and the analysis of their results. The terms are defined and research areas exposed first, then the problems can be described and the implementation testing can be explained more appropriately.
Chapter 2: Genetic and Evolutionary Algorithms

2.1 Definition

Evolutionary computation is a model of machine learning that uses a genetic or evolutionary metaphor. Evolutionary algorithms are one kind of evolutionary computation.

An evolutionary algorithm maintains a population of structures which evolve according to rules of selection, recombination, mutation and survival, referred to as genetic operators. A shared environment determines the fitness or performance of each individual in the population. The fittest individuals are more likely to be selected for reproduction, and two ‘fit’ parents are likely to give two ‘fit’ children. The fittest individuals are also more likely to be recombined or mutated to generate an even ‘fitter’ individual. It is this “survival of the fittest” aspect which gives the algorithm its name, as it incorporates aspects of selection found in nature and the theory of evolution.

A genetic algorithm is a type of evolutionary algorithm devised by John Holland, which generates each individual from some encoded form known as a "chromosome" or "genome". This selection and recombination routine generates new classifiers from existing population members. Chromosomes are combined or mutated to breed new individuals. The programming environment creates a population of individuals represented as character strings, comparable to base-4 chromosomes in DNA. The population then goes through a process of evolution, as its individuals are changed or suppressed, or new individuals are introduced by a genetic algorithm.

In practice, this can be used to optimise parameters. The values of the parameters to be optimised are encoded into chromosome-like strings or arrays of characters, and set to evolve in the environment. In other words, they are
placed in the population and manipulated by a genetic algorithm: simple bit manipulation operations allow the implementation of the genetic operations. The optimal values for the parameters are those dominating the population at the end of the run. Genetic algorithms are therefore useful for multidimensional optimisation problems. Implementations typically use fixed-length character strings to represent their genetic information, together with a population of individuals which undergo crossover and mutation in order to find interesting regions of the search space.

In a system the genetic algorithm is used to maintain the population of classifiers and to select the outputs to be sent to the environment. Induction is the action of creating new individuals in the population, using already existing information from the population of classifiers. Suppression, transformation and induction of classifiers in the population carry out the population maintenance, and require operations on the existing chromosomal binary strings. These are called genetic operations, described in the following section.

2.2 The Genetic Operations

The most frequently used genetic operations are reproduction, crossover, mutation and deletion. (The following definitions of these operations are taken from Alwyn Barry’s LCS website glossary [7].)

2.2.1 Reproduction

Reproduction selects strings of individuals to be copied for possible inclusion in the next generation.

The most commonly used method for the selection process is called the ‘roulette wheel method’. This statistically chooses the strings based on their relative fitness values. The strings are placed on the wheel in proportion to their relative
fitness. The roulette wheel is then spun three times, and the results indicate the string to be placed in the reproduction ‘mating pool’.

The following is a very simplistic example of a roulette wheel selection on three chromosomes [10].

<table>
<thead>
<tr>
<th>String</th>
<th>Fitness</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>01001</td>
<td>5</td>
<td>19%</td>
</tr>
<tr>
<td>10000</td>
<td>12</td>
<td>46%</td>
</tr>
<tr>
<td>01110</td>
<td>9</td>
<td>35%</td>
</tr>
</tbody>
</table>

In this example the string ‘10000’ is the fittest, and should be selected for reproduction approximately 46% of the time. ‘01001’ is the weakest, and should only be selected 19% of the time. If a classifier is particularly strong, the genetic algorithm is likely to select it for reproduction, in the hope that the good elements will be kept in the population to ensure better chances of survival.

It is possible that strings (especially the fitter strings) will be selected for reproduction more than once. This desirable effect will let the population evolve into a domination of the stronger elements and a near-extinction of the weaker.

Other types of reproduction selection exist. For example we could decide to always select the fittest and discard the worst, or use other strategies (Blicke, 1995).
2.2.2 Crossover

Crossover is supposed to correspond to the kind of recombination of chromosomes found in sexual reproduction in nature. An offspring's chromosome is created by joining segments chosen alternately from each of two parents' chromosomes which are of fixed length. It thus produces two new individuals from two existing individuals in the classifier population. Two strings of attributes from each of two parent classifiers are duplicated. The copied strings are sliced at a selected point, and the head of each is recombined with the tail of the other to form two new attribute strings. This is the most common type of crossover, called single point crossover, as alleles are swapped from a single locus on the original chromosomes.

Suppose the population holds the two binary strings ‘01100011’ and ‘10001101’. Doing a crossover on these two parents at the 5th digit would create the two new children strings ‘01101101’ and ‘10000011’. Crossover can also work with the central part of a string. A crossover on the two parents above between the 3rd and 6th digits included, would create the children strings ‘01001111’ and ‘10100001’.

The parents are selected at random from the ‘mating pool’, but crossover will only take place in 60 to 70 per cent of the time. In some cases it is decided not to use crossover but to simply copy the parent strings, thus creating exact duplicates for the next generation.

The rationale of using crossover is inspired from nature. Two parents A and B, where A has nice eyes and B has nice hair for instance, are probabilistically more likely to give children with both nice eyes and nice hair! Thus if the slicing point is well selected, a very strong individual with the advantages of both parents could be produced from crossover. Of course it is likely that the crossover operation will also produce a very non-adapted individual (if it only inherits negative aspects of the two parents). Although both offspring are usually kept, sometimes only one of them is created.
2.2.3 Mutation

Mutation selects an individual from the population and randomly chooses a single attribute value to be modified to another attribute value. In binary strings, this simply means that ‘1’ is set to ‘0’ or ‘0’ is set to ‘1’. Suppose a strong classifier in the population holds the binary string ‘01100111’. Performing mutation on the 5th digit (randomly chosen) would simply create the new string ‘01101011’.

Operations such as mutation are introduced in the hope that an already strong classifier could be made even stronger. Naturally it seems more efficient to operate on individuals which are selected for their strength rather than to start at scratch without taking fitness into account, in the hope of a miracle transformation of an unfit individual into a suddenly very useful individual: one may as well just create a random string and add it to the population. Starting with a strong classifier increases the chances that positive properties of the classifier will remain. It is also possible however that mutation carried out on a strong classifier will simply create a very ‘unfit’ classifier as a result. This is why the genetic algorithm carries a mutation probability (usually very low, about 0.001%) dictating the frequency at which mutation occurs. It checks whether to perform mutation or not for each string. Although the mutation probability is kept low, to avoid destroying fit strings, it is still recognised as being very important. Any amount of crossover operations, carried out on a population where all chromosomes have the same value for one allele (one bit on the binary string is the same in all classifiers), will not succeed in changing this value to discover alternatives and new areas of the problem space. Mutation however could easily change that value, thus avoiding complete population convergence and too much specialisation, by bringing new “blood” into the population to create a greater diversity and richness of classifiers.

2.2.4 Deletion

The genetic algorithm can be used not only to maintain the population but also to select the individuals to be used for output to the environment. Keeping
unhealthy individuals in the population can only increase the chance of their being selected, an undesirable effect which can only be solved by other genetic operations. In order for the population eventually to converge to a good solution, it is important to suppress some individuals, and increase the power and standing of others. Deletion is the selection and removal of a classifier from the population. Individuals are not selected for deletion simply on their fitness value, as may be expected. Fitness is not the only important characteristic of a population. Having the entire population being extremely good at solving one sub-problem does not suffice to cover knowledge of the entire problem space. It is important to use generality of a classifier when deciding whether to delete it or not. A classifier’s generality depends on the area of the problem space which it is able to deal with. If classifier C1 knows a little about problem A and a little about problem B, it must not necessarily be deleted simply because other classifiers know much more in total. Supposing C1 was the only classifier to be able to deal with problem B, then deleting it simply on fitness issues would lead to the system losing the only information it contained on problem B. Thus deletion selects chromosomes which are probabilistically the worst.

There remains a lot of uncertainty as to when it is useful to use certain operators. In his thesis on “The Role of Mutation and Recombination in Evolutionary Algorithms” (Spears, 2000), Spears highlights the advantages and disadvantages of using these on a population of classifiers. They can be quite disruptive on the population, but do reduce the time necessary to achieve a peak, sometimes several peaks, of fit individuals. It is important to remember that the crucial point of the genetic algorithm is not simply to know what to do but to know when to do it, in which conditions and how often.

2.3 The Genetic Algorithm cycle

The genetic algorithm usually involves the following cycle:

• Evaluate the fitness of all of the individuals in the population.
• Create a new population by performing operations such as crossover, fitness-proportionate reproduction and mutation on the individuals whose fitness has just been measured.
• Discard the old population and iterate using the new population.

One iteration of this loop is referred to as a ‘generation’. There is no theoretical reason for this cycle as an implementation model; it does not copy nature, but is merely convenient.

At generation 0 (at the beginning of a run) the process operates on a population of randomly generated individuals. From then on the genetic operations modify the population (in order to improve it) at each iteration of the algorithm. Two individuals are selected for ‘operation’ based on their fitness: the higher the fitness, the higher the chance of being selected. The algorithm stops when a suitable solution has been found or a certain number of generations have passed, depending on the needs of the system.

The genetic algorithm works in the following way. It starts with an initial time and a (usually) random population of individuals. It evaluates the fitness of all individuals in the initial population, and then increases the time counter for every iteration of the following:

1. Select a sub-population which will produce offspring
2. Recombine the genes of the selected parents, thus creating children
3. Mutate the children
4. Evaluate the new fitness of all individuals in the new population
5. Select the survivors from the actual fitness

The following diagram comes from [10]:
In so doing the genetic algorithm maintains the population very effectively. It allows us to quickly find a reasonable solution to a complex problem, by searching through a large and complex search space, even for which very little is known: this is the particular strength of these algorithms. Humans may know exactly what they want their problem solution to do, but not necessarily how the solution should go about doing its job. Genetic algorithms are able to produce solutions which solve a problem in ways that would not even have occurred to humans.
Chapter 3: Classifier Systems

3.1 Definition

Classifier systems originate from Holland’s cognitive models (Holland, 1971), systems capable of classifying the goings on in their environment, and then reacting to these goings on appropriately. They are one of the early applications of genetic algorithms, as they use this type of evolutionary algorithm to adapt their behaviour toward a changing environment. A classifier system is of the form:

INPUT: $x_1, x_2, \ldots, x_N$.

OUTPUT: classification of $x_1, x_2, \ldots, x_N$.

It produces one or more outputs as a result of one or more inputs, where the outputs represent some classification of the inputs.

A classifier system works in the following way. It starts with an initial time, an initially empty message list and a randomly generated population of classifiers. It then increases the time counter for every iteration of the following:

1. Detectors check whether input messages are present
2. Compare the message list to the classifiers and save the matches
3. Process new messages through the output interface to send output to the environment

The classical classifier system generally operates in the animat approach. In this context it consists of:

- An environment
- Receptors which give information to the system about the environment
- Effectors which let the system manipulate the environment;
- An animat, which uses receptors and effectors to ‘live’ in the environment.
The environment is usually an artificially created digital world, such as a 2-dimensional grid containing "food" and "poison", as in Booker's Gofer system (Booker, 1982). The animat walks across this grid and tries to:

(a) Learn to distinguish between food and poison
(b) Survive well fed.

3.2 Example of a Classifier System [9]
Suppose the animat is a very simplified model of a cat, called Tom. Tom lives in a garden environment and interacts with it through his sensorial input detectors (eyes, ears, nose, etc.) and his effectors (legs, mouth, etc.). The computer represents the inputs received by Tom from the environment, and the actions Tom carries out on the environment, as messages between Tom and the environment. Tom is a message system which maintains a message list and converts the input messages into output messages. For this Tom has to follow certain rules, generally if-then rules, called classifiers, which constitute the computer program's classifier population. The messages and classifiers are represented as binary strings for an easy manipulation by an algorithm (usually a genetic algorithm).

The program cycle is the following. The input interface generates messages to be added to the message list. These input messages are then matched against the condition-part of all classifiers of the system’s population, to find out which actions deal with the input message and therefore need to be triggered. The message list is then emptied. The chosen actions are placed on the message list, for the output interface to check and give orders to the effectors.

Tom moves around randomly in the environment. When he sees a mouse, he tries to catch it. When he sees a dog however, he runs away. Running away from a dog has priority over catching a mouse. This behaviour of the animat is expressed in the following set of if-then rules:

- If no dog around and see mouse to the right, then run to the right
- If no dog around and see mouse to the left, then run to the left
• If no dog around and see mouse in front, then run forward
• If see dog to the right, then run to the left
• If see dog to the left, then run to the right
• If see dog in front, then turn and run

These conditions and actions can be encoded as input and output string messages for use within the classifier system. The following is a possible translation into 4-bit input strings and 2-bit output strings.

<table>
<thead>
<tr>
<th>First 2 bits of the input message</th>
<th>Last 2 bits of the input message</th>
<th>Output message</th>
</tr>
</thead>
<tbody>
<tr>
<td>'00': 'no dog around'</td>
<td>'00': 'no mouse around'</td>
<td>'00': 'run to the left'</td>
</tr>
<tr>
<td>'01': 'dog to the right'</td>
<td>'01': 'mouse to the right'</td>
<td>'01': 'run to the right'</td>
</tr>
<tr>
<td>'10': 'dog to the left'</td>
<td>'10': 'mouse to the left'</td>
<td>'10': 'run forward'</td>
</tr>
<tr>
<td>'11': 'dog in front'</td>
<td>'11': 'mouse in front'</td>
<td>'11': 'run back'</td>
</tr>
</tbody>
</table>

Using this translation the following rules can then be applied to the binary strings:

<table>
<thead>
<tr>
<th>IF</th>
<th>THEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>01</td>
</tr>
<tr>
<td>0010</td>
<td>00</td>
</tr>
<tr>
<td>0011</td>
<td>10</td>
</tr>
<tr>
<td>01##</td>
<td>00</td>
</tr>
<tr>
<td>10##</td>
<td>01</td>
</tr>
<tr>
<td>11##</td>
<td>11</td>
</tr>
</tbody>
</table>

(Assuming that the symbol '#' is a wildcard, which can denote either 0 or 1. It is used in the cases where there is a dog around: the presence of a cat then becomes completely arbitrary.)
3.3 Limits of a simple Classifier System:

However Holland’s non-learning classifier system (Holland, 1975) has limits: it consists of a list of classifiers, a list of messages, an input interface (detector) and an output interface (effector). It communicates with the environment through its detector and effector, but it does not observe the effects its output actions have on the environment. Therefore it will always operate in the same way given the same inputs: it does not evolve and adapt its classifiers to improve its operation on the environment. This is what the "Evolutionary Reinforcement Learning" paradigm tries to achieve. This incorporates Learning Classifier Systems, "Q-Learning", devised by Watkins in 1989, and a paradigm of the same name, devised by Ackley and Littman. In 1986, Holland added a reinforcement component to the overall design of his classifier system to emphasize its ability to learn.

