The University of Alberta

LCS: A Learning Classifier System

by

Robert Andrew Chai

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

Department of Computing Science

Edmonton, Alberta
Fall, 1988

ISBN 0-315-45694-9

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Robert Andrew Chai

TITLE OF THESIS: LCS: A Learning Classifier System

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1988

(Signed) [illegible]

Permanent Address:
23 Lancaster Crescent
St. Albert, Alberta
Canada T8N 2N9

Dated 1 September 1988

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **LCS: A Learning Classifier System** submitted by **Robert Andrew Chai** in partial fulfillment of the requirements for the degree of **Master of Science.**

[illegible]
Supervisor

David [illegible]

[illegible]

Joseph Culberson

Date 31/08/88

5

ABSTRACT

Learning classifier systems are a relatively new area of AI research. Designed with domain independence in mind, these systems can be applied to a variety of learning environments. Their ability to