The following chapter describes the use of Learning Classifier Systems, and the need for Wilson’s improved version of LCS, called XCS.
Chapter 4: Learning Classifier Systems

4.1 From a Classifier System to LCS

The first implementation of a learning classifier system goes back to 1978 (Holland and Reitman, 1978). A learning classifier system differs mainly from a non-learning classifier system in the following ways:

- **Reinforcement from the environment.** This is essential for learning to take place: the system must observe the effects of the output it sends to the environment in order to remember the efficiency of its action on the environment, and use the information in the future. In Holland’s learning classifier system, the activation of each classifier depends on some additional parameter, which can be modified as a result of experience.

- **Manipulation of the initial classifiers.** A non-learning classifier uses its original population of classifiers throughout a run. In order to learn, the system needs to adapt itself to the environment (now that it has a better knowledge of the environment), by only keeping strong elements in its population, and even mutating them to create even stronger individuals.

Holland’s learning classifier system schematics are as follows [14]:
The system receives messages on the state of the environment. The message list is then checked against the if-then rules of the classifier population ([N] on the diagram). The genetic algorithm sorts out the population using its genetic operations on selected classifiers. The classifiers matching the input messages (those which suggest actions to deal with these particular environmental conditions) are placed in the Match Set ([M] on the diagram). An action is then selected and sent out to be carried out on the environment. The reward for the action is sent back to the system in order to remember how efficient the action is in its domain.

The algorithm for a Learning Classifier System is therefore very different from the non-learning algorithm. An iteration of a learning classifier system takes place in the following way. It starts with an initial time, an initially empty message list and a randomly generated population of classifiers. It then increases the time counter for every iteration of the following:

1. Detectors check whether input messages are present
2. If the message list is non-empty, compare the messages in the message list to the classifiers, and save the matches
3. Highest bidding classifier(s) collected in the message list are allowed to post their message(s)
4. Reduce the strength of the bidding classifiers
5. Effectors check the new message list for output messages
6. Receive payoff from environment (‘reinforcement’)
7. Distribute payoff/credit to classifiers
8. Eventually (depending on time), an evolutionary algorithm, usually a genetic algorithm, is applied to the classifier population

### 4.2 Limits of LCS

The typical LCS is adapted to learn from experience, and so the development of such systems has allowed progress in many areas (Lanzi and Riolo, 1999), such as autonomous robotics, traffic signal control, medical data analysis, sequence prediction and supervised classification, etc.
However certain intrinsic properties have negative effects on the learning classifier system’s capability to achieve optimal solutions.

- **Tax**: strength of a classifier improves only if it is being used. It is possible that a classifier with efficient actions would not be used very often, simply because it does not cover a very visited area of the environment space. In consequence it would never receive a reward. Because an LCS typically taxes those classifiers which do not receive an award each time an award is accorded, this classifier’s strength value would diminish at every trial and eventually be null: the classifier would be deleted even though it could be very effective in its area.

- **Stability**: Goldberg’s Simple Classifier System, devised in 1989, tends to encourage the development of strength in the single classifier used in a trial, rather than across a number of classifiers proposing the action carried out on the environment. Therefore a deletion operation could delete a classifier although it does propose efficient actions. There is also the situation where one classifier covers areas A and B, and another classifier covers areas B and C. If the first classifier is more efficient than the second on area B, the second classifier could be deleted despite its efficiency over area C, and the information on area C would be lost forever.

These properties of the typical LCS mean that classifiers compete against each other, when really they should be working together to reach the optimal solution over the entire problem space.

The last issue validating the need for an improved version of LCS is the use of individual classifiers’ strength values. A Learning Classifier System consists of many classifiers, each one carrying its own strength record, a representation of the classifier’s value to the system.

It uses the strength value in two ways:
• For selecting an action to try
• For reproducing an action using the genetic algorithm

This implies that the same value represents both the prediction payoff within a classifier and that classifier’s reproductive value. But in this configuration, a classifier which is very good at predicting a lower payoff can end up having the same strength as one which inaccurately predicts higher payoffs.

Wilson invented a system which differed from the classical LCS in that it did not hold all its information in one strength value: he separated the two issues by introducing new values within each classifier:

• Prediction, used for action selection
• Accuracy and Prediction Error, maintained to calculate Fitness
• Fitness, used for selection by the genetic algorithm

These values are defined by Barry [7]. The prediction corresponds to the reward level which is to be expected if the action proposed by the classifier is selected when its condition is matched. Accuracy is a calculated measure of the likelihood of the classifier obtaining the reward identified by the prediction, when invoked. The prediction error value represents the absolute difference between the prediction of the classifier and the reward or payoff received over a period of time. Fitness of a classifier can then be calculated as the accuracy of the classifier relative to that of other classifiers that propose the same action in the conditions in which the classifier fires. Fitness is a measure of adaptation to the specific environment.

The advantage of the X-Classifier System, as exposed by Tim Kovacs in his XCS optimality hypothesis (Kovacs, 1997) is that the XCS will tend towards populations which are complete, accurate and minimal. Because different properties of classifiers are identified by different variables, these can be used separately when appropriate. Instead of taking everything into account in a single variable and using it in both the population maintenance and the action selection phases, the genetic algorithm can select classifiers for reproduction and suppression according to their fitness, and it can also choose actions to be
carried out in the environment according to the corresponding classifiers’ prediction value.

Separating fitness from strength is only one of the changes Wilson and Butz introduced to Goldberg’s learning classifier system, to create their XCS. They also tried to modify the way classifiers were rewarded. In a classical LCS, a single classifier receives a reward: that whose action was used to get to the reward. However it figures that more classifiers should be sharing the reward: those which provide the same action, and therefore could just as easily have been chosen in the last step to the reward, and also all the classifiers used along the way before the final step of reaching the reward. Finally the genetic algorithm is applied over the entire population of classifiers, when it could be wiser to apply it only to those classifiers proposing the selected action.

4.3 Concluding Towards XCS

An XCS represents an improvement on a classical LCS in the way it maintains its variables to measure the fitness of its classifiers, in order to reach an optimal output, and in the way the genetic algorithm is used on the population.

In XCS, the values of a classifier’s fitness and prediction are separated, so that fitness is based on the accuracy of a classifier’s payoff prediction, rather than on the prediction itself. Fitness accuracy is also measured relatively to that of the other classifiers, and the genetic algorithm is used on a selected sub-population of classifiers, not on the entire population. Dynamic niching is the new strategy implemented in XCS, which encourages the convergence of the system towards an optimally general accurate population of classifiers. These criteria lead to stability and a population of classifiers which efficiently cover the environment problem space.
The following section describes the Butz-Wilson model of the XCS and its operation, explaining how it maintains its various variables to achieve optimal fitness of classifiers, and develops the idea of dynamic niching.
Chapter 5: The Butz-Wilson X-Classifier System

5.1 The XCS Model

Wilson’s XCS Model is represented as the following (adapted from Wilson, 1995):

From this diagram it is clear that the XCS applies the genetic algorithm in two areas: for maintaining the classifier population, and for selecting the action. The model also shows that XCS differs from a classical LCS in that it does not include
the conventional Message List. With the XCS, the environment sends out a single input message and receives in return a single output message. The operation of the XCS during an iteration is described in the next section.

### 5.2 How the XCS Operates

The XCS is run in iterations. According to Barry (Barry, 2000), each iteration operating as follows:

1. Reset to initial values.
2. Choose the mode (Explore or Exploit – see section 5.2.1 Iteration Mode), depending on the strategy being used.
3. Obtain a new detector message from the input interface to the environment.
4. Create the Match Set (see section 5.2.2 Match Set Creation).
5. If the match set does not contain a single classifier, create one which matches the message and re-apply step 4 to match the generated classifier.
6. Use the prediction and fitness of each classifier of the match set to calculate the system prediction for each of the classifiers’ actions; place these in the prediction array (see section 5.2.3 Prediction Array Creation).
7. If in Explore Mode, select a random action $e$ from those in the prediction array. If in Exploit Mode, select the action $e$ with the highest prediction in the prediction array. Then all the classifiers with action $e$ represent the Action Set. (see section 5.2.4 Action Selection)
8. Send message $e$ to the environment decoders for environment action.
9. Examine the environment for a reward.
10. Update the values of all the relevant classifiers (using Credit Allocation algorithms – see section 5.2.5 Reward Reception).
11. If in Exploit mode, invoke the reporting operations. If in Explore mode, test the triggers to determine whether to apply each of the remaining induction algorithms in order to generate new candidate classifiers.
5.2.1 Iteration Mode

Unlike a classical LCS, which only explores the environment and modifies a classifier at each run, the XCS works in two different modes: Explore and Exploit. The Explore mode, similarly to other learning classifier systems, visits new parts of the input space to expand its learnt experience: the system does not select the optimal action, in order to find out the value of alternative actions. The XCS Exploit mode uses the system’s existing knowledge of the optimal action within previously visited parts of the input space: it selects the optimal action, in order to reinforce the optimal classifiers or demonstrate the optimal learnt policy. Exploration is necessary in order to learn anything. But exploitation, which picks the highest-prediction action, is necessary in order to make the best use of what is learned.

A trial is a set of iterations of the LCS that conclude with the receipt of a reward. As an environment can be either single-step or multiple-step, a trial can consist of a single iteration or many iterations of the main cycle of the system operation. A trial of the XCS starts by determining whether the episode will be run as an Exploration or an Exploitation trial. Traditional ‘Reinforcement Learning’ methods tend to select the mode on each step even in multiple-step environments. In 1995, Wilson proposed a strategy which decides between Explore or Exploit by a random selection at the start of each trial, thus keeping the XCS within either Explore or Exploit mode for the duration of a trial. In 1996 Kovacs used a technique which consists in alternating deterministically between explore and exploit mode, thus ensuring that the reports, produced on each Exploit trial, are produced with equal regularity. Wilson described many other Explore-Exploit strategies in 1997.

5.2.2 Match Set Creation

The environment delivers a new message. The matching process compares it to all the classifiers for the given population. The message is said to match a
classifier if and only if it fits in perfectly with the classifier’s condition, usually a combination of ‘0’, ‘1’ or ‘#’ (wildcard) bits.

The classifiers whose condition matches the input message are recorded into a set, called the ‘Match Set’. This is initially set to the empty map before the matching process begins. Then as new actions are proposed by matching classifiers, each of these matching classifiers is added to the appropriate match record and into the match set.

If the match set is empty at the end of the matching process (that is, if no classifier has a condition matching the input message), a new classifier is created, whose condition does match the input.

### 5.2.3 Prediction Array Creation and Action Selection

The ‘Prediction Array’, emptied at each iteration, corresponds to the set of actions which can be performed by any of the classifiers in the match set, each action having a calculated system prediction value.

The system prediction value of an action depends partly on the fitness values of the classifiers proposing it: ‘fit’ classifiers contribute more to the prediction of their proposed actions.

The action to be carried out is chosen from the prediction array differently in the Explore and Exploit modes. In the Explore mode, it is selected at random, whereas in the Exploit mode, the action used is that with the highest system prediction value. (If many actions share the highest value, then one of these is selected at random.)

The ‘Action Set’ consists of all the classifiers of the match set which propose the chosen action (and have therefore contributed to its system prediction).
5.2.4 Reward Reception

The selected action is then sent to the environment interface for decoding, and is performed in the environment. If the action leads to a reward state, this reward is recorded to update the corresponding classifiers’ strength values appropriately.

The XCS learns by keeping track of all the different values, and updating them when a reward is reached. As Barry and Saxon pointed out (Barry and Saxon, 2000), learning should be allowed to take place whether in Explore or Exploit mode.

At the end of each explore mode trial, the rest of the triggered induction operators are given an opportunity to generate new rules before the start of the next iteration of the XCS.

This specific operation of the XCS solves several problems encountered in classical learning classifier systems. The next sections of this chapter describe how XCS deals with the fitness calculation aspect of classifier systems, how the genetic algorithm is used and the incorporation of niching into the XCS.

5.3 Classifier Fitness in the XCS

The aim of any learning classifier system is to concentrate increasingly upon a single locally optimal solution within its population. This is known as convergence. The XCS proves more efficient than other learning classifier systems at converging towards the optimal solution.

In the XCS, fitness corresponds to the accuracy of the classifier, relative to that of others in the action set. (Note that one classifier could occur in many match sets, and therefore in many action sets: evaluating a classifier’s fitness value involves making a timely estimate, according to the Widroff-Hoff technique.)
As it is calculated relatively to other classifiers in the action set, a classifier’s fitness will be lower if it has had to compete with more effective classifiers. In the context of one action set, the introduction of one very accurate classifier triggers the reduction of the existing classifiers’ fitness values. The new classifier has a high relative fitness value and is likely to be selected in the future by a genetic algorithm, and be duplicated. Here again, a high fitness classifier being duplicated decreases the fitness values of the other classifiers of the action set.

This mechanism implies an easier capture of the highest fitness classifier of an action set. Ideally in the end, each action set will be dominated in number, and therefore in chances of selection, by its fittest, and therefore most accurate, classifier.

However taking the problem from another angle, the selection of classifiers in separate action sets can be compared to the election of party members in separate constituencies: the elected party is the one with the most constituencies, even if it does not necessarily have a majority of all voters. Similarly, in the XCS, the fittest of all classifiers may not end up dominating the system in certain situations, because of different action set sizes.

Suppose the following preconditions are present in the XCS:

- Classifiers C1 and C2 propose action A, and classifier C3 proposes action B.
- Many other classifiers propose action A, but very few propose action B.
- In reality, C1 is more accurate than C2, and C2 is more accurate than C3.
- In the action set corresponding to action A, C1 is the most accurate of all classifiers.
- In the relatively small action set corresponding to action B, C3 is the most accurate of all classifiers.

As a result of the mechanism described above, after a sufficient number of trials, classifiers C1 and C3 will stand out as being the fittest classifiers in their respective action sets, and therefore have high fitness values. On the other hand,
classifier C2 will have a low fitness value, lower than C3’s, despite C2 being more accurate than C3.

(This example is only described as part of the explanation. If it only contains classifiers with a single action, this is purely for reasons of simplicity. In reality a classifier may propose many actions, and therefore be a part of many different action sets.)

The most accurate classifiers can thus end up being the most specific classifiers. Therefore the calculation of fitness needs to take into account the occurrence of a classifier linked to its generality, that is the proportion of condition positions at which are wildcard values. This puts pressure on the introduction of large action sets.

### 5.4 Niching

One of the problems found in LCS is that the classifiers compete with each other, thus the final population can be full of very good classifiers in one area only, and terrible at solving other domains of the problem at hand. Niching encourages the classifier populations to cooperate and work together to solve a problem. This mechanism is another inspired from nature, in which the natural formation of distinct species exploiting different resources, corresponds to niches of the computer environment: the creation of subpopulations in a fixed size population, with each subpopulation specializing at a particular domain of the problem.

Before niching, the population individuals competed against each other, the fittest taking over very fast but not necessarily being very useful as a whole. Two countries, one with all the all the natural resources, and the other with all the technology, will not survive very happily without cooperation, as neither covers the whole range of professional activities. In the same way, niching allows the classifier system to converge to a population of diverse species which, put together, cover the environmental space efficiently. It means that all species of classifiers, originally competing with each other, decide to share resources and
competencies. This can be compared to the beginnings of trade, when the early farmers in Mesopotamia stopped producing a variety of food for themselves alone and decided to specialize and trade with each other.

The following diagram points out that the optimal population of classifiers put together will not only be very good in domain A or domain B of the problem space, but will cover all areas.

Thus, niching is a very important addition to classical learning classifier systems. The XCS will not simply try to achieve the fittest population (in terms of task fitness) but will also have a generalised population covering a wider range of the problem space.

This chapter has described the novelties introduced in the XCS and explained their purpose in converging to an optimal solution. Chapters 6 and 7 describe two types of environments which an XCS could be used in to optimise parameters: the maze environment and the finite state world environment. The following chapter 8 introduces the main problems still left in XCS, before going on to the project experiments using XCS in chapters 9, 10 and 11.
Chapter 6: Maze Environment

6.1 Definition

Consider a computer simulation of some animal A evolving in some life environment E, which would typically contain basic food (which the animal does not need to chase) or prey, and predators (whose own goal could eventually be to chase animal A). To develop such a simulation requires knowledge of machine learning tools and techniques, as well as that of an animal’s behaviour and how its brain works in real-life.

The animal is able to react to the environment it lives in, in its own specific way. It can ‘see’ things such as food, animals or light, and has a set S of basic actions which it is able to perform, such as “move towards some direction”, “eat”, “sleep”, etc. A performs these actions according to hormones or chemicals present in its brain and neural system. The program is in charge of ‘injecting’ chemicals into the animal’s system as they could appear in the animal’s lifecycle, thus triggering some specific animal behaviour.

The animal is actually an agent in an agent-system and will therefore be referred to as an animat. This is a simulated animal within a constrained environment, consisting of a limited set of environmental detectors and a limited set of effectors.

Any simulation case consists of some sets, for instance:
- A set O of objects which can appear in the environment,
- A set P of perception methods available to A,
- A set S of actions which can be performed by A,
- A set C of chemicals which can be present in A’s neural system,
- A set R of assumptions, or environmental rules.
6.2 Example of a Maze Environment

Consider the simplistic case described here. The environment can be represented as a grid; each square can either be empty or contain some environment object (or an animat A).

\[ O = \{ "predator", "food" \} \]
\[ P = \{ \text{lookAhead()} – \text{returns null or the environment object present in the square right in front of A} \} \]
\[ S = \{ \text{moveForward(distance), turn(degrees), eat()} \} \]
\[ C = \{ "scared" \} \]
\[ R = \{ \]
  - A can step into any square adjacent to its present position square provided this new square is not occupied by a predator.
  - A can perceive anything in a radius of 5 squares, when looking in that direction.
  - A predator object can step into any square adjacent to its present position
  - A predator will only pursue prey if it is within a radius of 5 squares, and if the predator is still hungry.
  - Etc. (the list can grow rapidly with the programmer’s imagination)
\}

6.3 Issues in a Maze Environment

Thus the implementation would consist in placing the animat in a random square of the grid E and allowing it to evolve. The animat would then have to experiment various actions in various cases, remember the reward in each one and later apply this new knowledge gained when choosing an action to perform. Emotional variables can release a ‘chemical’ in the animat’s ‘brain’. Rules are established for each variable (for example “hungry”, “tired”, “scared”) to give the animat priorities: if “hungry” then eat() is a priority, if “scared” then eat() and sleep() will have to be disallowed for some time, etc. The program needs to learn which actions to allow or inhibit in which conditions, by experimenting on different actions in different circumstances.
Obviously when experimenting with very little information about a certain situation and the appropriate actions to carry out, there is the possibility that the animat will die. The knowledge acquired from the experience can be stored, but there is need for other instances of the animat to carry out further experimentation until the best rules of conduct for the animat in the environment emerge from the big group of possibilities.

In fact the ‘population’ can consist of many similar animats, living together in the same environment, each experimenting and giving the knowledge acquired to a common database, and each being able to use this information in further situations.

The environment is dynamic, as changes of different natures can take place (some suggestions are: day time/night time, predator movement and food appearance, etc.). The environment also needs to be discreet, as it has three dimensions (x, y and time), but a question is how to optimally choose the default interval of time.

The aim is to learn when to release the chemical. The animat has an initial set of actions allowed, and the chemical helps to limit actions later (inhibit actions).

Scenarios can of course become more complex. However to what extent can the model environment be simplified, considering that it must be complicated enough to be realistic and to give the animats something to learn?

A few ideas involve adding:
- many more actions, such as “sleep” and “wake up”,
- different chemicals
- different effects of chemicals in different proportions

Such an environment can be easily implemented in java, but the main issue is how to implement the agent-like animats and to analyse the time data they use: can this be done with binary strings which can be easily processed by the genetic algorithm? Further work could involve a full implementation of the predator
agents as well, rather than viewing predators only as normal objects, part of the
dynamic environment.

Another interest is also to find out more about the chemicals: would it be
possible for the animat to learn quantities and durations of these chemicals so as
to achieve an optimal efficiency in the environment?

The time variable needs to be integrated into what is being learnt by the
program through experience. One way of incorporating it would be to use time
as a changeable variable in the experiments: experiment with various lengths of
time (that is, various amounts of chemicals which can last different amounts of
time, or various ‘re-injections’ of chemicals, etc.). Another solution could be to
incorporate time into the reward system. When an action is tested in certain
conditions, take the necessary time for the action to be performed successfully
into account in the reward.

There is room for discussion as to which are the best strategies for teaching
animats appropriately. How do we want to take time into account in the reward?
Do we have the same basis for rewards when the animat encounters different
environment objects during the experiment? What unit of time do we use? How
often do we ‘breed’ new animals, that is, use the genetic algorithm on the
population to create new ‘children’ instances? For how long does a chemical
inhibit an action? In what quantities should we ‘inject chemicals’? How long after
chemical injection does each specific effect start? How many different types and
quantities of chemicals are we experimenting with? All these questions would be
answered in order to create a maze environment in an XCS system which uses
simulated chemicals to simulate the animat behaviour.

Mazes are not the only type of environment: many other kinds exist. The next
chapter describes a finite state world environment, in which animats do not
evolve randomly in a 2-dimensional digital simulated space, but in a world of
states from start to finish.
Chapter 7: Finite State World Environment

7.1 Definition

A finite state world (or FSW) is a type of environment where one evolves differently from in a maze, that is, not in 2 dimensions going from one square to another in a grid, but in a chain of linked states, going from one state to another. The environment now consists of a finite number of states, connected as a graph by labelled transitions. Each state is identified to the system by a pre-defined value, and each action is interpreted by the finite state world to a label that corresponds to a transition. A subset of states will be identified as start states from any of which a trial can begin, and another non-overlapping subset of states will be identified as reward states in which a trial will finish and a reward will become available to the system.

When in one of the non-end states, there is a choice of possible actions to carry out. Each of these actions will lead to another state. A trial consists of moving from a start state to a terminal state. On completion, once an end state is reached, the current position is set back to a start state in preparation for the next trial.

Each state has the following properties:

- a name, for example a number from 0 to n-1, n being the total number of states
- a type, a choice of start state, end state, and normal state (normal state being neither start nor end state)
- a list of actions with a corresponding following state for each action (if the state is not an end state)

An end state has an extra value called the Reward value, defined by Barry [7] as “A numerical representation of the value to the Animat of a detectable state change in the environment as a result of an action”. By definition this end state does not carry legal transitions to any other states.
7.2 Example of a Straightforward FSW

Take the following set of states as an example, where each state offers a choice of two actions.

<table>
<thead>
<tr>
<th>State Name</th>
<th>Action 1</th>
<th>Action 2</th>
<th>State Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>N</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>N</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>R 100</td>
</tr>
</tbody>
</table>

Where:

$S = \text{start state}$

$N = \text{normal (neither start nor end) state}$

$R = \text{reward (or end) state}$.

The extra value for the reward state is the reward value.

Depending on the outcome of running the genetic algorithm, many paths could be taken, each with a certain time delay before reaching the reward. The XCS should experiment on the various paths, and learn to take the shortest path. In this example the obvious path to be learnt is:

- State 0: take action 2, get to state 2
- State 2: take action 3, get to state 4: Reward of 100.

An obvious mistake would be to keep going from state 0 to state 1 back to state 0, without learning that this is an endless loop.
This is a very simple example, but many complications can be introduced. There may be several start states and/or end states, each end state carrying a different reward value. The number of states can be much more important, as can be the number of possible actions per state. Also, the number of actions per state may vary depending on the state. Finally a state could have an action which brings it back to itself (the action therefore being ineffective, or null).

### 7.3 Example of a More Complicated FSW

This example shows how complicated the paths can become, even when the world still only consists of a small number of states.

Suppose the states have actions as follows:

<table>
<thead>
<tr>
<th>State Name</th>
<th>Action 1</th>
<th>Action 2</th>
<th>Action 3</th>
<th>Action 4</th>
<th>State Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>R 70</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>N</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R 100</td>
</tr>
</tbody>
</table>
7.4 Example of a Common Simple FSW

In general however, finite state worlds look more like this (from Barry, 2000):

<table>
<thead>
<tr>
<th>State Name</th>
<th>Action 0</th>
<th>State Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S1</td>
<td>S</td>
</tr>
<tr>
<td>S1</td>
<td>S2</td>
<td>N</td>
</tr>
<tr>
<td>S2</td>
<td>S3</td>
<td>N</td>
</tr>
<tr>
<td>S3</td>
<td></td>
<td>R 1000</td>
</tr>
</tbody>
</table>

Here the path must go through each state, but has little ambiguity as only one action is possible from each non-end state: the optimal path is the only path possible from the start state to the end state.

7.5 Example of a Corridor FSW

In particular there exists a less straightforward finite state world model called the ‘corridor’ (from Barry, 2000):
<table>
<thead>
<tr>
<th>State Name</th>
<th>Action 0</th>
<th>Action 1</th>
<th>State Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S1</td>
<td>S5</td>
<td>S</td>
</tr>
<tr>
<td>S1</td>
<td>S6</td>
<td>S2</td>
<td>N</td>
</tr>
<tr>
<td>S2</td>
<td>S3</td>
<td>S7</td>
<td>N</td>
</tr>
<tr>
<td>S3</td>
<td>S8</td>
<td>S4</td>
<td>N</td>
</tr>
<tr>
<td>S4</td>
<td>S10</td>
<td>S9</td>
<td>N</td>
</tr>
<tr>
<td>S5</td>
<td>S1</td>
<td>S1</td>
<td>N</td>
</tr>
<tr>
<td>S6</td>
<td>S2</td>
<td>S2</td>
<td>N</td>
</tr>
<tr>
<td>S7</td>
<td>S3</td>
<td>S3</td>
<td>N</td>
</tr>
<tr>
<td>S8</td>
<td>S4</td>
<td>S4</td>
<td>N</td>
</tr>
<tr>
<td>S9</td>
<td>S5</td>
<td>S5</td>
<td>N</td>
</tr>
<tr>
<td>S10</td>
<td></td>
<td></td>
<td>R 1000</td>
</tr>
</tbody>
</table>

The optimal route in this environment would naturally be:

- State 0: take action 0, get to state 1
- State 1: take action 1, get to state 2
- State 2: take action 0, get to state 3
- State 3: take action 1, get to state 4
- State 4: take action 0, get to state 10: Reward of 1000.

This environment is more ambiguous and therefore contains sub-optimal routes. These routes, although they take a detour, never go back in progress towards the end state, and always rejoin the optimal path. There are also more states than in the previous examples, but the number of legal actions per state remains small, thus limiting exploration complexity. The number of states can also be increased by adding more increments, and the ability of the system to decide the optimal route can be determined in function of the number of states.

The layout of the states in the world can sometimes lead to problems for manipulation by a Genetic Algorithm, such as the Aliasing Problem, and in
particular, the Consecutive State Problem. These are described in the following chapter 8.
Chapter 8: Aliasing Within the XCS

The aliasing problem within classifier systems was first identified by Lanzi (Lanzi, 1997). It arises when the environment provides the same message for two states which generate different constant payoffs. The payoff is a value distributed to classifiers acting in the previous iteration from those active in the current iteration by the Credit Allocation mechanism. If two states have different payoff values but correspond to the same input message, the payoff prediction for that message cannot be determined correctly. In particular it occurs when applying Wilson’s XCS to certain simple multi-step environments which are non-Markovian, that is, which contain an environmental regularity. The term ‘Environmental Regularity’ is defined as “An area of the environment within which the states will produce the same messages and respond in the same manner to a given message as states within one or more other areas of the environment” (Barry, 2000). Aliasing states have a regularity and lead to different payoffs: they cannot be accurately represented (Lanzi, 1997). There exist a very few possible solutions so far. One is Lanzi’s very complex memory mechanism which aims to disambiguate between states. To understand the problem it is helpful to use the following example.

8.1 Example of the Aliasing Problem

The aliasing problem can be easily explained using a Woods maze environment. Each position in the maze has an input value encoding the contents of the 8 squares surrounding it.

Woods Environment: ‘0’ = Rock, ‘.’ = Space, ‘F’ = Food

```
0 0 0 0 0 0 0 0
0 . . . . . F
0 0 0 0 0 0
```
The two positions in the centre of the grid have exactly the same input vector V: 3 rocks in the 3 north positions, 3 rocks in the 3 south positions, an empty space on each side. However these two positions have a different payoff: the position on the right (square A in diagram below) is closer to the food square (square F) than the position on the left (square B in diagram below), and therefore carries a higher payoff.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>.</td>
<td>A</td>
<td>B</td>
<td>.</td>
<td>F</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The payoff function should give one payoff for each individual input vector, however it is impossible to give a unique payoff to the input vector V. Therefore it is impossible to give a payoff value to all classifiers of the match set which propose the input vector V. In a longer maze this problem would concern many more states, and the system’s overall accuracy would be greatly damaged.

### 8.2 The Consecutive State Problem

The Consecutive State Problem is a distinct sub-class of the Aliasing Problem, which can be addressed by a much simpler mechanism than Lanzi’s. It occurs when consecutive states of the environment provide the same message, thus affecting the learning of the state action payoff mapping within the XCS.

In his work to solve the aliasing problem on separate states, Lanzi investigated the effects of aliasing on the classifiers covering the aliased states. However when a classifier’s values are updated, the values of all preceding classifiers are also updated, in order that they may take into account the fact that they contributed partly in achieving a reward. Thus aliasing on one state would have consequences on preceding states as well, which Lanzi failed to address. Barry showed that “where the states were visited irregularly, [...] classifiers covering the preceding state were made inaccurate” (Barry, 1999). Investigating this he found that the inaccurate classifiers covering the aliased states were not however removed by the genetic algorithm for their inaccuracy, but were allowed to
remain in the population for their (over-)generality, which meant they would take over earlier states, which could be much more accurate, thus permitting a potential dominance of over-general inaccurate classifiers in the population.
Chapter 9: Experimenting with the XCS

9.1 Description of the Java XCS

9.1.1 Class Diagram

The X-Classifier System used was that devised by Butz and Wilson (Butz and Wilson, 2000 and Butz, 2000). This XCS works with the following class diagram:

```
XCSConstants

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>XCS</td>
</tr>
<tr>
<td>------------------------------</td>
</tr>
<tr>
<td>PredictionArray</td>
</tr>
<tr>
<td>------------------------------</td>
</tr>
<tr>
<td>XClassifierSet</td>
</tr>
<tr>
<td>XClassifier</td>
</tr>
<tr>
<td>------------------------------</td>
</tr>
<tr>
<td>Environment</td>
</tr>
<tr>
<td>------------------------------</td>
</tr>
<tr>
<td>MazeEnvironment</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>MPEnvironment</td>
</tr>
</tbody>
</table>
```

Because of the design, the XCS requires a separate XCSConstants class, which contains all the necessary parameters and flags for running the XCS. The XCS
naturally has a population of classifiers, match sets and action sets, dealt with by the XClassifierSet class. These are made up of instances of XClassifier. The PredictionArray class links the XCS to the individual classifiers: this class contains all methods for dealing with the prediction array creation and the action selection. Finally the XCS contains an environment, which can be of any of the types present. The Butz-Wilson XCS Java implementation comes with two sub-Environment classes: a maze environment and a multiplexer problem environment. Part of the project work was to implement a third environment class, called FSWEnvironment, which allows XCS to be used in a finite state world.

9.1.2 What Happens at Runtime:

The XCS main method is called when running the program. This method reads in the arguments (see section 9.2.1 FSW Arguments, for the arguments required in finite state world environments) and creates the environment, depending on the type of environment chosen by the user, by calling the corresponding constructor method (in particular, the FSW constructor is described in section 9.2.2 FSW Constructor). It then calls its runXCS method.

The runXCS method of the XCS class creates the file writer objects necessary for printing the results to text files, then starts the experiments by calling its startExperiments method.

The startExperiments method goes into a for-loop: for all experiments to be carried out, it initialises the population, and depending on the type of experiments to be carried out (single step or multiple step) it calls one of these two methods: doOneSingleStepExperiment or doOneMultiStepExperiment. In the case of a finite state world, the type of experiments to be carried out is always multiple-step, as it can take more than 1 step to get from a start state to an end state, so the only method to be considered by the finite state world is doOneMultiStepExperiment.
The `doOneMultiStepExperiment` method chooses the trial mode (every second trial is in Explore mode, every other trial is in Exploit mode), and calls the corresponding method: `doOneMultiStepProblemExplore` or `doOneMultiStepProblemExploit`.

Each of these two methods goes into a for-loop: for all steps until a reward is received, they create the XClassifierSet and PredictionArray instances. The action set is then created if in Explore mode, and an action is selected (a random action in the case of an Explore mode trial, or the best action when in Exploit mode). The reward is then returned by calling the `executeAction` method of the Environment class, described below. Because it is a multiple step problem, the action set of the previous step has to be updated with the information received by applying the current action.

The `executeAction` method is rewritten in each specific environment class. In particular this has been done for the FSWEnvironment class (see section 9.2.3 FSW methods). In Explore mode, the action set is updated and the genetic algorithm is run on its classifiers. In Exploit mode, the system error value is updated.

9.2 What the Work Involved

The implementation part of the project involved doing the following:

- Changing the environment (Environment.java) interface so that it includes a finite state world environment (as well as maze and multiplexer problem environments): different parameters and arguments were needed when running the XCS main method
- Creating a new class file called FSWEnvironment.java for running XCS in a finite state world environment
- Testing the XCS using different arguments for a finite state world environment
9.2.1 FSW Arguments

Running a finite state world required the following arguments:

<table>
<thead>
<tr>
<th>Index in args[] Array</th>
<th>Corresponding Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Output text file, for example Output.txt</td>
</tr>
<tr>
<td>1</td>
<td>FSW input file, for example FSW\Test1.txt</td>
</tr>
<tr>
<td>2</td>
<td>Nº of bits (optional)</td>
</tr>
<tr>
<td>3</td>
<td>Nº of trials (optional)</td>
</tr>
<tr>
<td>4</td>
<td>Nº of experiments (optional)</td>
</tr>
</tbody>
</table>

The consequent changes were made to the main method of the XCS class (see Appendix A.1).

The FSWEnvironment class includes a constructor, which reads in the finite state world input file, and all methods for evolving in the finite state world and creating the necessary output.

9.2.2 FSW Constructor

The constructor reads the input file in order to create a states vector which characterises the finite state world. If the input world consists of n states, then the vector has n elements, each one being a state array with the information on the corresponding state:

<table>
<thead>
<tr>
<th>Index</th>
<th>Corresponding Value, type and range</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>State name (integer from 0 to n-1, n being the total number of states)</td>
</tr>
<tr>
<td>1</td>
<td>State reached by applying action 0 (integer from 0 to n-1)</td>
</tr>
<tr>
<td>2</td>
<td>State reached by applying action 1 (integer from 0 to n-1)</td>
</tr>
<tr>
<td>3</td>
<td>State type (integer from 0 to 2: 0=start, 1=normal, 2=end state)</td>
</tr>
<tr>
<td>4</td>
<td>Reward value (integer)</td>
</tr>
</tbody>
</table>
The constructor needs two arguments: a String ‘inFileString’ and an integer ‘conditionLength’. Its pseudo code is:

*Get the number of states from the first line.*
*Get the number of actions per state from the second line. (In the implemented version, there can only be 2 actions per state.)*

while (not done):
  • Create string from each line of the input file: each line corresponds to one state
  • If 1st line, get the number of states
  • For all the following lines of the input file, fill the states vector with state information:
    o Create integer array for each state
    o Read through each character in the line
    o Enter state name: index 0 in array
    o Enter first action state: index 1 in array
    o Enter second action state: index 2 in array
    o Enter state type: index 3 in array
    o If state type is 0 (start state), add to start states vector
    o If state type is 2 (end state), enter reward
    o Else reward is 0
    o Add array to states vector
  • Reset the current state to a start state

### 9.2.3 FSW Methods

The methods of the FSWEnvironment deal with all functions allowing the program to evolve from the finite state world’s start state to its end state, where it receives an award.

**executeAction** executes one action in the environment.

- Arguments: integer action
- Returns: double (corresponds to the reward)
This method has the following pseudo-code for a finite state world:

- Check that the action is legal, that is either 0 or 1;
- In the current state’s information array, find the state corresponding to the action: this becomes the new current state
- In the new current state’s information array, check the state’s type. If the state is an end state, return the reward amount. Otherwise return 0.

**ToBinary** converts the integer to the binary string corresponding to the situation.

- Arguments: integer value, int bits
- Returns: String

**nextState** executes the specified current state in the environment and returns possible payoff

- Arguments: integer current
- Returns: void

**resetState** Resets the current state to a random start state

- Arguments: none
- Returns: String

**getCurrentState** returns the current situation. The situation is the binary string.

- Arguments: none
- Returns: String

**wasCorrect** returns if this action was a good/correct action. This function is essentially necessary in single-step (classification) problems in order to evaluate the performance, however in a finite state world there is no correct or wrong action, so this method always returns false.

- Arguments: none
- Returns: Boolean (always “false”)
**doReset** returns if the agent has reached the end of a problem. In a finite state world environment, this function should return true when a reward has been achieved (when an end state has been reached).

Arguments: none
Returns: Boolean ("reset" flag)

**getConditionLength** returns the length of the coded situations.

Arguments: none
Returns: integer

**getMaxPayoff** returns the maximal payoff receivable in an environment.

Arguments: none
Returns: integer

**isMultiStepProblem** returns true if the problem is a multi-step problem. A finite state world environment only uses multi-step problems, so this is always set to true.

Arguments: none
Returns: Boolean (always "true")

**getNrActions** returns the number of possible actions in the environment.

Arguments: none
Returns: integer

### 9.3 Outputs

The Butz-Wilson XCS does not provide an interface for testing: it does not send outputs to any kind of graphical tool, but simply to the specified text file (in the second argument when running the program). Testing the XCS involved doing the following for each run:

- Run the main method of the XCS (see Appendix A.1)
• Get the results in a csv (comma separated format) file: “outfile.csv” (given as argument when running), and the population evolution in “popfile.csv”
• Run a perl program to average the results over all the experiments (see Appendix B)
• Use a graphical tool (Gnuplot) to plot the averages

The results output file, created by the PrintWriter object pW of the XCS object, gives the following data, in comma separated format:
- trial number
- performance: number of steps to reach end state
- error
- population size

The population file, created by the PrintWriter object popW of the XCS object, gives the following data, in comma separated format:
- Classifier Condition String
- Action
- Prediction
- Error
- Fitness
- Numerosity
- Exp
- Action Set Size
- Birth (age)

Examples of the results output file, the population file and the graph produced using Gnuplot, appear on the accompanying CD, in the 'Output Examples' folder.
10.1 Test 1: Simple Finite State World

10.1.1 Test 1 Experimental Hypothesis

The goal of the project was to investigate the effectiveness of XCS when facing problems such as the consecutive state problem, described in chapter 8. To this end a java version of the Butz-Wilson X-Classifier System was implemented, which would deal with finite state world environments. The first test to be carried out using the implementation was simply to check that the new version could learn simple straightforward problems. The finite state world tested on for Test 1 was similar to an example described in chapter 7. This input world is the following:

The fsw file used figures in Appendix D.1. The shortest path through the world is obviously:

- State 0: take action 2, get to state 2
- State 2: take action 3, get to state 4: Reward of 100.

<table>
<thead>
<tr>
<th>State Name</th>
<th>Action 0</th>
<th>Action 1</th>
<th>State Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>N</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>N</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>R 1000</td>
</tr>
</tbody>
</table>
The required results using the optimal gamma value were that the system would learn to reach the end state (state 4) in 2 steps, after a series of Explore and Exploit mode trials. The XCS was run with a gamma parameter of 0.71 and the following argument values:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>N° of bits</td>
<td>4</td>
</tr>
<tr>
<td>N° of trials</td>
<td>500</td>
</tr>
<tr>
<td>N° of experiments</td>
<td>10</td>
</tr>
</tbody>
</table>

### 10.1.2 Test 1 Results

The output file with experiment results figures on the accompanying CD in the 'Test 1 Results - Appendix B' folder as 'Test 1 Experiments'. The population file showing the evolution of the classifier population figures on the accompanying CD in the 'Test 1 Results - Appendix B' folder as 'Test 1 Population'. The corresponding average results graph is the following:

This shows an average number of steps needed to reach the end state equal to 2, which was the desired output. Although over a number of experiments, the
system does seem to be learning to take only two steps to reach the end state, it is interesting to see what happened at experiment 3. The performance table for experiment 3 appears in Appendix D.1. The steps graph is the following:

From this it is clear that the system has not learnt anything about the environment in this particular experiment. The explanation can be found by looking at the population table, in Appendix D.1, with the following graph:

In this experiment, the `#####` classifier must have somehow managed to subsume all other classifiers at one point, and now dominates the population, although it is no use at all for learning. Indeed, looking simply at the fitness of
each classifier remaining in the final population, we can tell that it was not even very ‘fit’, only very general, but ‘fit’ enough to subsume at some point:

![Population Fitness Graph]

This tends to mean that Butz’ subsumption algorithm could be improved, to avoid this kind of error.

It would be much more sure however to test the XCS in other ways:

- Test on the same finite state world but using different arguments
- Test on a different (but still Markovian) finite state world

### 10.2 Using Different Argument Values

The XCS was tested several times on the same straightforward finite state world environment, only with a different argument value for the number of trials. (It is important to keep the number of experiments at 10 simply to be able to compare results in different experiments. The number of bits is also to be kept at 4 as this is sufficient given the small number of states and actions.)
### 10.2.1 Test 2.1: 1000 Trials Per Experiment

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nº of bits</td>
<td>4</td>
</tr>
<tr>
<td>Nº of trials</td>
<td>1000</td>
</tr>
<tr>
<td>Nº of experiments</td>
<td>10</td>
</tr>
</tbody>
</table>

The output file with experiment results figures on the accompanying CD in the ‘Test 2.1 Results - Appendix C.1’ folder as ‘Test 2 1 Experiments’. The population file showing the evolution of the classifier population figures on the accompanying CD in the ‘Test 2.1 Results - Appendix C.1’ folder as ‘Test 2 1 Population’. The results for these experiments also show the same evolution during the first few trials, and the number of steps stays at 2 till the end.

![Average XCS Output](image)

Here the same phenomenon appears as previously, in experiment 6. The corresponding population file, in Appendix C.1, has the following chart:
Here again one of the dominating classifiers is ‘####’. The fitness of each classifier appears here:

For this experiment, the graphs shows the absence of learning yet again, and the importance attached to general classifiers. The consequences appear in the performance graph:
This experiment had a less dramatic result as Test 1 Experiment 3, as the ‘####’ classifier was not the only dominant individual of the population, but the problem still persists.

### 10.2.2 Test 2.2: 750 Trials Per Experiment

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nº of bits</td>
<td>4</td>
</tr>
<tr>
<td>Nº of trials</td>
<td>750</td>
</tr>
<tr>
<td>Nº of experiments</td>
<td>10</td>
</tr>
</tbody>
</table>

Here again, the overall graph shows that the program tends to learn the correct path fairly easily.
However another over-generalisation issue has occurred in experiment 5, where the population has the following numerosity and fitness values:

looking closer at the fitness graph, it seems the ‘#####’ classifiers were thought to be the ‘fittest’ by the system:
It is not surprising that the performance graph shows terrible learning results again:

Of course, each time only 1 or 2 experiments go wrong: this over-generality of dominant classifiers problem does not occur every time, which is why the graphs show excellent learning. However the following graphs show what should be happening with good results, for comparison’s sake.
None of the classifiers mentioned on these graphs have ‘###’, and this makes the performance graph so even:

### Got it right First Time and Remembered

#### 10.3 Using a Corridor Finite State World

### 10.3.1 Test 3 Experimental Hypothesis

Tests were then carried out on a typical corridor world, used as an example in chapter 7:
The optimal route in this environment is:
- State 0: take action 0, get to state 1
- State 1: take action 1, get to state 2
- State 2: take action 0, get to state 3
- State 3: take action 1, get to state 4
- State 4: take action 0, get to state 10
We would therefore expect the results to show that the system learns to average 5 steps in order to reach the end state. The arguments used here were the following:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nº of bits</td>
<td>4</td>
</tr>
<tr>
<td>Nº of trials</td>
<td>2000</td>
</tr>
<tr>
<td>Nº of experiments</td>
<td>10</td>
</tr>
</tbody>
</table>
10.3.2 Test 3 Results

The output file with experiment results figures on the accompanying CD in the 'Test 3 Results - Appendix D' folder as 'Test 3 Experiments'. The population file showing the evolution of the classifier population figures on the accompanying CD in the 'Test 3 Results - Appendix D' folder as ‘Test 3 Population’. The corresponding average results graph is the following:

This graph does not show very good results: even after 2000 trials, the system does not reach an average of 5 steps to the reward state. Here again, some experiments went well and others very poorly.

Experiment 6 learnt very well: the performance graph, together with the population fitness and numerosity graphs, show this. On the other hand, experiment 8 graphs show very poor learning. The graphs are shown together to emphasize the contrast.
On one side, the population seems very diverse, with many specific classifiers dominating. On the other, the ‘####’ classifier dominates the population and as a result the system is not learning. It is because of experiments such as experiment 8 (on the right) that the overall graph does not have an even performance line, and does not reach the optimal average number of steps. It is clear that if all experiments had been carried out as number 6, the classifier system would have learnt very fast and reached an optimal population of diverse classifiers.
11.1 Value of the Gamma Parameter

Each state receives a share of the reward received by the following state encountered. This share is calculated in function of a gamma parameter, so that the state before an end state receives $\gamma \times \text{reward}$, the state before that receives $\gamma^2 \times \text{reward}$, etc. It has been shown that an optimal value to use for this parameter is 0.71. The graph below shows the reward values per state using various values of gamma, in a straight line chain of states.

The higher the value of gamma, the smaller the difference between the values $\text{reward}(n) - \text{reward}(n-1)$ and $\text{reward}(n-k) - \text{reward}(n-k-1)$. Therefore the system is likely to overlook the error and make wrong assumptions as to the prediction of classifiers, throughout the range of states. However a low value of gamma leads to very important differences in the penultimate states, which is desirable, but the starting states will have very close prediction values, which encourages the system to view many states as one. We therefore perceive the ‘opposite’ effect of the aliasing problem. The system maintains a one-to-one function between state
and action, and a one-to-one function between action and prediction. The aliasing problem occurs when two distinct states have the same input; this makes the accurate choice of optimal action to be taken from that state impossible. With a very low value of gamma, two distinct states which are far from the end state have more or less the same prediction and therefore they can only be regarded as one state by the system.

All experiments were carried out using a gamma parameter equal to 0.71, to reduce any effects that this simple value could have, by encouraging inaccuracy, on the system’s learning.

### 11.2 Subsumption Algorithm

The previous tests were only carried out on Markovian environments, that is on finite state worlds which do not present an aliased situation. Yet even in these ‘simple’ cases some problems were encountered which hindered the system’s accurate learning. Barry exposed some issues of the Butz algorithm for subsumption which could explain these results (Barry, 2003).

The algorithm used by Butz in his X-Classifier System starts by finding the most general accurate classifier of the action set, to then compare it with all the others for accuracy:

```java
//select the most general accurate classifier in action set
bestClassifier = null;
bestGenerality = 0.0;
for (all classifiers c in the action set, from first to last) {
    if (c is accurate AND c.generality > bestGenerality)
        bestClassifier = c
}

//subsumption
for (all classifiers c in the action set, from first to last) {
    if (bestClassifier subsumes c) {
```
The main problem underlying this algorithm is that it promotes generality over accuracy. Because subsumption is checked against the most general classifier, it would be sufficient for the most general classifier (whose condition was made up only of wildcard symbols) to be accurate at one point, and it would subsume all others, making potentially accurate classifiers disappear and the system lose important information. What Barry suggested was to select a random classifier from the action set, rather than necessarily the most general, against which to compare the others. Generality and accuracy are both very important, but there is such a thing as “too much generality”. Indeed the appearance and multiplication of a wildcard-full conditioned classifier reduces the accuracy of the entire system and deletes information of the potentially useful subsumed classifiers. Barry’s algorithm would give more opportunity for other potentially slightly more accurate general classifiers to subsume, thus allowing the system to find its true prediction value:

```java
// select a random accurate classifier in action set
startPosition = random in action set;
accurateClassifier = null;
for (all classifiers c in action set, from startPosition then wrapping around action set randomly) {
    if (c is accurate) {
        accurateClassifier = c;
        exit;
    }
}
// subsumption
for (all classifiers c in action set, from first to last) {
    if (accurateClassifier subsumes c) {
        accurateClassifier.numerosity=
```
accurateClassifier.numerosity+c.numerosity;
remove c;
}
}

This algorithm would reduce the chances of an over-general classifier subsuming other classifiers which are potentially accurate even though they were not as accurate as the over-general one when subsumption took place at some early point. Over-generality of dominant classifiers potentially increases the effects of aliasing, because aliasing is due to lack of specificity in the input string which is matched with the classifier conditions. Replacing the original subsumption algorithm of the Butz-Wilson XCS by one such as Barry’s suggestion could therefore possibly encourage more accurate learning in some environments, and even reduce effects of other persistent problems.
Chapter 12: Conclusion

There are an important number of fields of research in Machine Learning, and so this paper tries to combine many areas, even though each could be explored in much more depth. The nature of the project, quite research-based, meant that it was difficult to combine both a detailed description and analysis of the fields, and a considerable amount of code to exploit all ideas behind Machine Learning, Classifier Systems, Genetic Algorithms and Finite State Worlds.

In writing this paper I learnt that research does not involve reading one book and rewriting it. There is a considerable amount of literature on the project’s areas of concern, and sources cannot be entirely trusted separately but need to be read and reread together. These fields also involve understanding a great many concepts which I had never encountered before, and making sense of them all was a significant part of the work. Reading expert papers when one is not an expert, and analysing them deeply enough to understand the problems, can be very time consuming. Therefore in this dissertation, it seemed essential to explain in detail all the background notions which are unknown to most computer scientist undergraduates, in order to lay the bases for the experimental analysis of the classifier system.

Many ideas and issues were raised in the literature review, which in the end were not implemented. As the code implementation progressed it seemed more interesting to explore the way the Butz-Wilson XCS deals with the finite state world, and how its subsumption algorithm has consequences on the learning abilities of the system.

This paper has described the basics of learning classifier systems and how they operate, including explanations of various environments and the genetic algorithms used in order to ‘learn’. It has concentrated on the Butz-Wilson X-Classifier System which was used for implementation purposes, to carry out experiments on the effectiveness of the XCS algorithm in the domain of machine learning. It has introduced the possibility that there is still some work to be done
on XCS, in particular concerning the subsumption algorithm used upon the population of classifiers. It has discussed the issues of over generality and inaccuracy of some classifiers which remain and survive in the population due to certain properties of the system.
Bibliography


Barry, AM (1999) "Aliasing in XCS and the Consecutive State Problem : 2 - Solutions" submitted to the International Conference on Genetic and Evolutionary Computing, 14-17, Jul 1999


Booker, LB (1982) "Intelligent behavior as an adaption to the task environment" PhD Dissertation, Univ. of Michigan, Logic of Computers Group, Ann Arbor, MI


Goldberg, DE, Horn J & Deb, K (1992) "What makes a problem hard for a classifier system?" presented at the First International Workshop on Learning Classifier Systems, Johnson Space Center, Houston, USA


Horn, J (1997) "The Nature of Niching: Genetic Algorithms and the Evolution of Optimal, Cooperative Populations" Department of Computer Science, University of Illinois at Urbana-Champaign


Kovacs, T & Lanzi, PL (1999) "A Learning Classifier Systems Bibliography" School of Computer Science, University of Birmingham


Kovacs, T (2001) "What Should a Classifier System Learn?" School of Computer Science, University of Birmingham

Kovacs, T (2002) "Learning Classifier Systems Resources" School of Computer Science, University of Birmingham


Lanzi, PL & Riolo, RL (1999) "A Roadmap to the Last Decade of Learning Classifier System Research (From 1988 to 1999)" in Lanzi et al. [21], 33-62


Mahfoud, SW (1995) "Niching Methods for Genetic Algorithms” Illinois Genetic Algorithms Laboratory (IlliGAL), Department of General Engineering, University of Illinois at Urbana-Champaign

Michalski, RS, Dietterich, TG (1986), "Machine Learning, an Artificial Intelligence Approach" Volume 2, chapter Learning to predict sequences, 63-106, Morgan Kaufmann, Los Altos, CA


Wilson, SW & Goldberg, DE (1989) "A critical review of classifier systems" Proceedings of the Third International Conference on Genetic Algorithms [ICGA89], 244-255, Morgan Kauffman


Wilson, SW (1987) "Classifier Systems and the Animat Problem" Machine Learning, 2, 199-228

Wilson, SW (1988) "Bid Competition and Specificity Reconsidered" Complex Systems, 2(5), 705-723

Wilson, SW (1994) "ZCS, a zeroth level classifier system" Evolutionary Computation 1(2), 1-18

Wilson, SW (1995) "Classifier fitness based on accuracy" Evolutionary Computation 3(2), 149-175


Web Resources

This information is correct at the time of publishing.

[1] www.hyperdictionary.com
An online dictionary of technical words in specific fields

The machine learning research group homepage

School of Information and Computer Science of the University of California, Irvine

Navy Center for Applied Research in Artificial Intelligence: The Genetic Algorithms archive

The University of Bristol (U.K.) Computer Science Department homepage

The computer society of the IEEE

The Learning Classifier Systems Web (Alwyn Barry, University of Bath)

[8] www-2.cs.cmu.edu
The School of Computer Science at Carnegie Mellon University, Pennsylvania, USA

Internet Frequently Asked Questions Archive – Online Education

Queen’s university Belfast, School of Computer Science, Matt Sullivan’s page

Genetic Algorithms warehouse, from the Artificial Intelligence Depot


Bull, L "Learning Classifier Systems: A Brief Introduction" Faculty of Computing, Engineering & Mathematical Sciences, University of the West of England
Appendices

A) Java XCS Code

This Appendix includes only the code which was modified in order to run the tests: the XCS and FSWEnvironment classes. The entire code making up the XCS program figures on the accompanying CD in the ‘XCS Code - Appendix A’ folder.

A.1 XCS.java

```java
public class XCS implements Serializable {
    private Environment env;
    private XClassifierSet pop;
    private File outFile;
    private File popFile;
    private int maxProblems=20000;
    private int nrExps=10;
    private static XCSConstants cons;
    public static void main(String args[])
    {
        String envFileString=null;
        Environment e = null;
        XCSConstants.setSeed(1 + (new Date()).getTime() % 10000);
        if(args.length > 4 && (args[0].equals("maze") || args[0].equals("mp") || args[0].equals("fsw"))){
            if(args[0].equals("maze")){
                e=new MazeEnvironment(args[3],(new Integer(args[4])).intValue());
            }else if(args[0].equals("fsw")){
                System.out.println("Construct finite state world environment, condition length " + args[4]);
                e=new FSWEnvironment(args[3], (new Integer(args[4])).intValue());
            }else if(args[0].equals("mp")){
                System.out.println("Construct Multiplexer problem of length "+args[3]+" and payoff type "+args[4]);
                e=new MPEnvironment((new Integer(args[3])).intValue(), (new Integer(args[4])).intValue());
            }
        }else{
            System.out.println("Usage: java XCS problemType(maze,mp,fsw) outputFile popFile {problemLength(1-1), mazeEnvironment} {payoffLandscape(0,1), codingLength(2,3)} {codingLength, fswEnvironment} [MaxNumberOfTrials] [NumberOfExperiments] ");
            return;
        }
        XCS xcs=new XCS(e, args[1], args[2]);
        if(args.length > 5){
            xcs.setNumberOfTrials((new Integer(args[5])).intValue());
        }
        if(args.length > 6){
            xcs.setNumberOfExperiments((new Integer(args[6])).intValue());
        }
        xcs.runXCS();
        return;
    }
}
```
public XCS(Environment e, String outFileString, String popFileString)
{
    env=e;
    outFile = new File(outFileString);
    popFile = new File(popFileString);
    pop=null;
    cons = new XCSConstants();
}

public void setNumberOfTrials(int trials)
{
    maxProblems=trials;
}

public void setNumberOfExperiments(int exps)
{
    nrExps=exps;
}

public void runXCS()
{
    FileWriter fW=null;
    BufferedWriter bW=null;
    PrintWriter pW = null;
    PrintWriter popW = null;
    try{
        fW = new FileWriter(outFile);
        bW = new BufferedWriter(fW);
        pW = new PrintWriter(bW);
        /* Added to create a population writer */
        popW = new PrintWriter(new BufferedWriter(new FileWriter(popFile)));
    }catch(Exception e){System.out.println("Mistake in create file Writers"+e);
    startExperiments(pW, popW);
    try{
        pW.flush();
        bW.flush();
        fW.flush();
        popW.flush();
        popW.close();
    }catch(Exception e){System.out.println("Mistake in closing the file writer!"+e);
}
}

private void startExperiments(PrintWriter pW, PrintWriter popW)
{
    for(int expCounter=0; expCounter < nrExps; expCounter++){
        pW.println("# Next Experiment");
        System.out.println("Experiment Nr."+(expCounter+1));
        pop = new XClassifierSet(env.getNrActions());
        if(!env.isMultiStepProblem())
        doOneSingleStepExperiment(pW);
        else
        doOneMultiStepExperiment(pW);
        writePopulation(popW);
        pop=null;
    }
}

void doOneSingleStepExperiment(PrintWriter pW)
{
    int explore=0;
    int[] correct = new int[50];
    double[] sysError = new double[50];
    for (int exploreProbC=0; exploreProbC < maxProblems; exploreProbC++){
        explore = (explore+1)%2;
        String state = env.resetState();
        if(explore==1){
            doOneSingleStepProblemExplore(state, exploreProbC);
        }else{
            doOneSingleStepProblemExploit(state, exploreProbC, correct, sysError);
        }
        if(exploreProbC%50==0 && explore==0 && exploreProbC>0){
            pW.println("Experiment "+exploreProbC+1);
private void doOneSingleStepProblemExplore(String state, int counter) {
    XClassifierSet matchSet = new XClassifierSet(state, pop, counter, env.getNrActions());
    PredictionArray predictionArray = new PredictionArray(matchSet, env.getNrActions());
    int actionWinner = predictionArray.randomActionWinner();
    XClassifierSet actionSet = new XClassifierSet(matchSet, actionWinner);
    double reward = env.executeAction(actionWinner);
    actionSet.updateSet(0.0, reward);
    actionSet.runGA(counter, state, env.getNrActions());
}

private void doOneSingleStepProblemExploit(String state, int counter, int[] correct, double[] sysError) {
    XClassifierSet matchSet = new XClassifierSet(state, pop, counter, env.getNrActions());
    PredictionArray predictionArray = new PredictionArray(matchSet, env.getNrActions());
    int actionWinner = predictionArray.bestActionWinner();
    double reward = env.executeAction(actionWinner);
    if (env.wasCorrect()) {
        correct[counter % 50] = 1;
    } else {
        correct[counter % 50] = 0;
        sysError[counter % 50] = Math.abs(reward - predictionArray.getBestValue());
    }
}

void doOneMultiStepExperiment(PrintWriter pW) {
    int explore = 0, exploreStepCounter = 0;
    int[] stepsToFood = new int[50];
    double[] sysError = new double[50];
    for (int exploreTrialC = 0; exploreTrialC < maxProblems; exploreTrialC += explore) {
        if (explore == 1) {
            exploreStepCounter = doOneMultiStepProblemExplore(state, exploreStepCounter);
        } else {
            doOneMultiStepProblemExploit(state, stepsToFood, sysError, exploreTrialC, exploreStepCounter);
        }
        if (exploreTrialC % 50 == 0 && explore == 0 && exploreTrialC > 0) {
            writePerformance(pW, stepsToFood, sysError, exploreTrialC);
        }
    }
}

private int doOneMultiStepProblemExplore(String state, int stepCounter) {
    XClassifierSet prevActionSet = null;
    double prevReward = 0.0;
    int steps;
    String prevState = null;
    for (steps = 0; steps < cons.teletransportation; steps++) {
        XClassifierSet matchSet = new XClassifierSet(state, pop, stepCounter + steps, env.getNrActions());
        PredictionArray predictionArray = new PredictionArray(matchSet, env.getNrActions());
        int actionWinner = predictionArray.randomActionWinner();
        XClassifierSet actionSet = new XClassifierSet(matchSet, actionWinner);
        double reward = env.executeAction(actionWinner);
        if (prevActionSet != null) {
            prevActionSet.confirmClassifiersInSet();
            prevActionSet.updateSet(predictionArray.getBestValue(), prevReward);
            prevActionSet.runGA(stepCounter + steps, prevState, env.getNrActions());
        }
        if (env.doReset()) {
            actionSet.confirmClassifiersInSet();
            actionSet.updateSet(0.0, reward);
            actionSet.runGA(stepCounter + steps, state, env.getNrActions());
            break;
        }
        prevActionSet = actionSet;
    }
}
prevReward = reward;
prevState = state;
state = env.getCurrentState();
}
return stepCounter + steps;
}

private void doOneMultiStepProblemExploit(String state, int[] stepsToGoal, double[] sysError, int trialCounter, int stepCounter)
{
XClassifier prevActionSet = null;
double prevReward = 0., prevPrediction = 0.;
int steps;
sysError[trialCounter % 50] = 0.;
for( steps = 0; steps < cons.teletransportation; steps++) {
    XClassifier matchSet = new XClassifier(state, pop, stepCounter, env.getNrActions());
    PredictionArray predictionArray = new PredictionArray(matchSet, env.getNrActions());
    int actionWinner = predictionArray.bestActionWinner();
    XClassifier actionSet = new XClassifier(matchSet, actionWinner);
    double reward = env.executeAction( actionWinner );
    if(prevActionSet != null) {
        prevActionSet.confirmClassifiersInSet();
        prevActionSet.updateSet(predictionArray.getBestValue(), prevReward);
        sysError[trialCounter % 50] += (double) Math.abs(cons.gamma * predictionArray.getValue(actionWinner) + prevReward - prevPrediction) / (double) env.getMaxPayoff();
    }
    if(env.doReset()) {
        actionSet.confirmClassifiersInSet();
        actionSet.updateSet(predictionArray.getBestValue(), prevReward);
        sysError[trialCounter % 50] += (double) Math.abs(reward - predictionArray.getValue(actionWinner)) / (double) env.getMaxPayoff();
        steps++;
        break;
    }
    prevActionSet = actionSet;
    prevPrediction = predictionArray.getValue(actionWinner);
    prevReward = reward;
    state = env.getCurrentState();
}
sysError[trialCounter % 50] /= steps;
stepsToGoal[trialCounter % 50] = steps;
}

private void writePerformance(PrintWriter pW, int[] performance, double[] sysError, int exploreProbC)
{
double perf = 0.;
double serr = 0.;
for(int i = 0; i < 50; i++) {
    perf += performance[i];
    serr += sysError[i];
}
perf /= 50.;
serr /= 50.;
/* Modified to give out CSV format */
pW.println(exploreProbC + "," + (float) perf + "," + (float) serr + "," + pop.getSize());
System.out.println(exploreProbC + "," + (float) perf + "," + (float) serr + "," + pop.getSize());
}

private void writePopulation(PrintWriter popW)
{
    popW.println("Next Experiment");
    pop.printSet(popW);
}
public class FSWEnvironment implements Environment, Serializable
{
    private final boolean debug = false;
    private int nrActions = 2;
    private int stateLength = 3;
    private int nrChars;
    private int nrLines = 0;
    private int nrStates = 0;
    private int maxPayoff = 0;
    private int conLength = 0;
    private Vector states = new Vector();
    private Vector start = new Vector();
    private int current;
    private boolean reset;

    public FSWEnvironment(String inFileString, int conditionLength)
    {
        FileReader fr = null;
        BufferedReader br = null;
        conLength = conditionLength;
        try{
            fr = new FileReader(inFileString);
            br = new BufferedReader(fr);
            while (br.ready()){
                String[] values = inFileString.trim();
                if (nrLines == 0)
                    nrStates = Integer.parseInt(values[0].trim());
                else {
                    int[] oneState = new int[5];
                    oneState[0] = Integer.parseInt(values[0].trim());
                    oneState[1] = Integer.parseInt(values[1].trim());
                    oneState[2] = Integer.parseInt(values[2].trim());
                    oneState[3] = Integer.parseInt(values[3].trim());
                    if (oneState[3] == 0)
                        start.addElement(oneState[0]);
                    else {
                        oneState[4] = Integer.parseInt(values[4].trim());
                        maxPayoff = oneState[4];
                    }
                }
                states.addElement(oneState);
                nrLines ++;
            }
        }
    }
}
```java
} catch (Exception e) {
    System.out.println("Could not Read File!" + e);
}

System.out.println("Got environment: ");
for (int i = 0; i < states.size(); i++) {
    int[] currentState = (int[]) states.elementAt(i);
}
reset=false;
resetState();

private static String toBinary(int value, int bits)
{
    char[] result = new char[bits];
    for (int i = 0; i < bits; i++)
        result[i] = '0';
    if (value > 0) {
        String binary = Integer.toBinaryString(value);
        int stringIndex = binary.length() - 1;
        int index = bits - 1;
        while ((index >= 0) && (stringIndex >= 0)) {
            result[index] = binary.charAt(stringIndex);
            --index;
            --stringIndex;
        }
    }
    return new String(result);
}

public String resetState(){
    if (debug) System.out.println("Reset");
    java.util.Random generator = new java.util.Random(System.currentTimeMillis());
    newStart = generator.nextInt(start.size());
    current = newStart;
    reset = false;
    return toBinary(0,conLength);
}

public String getCurrentState(){
    return toBinary(current,conLength);
}

public double executeAction(int action){
    if (debug) System.out.println("Action: "+ action);
    if (action < 0 || action > 1){
        System.out.println("Not an action!");
        System.exit(0);
    }
}
```
int[] currentArray = (int[])states.elementAt(current);
if (debug) System.out.print(" From: "+current);
    current = currentArray[1 + action];
if (debug) System.out.println(" to: "+current);
    currentArray = (int[])states.elementAt(current);
if(currentArray[3]==2) {
    reset = true;
    if (debug) System.out.println("Reset signalled");
    return currentArray[4];
}
    return 0.;
}
public boolean wasCorrect(){
    return false; //no correct or wrong action in this environment
}
public boolean doReset(){
    return reset;
}
public int getConditionLength(){
    return conLength;
}
public int getMaxPayoff(){
    return maxPayoff;
}
public boolean isMultiStepProblem(){
    return true;
}
public int getNrActions(){
    return nrActions;
}
}
• Reward value (for end states only)

The other files given for tests are the population tables and performance tables for certain experiments which went wrong (graphs and explanation in document).

**The finite state world** used for this test was:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,</td>
<td>1,</td>
<td>2,</td>
<td>0</td>
</tr>
<tr>
<td>1,</td>
<td>0,</td>
<td>2,</td>
<td>1</td>
</tr>
<tr>
<td>2,</td>
<td>3,</td>
<td>4,</td>
<td>1</td>
</tr>
<tr>
<td>3,</td>
<td>4,</td>
<td>4,</td>
<td>1</td>
</tr>
<tr>
<td>4,</td>
<td>4,</td>
<td>4,</td>
<td>2,1000</td>
</tr>
</tbody>
</table>

//number of states in the finite state world
//state 0: start state
//state 1
//state 2
//state 3
//state 4: end state with reward of 1000

**Test 1 population table** for experiment 3 was:

<table>
<thead>
<tr>
<th>Cond Action</th>
<th>Prediction</th>
<th>Error</th>
<th>Fitness</th>
<th>Num</th>
<th>Exp</th>
<th>[A]size</th>
<th>birth</th>
</tr>
</thead>
<tbody>
<tr>
<td>0##0</td>
<td>1 833.8862</td>
<td>159.07</td>
<td>1.25E-08</td>
<td>4</td>
<td>1558</td>
<td>81.9329</td>
<td>1237</td>
</tr>
<tr>
<td>00##</td>
<td>1 833.9613</td>
<td>159.14</td>
<td>8.04E-08</td>
<td>26</td>
<td>1884</td>
<td>81.9186</td>
<td>1237</td>
</tr>
<tr>
<td>#01#</td>
<td>0 862.8875</td>
<td>159.34</td>
<td>0.115875</td>
<td>7</td>
<td>368</td>
<td>105.3122</td>
<td>1223</td>
</tr>
<tr>
<td>#0##</td>
<td>1 833.9613</td>
<td>159.14</td>
<td>3.40E-08</td>
<td>11</td>
<td>1802</td>
<td>81.9186</td>
<td>1237</td>
</tr>
<tr>
<td>##1#</td>
<td>0 862.8875</td>
<td>159.34</td>
<td>0.033107</td>
<td>2</td>
<td>353</td>
<td>105.3122</td>
<td>1223</td>
</tr>
<tr>
<td>0###</td>
<td>1 833.9613</td>
<td>159.14</td>
<td>1.24E-08</td>
<td>4</td>
<td>1745</td>
<td>81.9186</td>
<td>1237</td>
</tr>
<tr>
<td>####</td>
<td>0 726.842</td>
<td>208.64</td>
<td>0.23571</td>
<td>80</td>
<td>803</td>
<td>100.0495</td>
<td>1223</td>
</tr>
<tr>
<td>###0</td>
<td>1 833.8862</td>
<td>159.07</td>
<td>6.23E-09</td>
<td>2</td>
<td>1170</td>
<td>81.9329</td>
<td>1237</td>
</tr>
<tr>
<td>##10</td>
<td>0 710 7.87E-05</td>
<td>0.999999</td>
<td>29</td>
<td>176</td>
<td>122.977</td>
<td>1223</td>
<td></td>
</tr>
<tr>
<td>###0</td>
<td>0 619.1783</td>
<td>113.69</td>
<td>2.17E-07</td>
<td>6</td>
<td>349</td>
<td>111.4178</td>
<td>1223</td>
</tr>
<tr>
<td>####</td>
<td>1 710 2.04E-06</td>
<td>1.38E-05</td>
<td>38</td>
<td>534</td>
<td>85.12169</td>
<td>1237</td>
<td></td>
</tr>
<tr>
<td>##10</td>
<td>1 1000 8.38E-25</td>
<td>0.173109</td>
<td>5</td>
<td>276</td>
<td>76.8893</td>
<td>1219</td>
<td></td>
</tr>
<tr>
<td>##0#</td>
<td>0 504.1 1.19E-06</td>
<td>0.999999</td>
<td>10</td>
<td>139</td>
<td>92.9053</td>
<td>1204</td>
<td></td>
</tr>
<tr>
<td>0##1#</td>
<td>1 1000</td>
<td>0.831301</td>
<td>24</td>
<td>247</td>
<td>76.64179</td>
<td>1219</td>
<td></td>
</tr>
<tr>
<td>#####</td>
<td>1 833.9613</td>
<td>159.14</td>
<td>3.09E-09</td>
<td>1</td>
<td>378</td>
<td>81.9186</td>
<td>1237</td>
</tr>
</tbody>
</table>

**Test 1 performance table** for experiment 3 was:

<table>
<thead>
<tr>
<th>Trials</th>
<th>Steps</th>
<th>Error</th>
<th>PopSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>2</td>
<td>0.22379</td>
<td>11</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>0.222689</td>
<td>13</td>
</tr>
<tr>
<td>150</td>
<td>2</td>
<td>0.22139</td>
<td>15</td>
</tr>
<tr>
<td>200</td>
<td>2.04</td>
<td>0.209816</td>
<td>10</td>
</tr>
<tr>
<td>250</td>
<td>2.12</td>
<td>0.126983</td>
<td>12</td>
</tr>
<tr>
<td>300</td>
<td>2.24</td>
<td>0.013482</td>
<td>12</td>
</tr>
<tr>
<td>350</td>
<td>2.08</td>
<td>0.004401</td>
<td>14</td>
</tr>
<tr>
<td>400</td>
<td>2</td>
<td>1.38E-05</td>
<td>15</td>
</tr>
<tr>
<td>450</td>
<td>2</td>
<td>2.81E-06</td>
<td>15</td>
</tr>
</tbody>
</table>
C) Test 2 Results

C.1 Test 2.1 Results

The finite state world used for this test was:

```
5,     //number of states in the finite state world
0, 1, 2, 0   //state 0: start state
1, 0, 2, 1   //state 1
2, 3, 4, 1   //state 2
3, 4, 4, 1   //state 3
4, 4, 4, 2, 1000  //state 4: end state with reward of 1000
```

Test 2.1 population table for experiment 6 was:

<table>
<thead>
<tr>
<th>Cond</th>
<th>Action</th>
<th>Prediction</th>
<th>Error</th>
<th>Fitness</th>
<th>Num</th>
<th>Exp</th>
<th>[A]size</th>
<th>birth</th>
</tr>
</thead>
<tbody>
<tr>
<td>###1</td>
<td>1</td>
<td>754.9898 105.5453</td>
<td>6.67E-09</td>
<td>5</td>
<td>569</td>
<td>178.7335</td>
<td>2471</td>
<td></td>
</tr>
<tr>
<td>0#1</td>
<td>1</td>
<td>754.9898 105.5453</td>
<td>2.22E-09</td>
<td>1</td>
<td>564</td>
<td>178.7335</td>
<td>2471</td>
<td></td>
</tr>
<tr>
<td>00##</td>
<td>1</td>
<td>822.2899 156.9349</td>
<td>3.11E-09</td>
<td>5</td>
<td>3697</td>
<td>175.8154</td>
<td>2471</td>
<td></td>
</tr>
<tr>
<td>00#0</td>
<td>1</td>
<td>873.012 160.5913</td>
<td>1.22E-09</td>
<td>2</td>
<td>3106</td>
<td>173.4782</td>
<td>2427</td>
<td></td>
</tr>
<tr>
<td>###0</td>
<td>1</td>
<td>710 9.27E-08</td>
<td>1</td>
<td>160</td>
<td>1938</td>
<td>178.9773</td>
<td>2471</td>
<td></td>
</tr>
<tr>
<td>00#1</td>
<td>1</td>
<td>754.9898 105.5453</td>
<td>2.22E-09</td>
<td>1</td>
<td>532</td>
<td>178.7335</td>
<td>2471</td>
<td></td>
</tr>
<tr>
<td>0###</td>
<td>1</td>
<td>822.2899 156.9349</td>
<td>1.24E-09</td>
<td>2</td>
<td>3440</td>
<td>175.8154</td>
<td>2471</td>
<td></td>
</tr>
<tr>
<td>00##</td>
<td>1</td>
<td>822.2899 156.9349</td>
<td>5.60E-09</td>
<td>9</td>
<td>3322</td>
<td>175.8154</td>
<td>2471</td>
<td></td>
</tr>
<tr>
<td>0#1#</td>
<td>0</td>
<td>855.1822 158.7745</td>
<td>0.011444</td>
<td>5</td>
<td>629</td>
<td>176.0432</td>
<td>2476</td>
<td></td>
</tr>
<tr>
<td>###1#</td>
<td>1</td>
<td>1000 0</td>
<td>1</td>
<td>152</td>
<td>1395</td>
<td>169.9892</td>
<td>2460</td>
<td></td>
</tr>
<tr>
<td>##1##</td>
<td>0</td>
<td>662.4724 183.4092</td>
<td>0.173046</td>
<td>143</td>
<td>1196</td>
<td>184.1367</td>
<td>2476</td>
<td></td>
</tr>
<tr>
<td>#11</td>
<td>0</td>
<td>1000 0</td>
<td>1</td>
<td>17</td>
<td>159</td>
<td>182.3774</td>
<td>2429</td>
<td></td>
</tr>
<tr>
<td>#1#</td>
<td>0</td>
<td>855.1822 158.7745</td>
<td>0.03662</td>
<td>16</td>
<td>462</td>
<td>176.0432</td>
<td>2476</td>
<td></td>
</tr>
<tr>
<td>##0#</td>
<td>0</td>
<td>504.1 1.05E-07</td>
<td>0.999999</td>
<td>41</td>
<td>452</td>
<td>188.616</td>
<td>2451</td>
<td></td>
</tr>
<tr>
<td>###0</td>
<td>0</td>
<td>573.4221 98.03549</td>
<td>0.015464</td>
<td>2</td>
<td>503</td>
<td>182.357</td>
<td>2476</td>
<td></td>
</tr>
<tr>
<td>0###</td>
<td>0</td>
<td>662.4724 183.4092</td>
<td>0.002208</td>
<td>2</td>
<td>225</td>
<td>184.1367</td>
<td>2476</td>
<td></td>
</tr>
<tr>
<td>#0##</td>
<td>0</td>
<td>662.4724 183.4092</td>
<td>0.002425</td>
<td>2</td>
<td>213</td>
<td>184.1367</td>
<td>2476</td>
<td></td>
</tr>
</tbody>
</table>

Test 2.1 performance table for experiment 6 was:

<table>
<thead>
<tr>
<th>Trials</th>
<th>Steps</th>
<th>Error</th>
<th>PopSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>3.16</td>
<td>0.019243</td>
<td>8</td>
</tr>
<tr>
<td>100</td>
<td>2.02</td>
<td>0.00262</td>
<td>10</td>
</tr>
<tr>
<td>150</td>
<td>2</td>
<td>2.13E-05</td>
<td>12</td>
</tr>
<tr>
<td>200</td>
<td>2</td>
<td>3.95E-09</td>
<td>13</td>
</tr>
<tr>
<td>250</td>
<td>2</td>
<td>3.22E-09</td>
<td>13</td>
</tr>
<tr>
<td>300</td>
<td>2</td>
<td>2.72E-09</td>
<td>15</td>
</tr>
<tr>
<td>350</td>
<td>2</td>
<td>2.32E-09</td>
<td>17</td>
</tr>
<tr>
<td>400</td>
<td>2</td>
<td>2.07E-09</td>
<td>18</td>
</tr>
<tr>
<td>450</td>
<td>2</td>
<td>1.96E-09</td>
<td>18</td>
</tr>
<tr>
<td>500</td>
<td>2.18</td>
<td>0.009132</td>
<td>13</td>
</tr>
</tbody>
</table>
C.2 Test 2.2 Results

The finite state world used for this test was:

```
5,    //number of states in the finite state world
0, 1, 2, 0  //state 0: start state
1, 0, 2, 1  //state 1
2, 3, 4, 1  //state 2
3, 4, 4, 1  //state 3
4, 4, 4, 2, 1000 //state 4: end state with reward of 1000
```

Test 2.2 population table for experiment 5 was:

<table>
<thead>
<tr>
<th>Cond</th>
<th>Action</th>
<th>Prediction</th>
<th>Error</th>
<th>Fitness</th>
<th>Num</th>
<th>Exp</th>
<th>[A]size</th>
<th>birth</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0##</td>
<td>1</td>
<td>838.597</td>
<td>161.0114</td>
<td>1.49E-07</td>
<td>49</td>
<td>2819</td>
<td>121.6393</td>
<td>1924</td>
</tr>
<tr>
<td>00##</td>
<td>1</td>
<td>838.597</td>
<td>161.0114</td>
<td>3.96E-08</td>
<td>13</td>
<td>2707</td>
<td>121.6393</td>
<td>1924</td>
</tr>
<tr>
<td>0###</td>
<td>1</td>
<td>838.597</td>
<td>161.0114</td>
<td>1.52E-08</td>
<td>5</td>
<td>2642</td>
<td>121.6393</td>
<td>1924</td>
</tr>
<tr>
<td>#01#</td>
<td>0</td>
<td>829.1113</td>
<td>153.62</td>
<td>0.009755</td>
<td>1</td>
<td>510</td>
<td>150.981</td>
<td>1892</td>
</tr>
<tr>
<td>####</td>
<td>0</td>
<td>681.3774</td>
<td>179.4458</td>
<td>0.968093</td>
<td>145</td>
<td>1049</td>
<td>148.4867</td>
<td>1917</td>
</tr>
<tr>
<td>####</td>
<td>1</td>
<td>838.597</td>
<td>161.0114</td>
<td>2.74E-08</td>
<td>9</td>
<td>1428</td>
<td>121.6393</td>
<td>1924</td>
</tr>
<tr>
<td>#0#</td>
<td>1</td>
<td>710</td>
<td>1.41E-06</td>
<td>1</td>
<td>68</td>
<td>580</td>
<td>143.1677</td>
<td>1924</td>
</tr>
<tr>
<td>#1#</td>
<td>1</td>
<td>1000</td>
<td>2.15E-41</td>
<td>1</td>
<td>18</td>
<td>441</td>
<td>93.95495</td>
<td>1906</td>
</tr>
<tr>
<td>#1#</td>
<td>0</td>
<td>829.1113</td>
<td>153.62</td>
<td>0.029266</td>
<td>3</td>
<td>104</td>
<td>150.981</td>
<td>1892</td>
</tr>
<tr>
<td>0#1#</td>
<td>0</td>
<td>829.1113</td>
<td>153.6198</td>
<td>0.027508</td>
<td>3</td>
<td>65</td>
<td>150.981</td>
<td>1892</td>
</tr>
</tbody>
</table>

Test 2.2 performance table for experiment 5 was:

<table>
<thead>
<tr>
<th>Trials</th>
<th>Steps</th>
<th>Error</th>
<th>PopSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>2</td>
<td>0.224944</td>
<td>4</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>0.226938</td>
<td>6</td>
</tr>
<tr>
<td>150</td>
<td>2</td>
<td>0.216932</td>
<td>8</td>
</tr>
<tr>
<td>200</td>
<td>2</td>
<td>0.220083</td>
<td>10</td>
</tr>
<tr>
<td>250</td>
<td>2</td>
<td>0.226164</td>
<td>10</td>
</tr>
<tr>
<td>300</td>
<td>2</td>
<td>0.169035</td>
<td>13</td>
</tr>
<tr>
<td>350</td>
<td>2</td>
<td>0.143832</td>
<td>15</td>
</tr>
<tr>
<td>400</td>
<td>2</td>
<td>0.145689</td>
<td>8</td>
</tr>
<tr>
<td>450</td>
<td>2.08</td>
<td>0.143238</td>
<td>9</td>
</tr>
</tbody>
</table>
D) Test 3 Results

The finite state world used for these tests was:

<table>
<thead>
<tr>
<th>State</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,</td>
<td>10</td>
</tr>
<tr>
<td>0, 1, 5, 0</td>
<td>0</td>
</tr>
<tr>
<td>1, 6, 2, 1</td>
<td>1</td>
</tr>
<tr>
<td>2, 3, 7, 1</td>
<td>2</td>
</tr>
<tr>
<td>3, 8, 4, 1</td>
<td>3</td>
</tr>
<tr>
<td>4, 5, 10, 1</td>
<td>4</td>
</tr>
<tr>
<td>5, 1, 1, 1</td>
<td>5</td>
</tr>
<tr>
<td>6, 2, 2, 1</td>
<td>6</td>
</tr>
<tr>
<td>7, 3, 3, 1</td>
<td>7</td>
</tr>
<tr>
<td>8, 4, 4, 1</td>
<td>8</td>
</tr>
<tr>
<td>9, 5, 5, 1</td>
<td>9</td>
</tr>
<tr>
<td>10, 10, 10, 2, 1000</td>
<td>10</td>
</tr>
</tbody>
</table>

Test 3 population table for experiment 6 was:

<table>
<thead>
<tr>
<th>Cond</th>
<th>Action</th>
<th>Prediction</th>
<th>Error</th>
<th>Fitness</th>
<th>Num</th>
<th>Exp</th>
<th>[A]size</th>
<th>birth</th>
</tr>
</thead>
<tbody>
<tr>
<td>##10</td>
<td>1</td>
<td>354.6339</td>
<td>4.039842</td>
<td>1</td>
<td>69</td>
<td>3159</td>
<td>66.86472</td>
<td>25515</td>
</tr>
<tr>
<td>1###</td>
<td>1</td>
<td>709.9179</td>
<td>0.100404</td>
<td>1</td>
<td>69</td>
<td>927</td>
<td>67.68619</td>
<td>25505</td>
</tr>
<tr>
<td>#100</td>
<td>1</td>
<td>1000</td>
<td>0</td>
<td>0.999838</td>
<td>74</td>
<td>3442</td>
<td>72.64607</td>
<td>25519</td>
</tr>
<tr>
<td>#101</td>
<td>1</td>
<td>251.3195</td>
<td>1.484454</td>
<td>1</td>
<td>68</td>
<td>1403</td>
<td>64.85462</td>
<td>25513</td>
</tr>
<tr>
<td>1###</td>
<td>1</td>
<td>709.8896</td>
<td>0.209705</td>
<td>1</td>
<td>56</td>
<td>845</td>
<td>58.82749</td>
<td>25486</td>
</tr>
<tr>
<td>#001</td>
<td>1</td>
<td>353.0669</td>
<td>5.553881</td>
<td>0.99996</td>
<td>72</td>
<td>3375</td>
<td>72.31082</td>
<td>25477</td>
</tr>
<tr>
<td>#111</td>
<td>1</td>
<td>504.0065</td>
<td>0.074596</td>
<td>1</td>
<td>57</td>
<td>843</td>
<td>55.83817</td>
<td>25491</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>182.5282</td>
<td>4.164416</td>
<td>1</td>
<td>51</td>
<td>774</td>
<td>57.32795</td>
<td>25427</td>
</tr>
<tr>
<td>#110</td>
<td>0</td>
<td>350.6667</td>
<td>8.323915</td>
<td>0.993524</td>
<td>9</td>
<td>761</td>
<td>98.71462</td>
<td>25443</td>
</tr>
<tr>
<td>#011</td>
<td>1</td>
<td>709.9666</td>
<td>0.059813</td>
<td>1</td>
<td>63</td>
<td>1447</td>
<td>61.60793</td>
<td>25511</td>
</tr>
<tr>
<td>##1#</td>
<td>0</td>
<td>488.7884</td>
<td>27.89798</td>
<td>0.896177</td>
<td>59</td>
<td>1703</td>
<td>66.56329</td>
<td>25503</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>248.2759</td>
<td>2.026254</td>
<td>0.985532</td>
<td>47</td>
<td>335</td>
<td>50.92909</td>
<td>25453</td>
</tr>
<tr>
<td>##01</td>
<td>0</td>
<td>254.5712</td>
<td>5.580907</td>
<td>0.789318</td>
<td>43</td>
<td>119</td>
<td>43.40958</td>
<td>25494</td>
</tr>
<tr>
<td>#1#0</td>
<td>0</td>
<td>217.3727</td>
<td>59.1175</td>
<td>0.768101</td>
<td>58</td>
<td>38</td>
<td>72.73659</td>
<td>25499</td>
</tr>
<tr>
<td>#000</td>
<td>1</td>
<td>581.5476</td>
<td>250.381</td>
<td>4.25E-05</td>
<td>1</td>
<td>17</td>
<td>59.63643</td>
<td>25486</td>
</tr>
<tr>
<td>#11#</td>
<td>0</td>
<td>455.1211</td>
<td>77.39431</td>
<td>0.001022</td>
<td>1</td>
<td>9</td>
<td>80.26112</td>
<td>25473</td>
</tr>
<tr>
<td>000#</td>
<td>0</td>
<td>253.1191</td>
<td>3.239629</td>
<td>0.069704</td>
<td>1</td>
<td>12</td>
<td>46.01349</td>
<td>25459</td>
</tr>
<tr>
<td>#0#1</td>
<td>0</td>
<td>379.1523</td>
<td>142.8678</td>
<td>7.02E-04</td>
<td>1</td>
<td>10</td>
<td>51.94547</td>
<td>25459</td>
</tr>
</tbody>
</table>
### Test 3 performance table for experiment 6 was:

<table>
<thead>
<tr>
<th>Trials</th>
<th>Steps</th>
<th>Error</th>
<th>PopSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>7</td>
<td>0.177347</td>
<td>35</td>
</tr>
<tr>
<td>100</td>
<td>6.86</td>
<td>0.152881</td>
<td>49</td>
</tr>
<tr>
<td>150</td>
<td>6.7</td>
<td>0.140174</td>
<td>62</td>
</tr>
<tr>
<td>200</td>
<td>7.06</td>
<td>0.125862</td>
<td>54</td>
</tr>
<tr>
<td>250</td>
<td>6.88</td>
<td>0.126907</td>
<td>58</td>
</tr>
<tr>
<td>300</td>
<td>6.7</td>
<td>0.104162</td>
<td>61</td>
</tr>
<tr>
<td>350</td>
<td>5.94</td>
<td>0.033833</td>
<td>43</td>
</tr>
<tr>
<td>400</td>
<td>5.88</td>
<td>0.026145</td>
<td>29</td>
</tr>
<tr>
<td>450</td>
<td>5.72</td>
<td>0.022702</td>
<td>22</td>
</tr>
<tr>
<td>500</td>
<td>5</td>
<td>0.01114</td>
<td>29</td>
</tr>
<tr>
<td>550</td>
<td>5</td>
<td>0.008236</td>
<td>23</td>
</tr>
<tr>
<td>600</td>
<td>5</td>
<td>0.00629</td>
<td>18</td>
</tr>
<tr>
<td>650</td>
<td>5</td>
<td>0.00487</td>
<td>20</td>
</tr>
<tr>
<td>700</td>
<td>5</td>
<td>0.005751</td>
<td>25</td>
</tr>
<tr>
<td>750</td>
<td>5</td>
<td>0.004704</td>
<td>25</td>
</tr>
<tr>
<td>800</td>
<td>5</td>
<td>0.005109</td>
<td>24</td>
</tr>
<tr>
<td>850</td>
<td>5</td>
<td>0.005525</td>
<td>29</td>
</tr>
<tr>
<td>900</td>
<td>5</td>
<td>0.002963</td>
<td>27</td>
</tr>
<tr>
<td>950</td>
<td>5</td>
<td>0.005168</td>
<td>21</td>
</tr>
<tr>
<td>1000</td>
<td>5</td>
<td>0.005492</td>
<td>24</td>
</tr>
<tr>
<td>1050</td>
<td>5</td>
<td>0.005881</td>
<td>29</td>
</tr>
<tr>
<td>1100</td>
<td>5</td>
<td>0.002755</td>
<td>20</td>
</tr>
<tr>
<td>1150</td>
<td>5</td>
<td>0.004469</td>
<td>23</td>
</tr>
<tr>
<td>1200</td>
<td>5</td>
<td>0.008946</td>
<td>24</td>
</tr>
<tr>
<td>1250</td>
<td>5</td>
<td>0.007467</td>
<td>24</td>
</tr>
<tr>
<td>1300</td>
<td>5</td>
<td>0.013005</td>
<td>26</td>
</tr>
<tr>
<td>1350</td>
<td>5</td>
<td>0.006097</td>
<td>27</td>
</tr>
<tr>
<td>1400</td>
<td>5</td>
<td>0.007314</td>
<td>20</td>
</tr>
<tr>
<td>1450</td>
<td>5</td>
<td>0.006213</td>
<td>28</td>
</tr>
<tr>
<td>1500</td>
<td>5</td>
<td>0.00576</td>
<td>19</td>
</tr>
<tr>
<td>1550</td>
<td>5</td>
<td>0.003214</td>
<td>23</td>
</tr>
<tr>
<td>1600</td>
<td>5</td>
<td>0.002551</td>
<td>21</td>
</tr>
<tr>
<td>1650</td>
<td>5</td>
<td>0.004954</td>
<td>23</td>
</tr>
<tr>
<td>1700</td>
<td>5</td>
<td>0.005588</td>
<td>23</td>
</tr>
<tr>
<td>1750</td>
<td>5</td>
<td>0.005057</td>
<td>21</td>
</tr>
<tr>
<td>1800</td>
<td>5</td>
<td>0.006984</td>
<td>23</td>
</tr>
<tr>
<td>1850</td>
<td>5</td>
<td>0.0057</td>
<td>26</td>
</tr>
<tr>
<td>1900</td>
<td>5</td>
<td>0.006253</td>
<td>21</td>
</tr>
<tr>
<td>1950</td>
<td>5</td>
<td>0.005023</td>
<td>24</td>
</tr>
</tbody>
</table>

### Test 3 population table for experiment 8 was:

<table>
<thead>
<tr>
<th>Cond</th>
<th>Action</th>
<th>Prediction</th>
<th>Error</th>
<th>Fitness</th>
<th>Num</th>
<th>Exp</th>
<th>A size</th>
<th>birth</th>
</tr>
</thead>
<tbody>
<tr>
<td>###</td>
<td>0</td>
<td>196.5217</td>
<td>65.16572</td>
<td>0.38797</td>
<td>151</td>
<td>14022</td>
<td>184.1253</td>
<td>25027</td>
</tr>
<tr>
<td>###</td>
<td>1</td>
<td>441.85</td>
<td>336.9125</td>
<td>0.270128</td>
<td>34</td>
<td>26356</td>
<td>128.292</td>
<td>25034</td>
</tr>
<tr>
<td>#00#</td>
<td>1</td>
<td>139.1675</td>
<td>10.9991</td>
<td>0.542421</td>
<td>40</td>
<td>4643</td>
<td>172.0497</td>
<td>25011</td>
</tr>
<tr>
<td>##1#</td>
<td>1</td>
<td>216.1478</td>
<td>63.8594</td>
<td>9.23E-04</td>
<td>2</td>
<td>6524</td>
<td>150.2031</td>
<td>25034</td>
</tr>
</tbody>
</table>
Test 3 performance table for experiment 8 was:

<table>
<thead>
<tr>
<th>Trials</th>
<th>Steps</th>
<th>Error</th>
<th>PopSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>7.08</td>
<td>0.159782</td>
<td>25</td>
</tr>
<tr>
<td>100</td>
<td>7.1</td>
<td>0.155755</td>
<td>32</td>
</tr>
<tr>
<td>150</td>
<td>7.28</td>
<td>0.149958</td>
<td>25</td>
</tr>
<tr>
<td>200</td>
<td>7.14</td>
<td>0.152345</td>
<td>37</td>
</tr>
<tr>
<td>250</td>
<td>7.08</td>
<td>0.151314</td>
<td>42</td>
</tr>
<tr>
<td>300</td>
<td>7.04</td>
<td>0.150775</td>
<td>33</td>
</tr>
<tr>
<td>350</td>
<td>7.04</td>
<td>0.153062</td>
<td>36</td>
</tr>
<tr>
<td>Value</td>
<td><em>x</em></td>
<td><em>y</em></td>
<td><em>z</em></td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>400</td>
<td>7</td>
<td>0.150679</td>
<td>37</td>
</tr>
<tr>
<td>450</td>
<td>7</td>
<td>0.150275</td>
<td>33</td>
</tr>
<tr>
<td>500</td>
<td>7.72</td>
<td>0.162238</td>
<td>7</td>
</tr>
<tr>
<td>550</td>
<td>7.08</td>
<td>0.16221</td>
<td>14</td>
</tr>
<tr>
<td>600</td>
<td>7.2</td>
<td>0.152594</td>
<td>10</td>
</tr>
<tr>
<td>650</td>
<td>7.08</td>
<td>0.15674</td>
<td>17</td>
</tr>
<tr>
<td>700</td>
<td>7.48</td>
<td>0.175003</td>
<td>11</td>
</tr>
<tr>
<td>750</td>
<td>7.46</td>
<td>0.177017</td>
<td>9</td>
</tr>
<tr>
<td>800</td>
<td>7.68</td>
<td>0.171537</td>
<td>9</td>
</tr>
<tr>
<td>850</td>
<td>7.14</td>
<td>0.16197</td>
<td>14</td>
</tr>
<tr>
<td>900</td>
<td>7.12</td>
<td>0.157329</td>
<td>17</td>
</tr>
<tr>
<td>950</td>
<td>7.12</td>
<td>0.155395</td>
<td>19</td>
</tr>
<tr>
<td>1000</td>
<td>7.14</td>
<td>0.154106</td>
<td>18</td>
</tr>
<tr>
<td>1050</td>
<td>7.16</td>
<td>0.153891</td>
<td>20</td>
</tr>
<tr>
<td>1100</td>
<td>6.92</td>
<td>0.155184</td>
<td>23</td>
</tr>
<tr>
<td>1150</td>
<td>7</td>
<td>0.156383</td>
<td>21</td>
</tr>
<tr>
<td>1200</td>
<td>7.12</td>
<td>0.135151</td>
<td>23</td>
</tr>
<tr>
<td>1250</td>
<td>7.06</td>
<td>0.141665</td>
<td>31</td>
</tr>
<tr>
<td>1300</td>
<td>6.9</td>
<td>0.141068</td>
<td>33</td>
</tr>
<tr>
<td>1350</td>
<td>6.92</td>
<td>0.144341</td>
<td>32</td>
</tr>
<tr>
<td>1400</td>
<td>6.76</td>
<td>0.139585</td>
<td>35</td>
</tr>
<tr>
<td>1450</td>
<td>6.76</td>
<td>0.139836</td>
<td>41</td>
</tr>
<tr>
<td>1500</td>
<td>6.92</td>
<td>0.140771</td>
<td>32</td>
</tr>
<tr>
<td>1550</td>
<td>6.94</td>
<td>0.137535</td>
<td>31</td>
</tr>
<tr>
<td>1600</td>
<td>6.94</td>
<td>0.138593</td>
<td>31</td>
</tr>
<tr>
<td>1650</td>
<td>7</td>
<td>0.138511</td>
<td>31</td>
</tr>
<tr>
<td>1700</td>
<td>6.98</td>
<td>0.140973</td>
<td>38</td>
</tr>
<tr>
<td>1750</td>
<td>6.68</td>
<td>0.145482</td>
<td>41</td>
</tr>
<tr>
<td>1800</td>
<td>7.06</td>
<td>0.136029</td>
<td>41</td>
</tr>
<tr>
<td>1850</td>
<td>6.92</td>
<td>0.139782</td>
<td>45</td>
</tr>
<tr>
<td>1900</td>
<td>6.56</td>
<td>0.139017</td>
<td>45</td>
</tr>
<tr>
<td>1950</td>
<td>6.72</td>
<td>0.135858</td>
<td>47</td>
</tr>
</tbody>
</table>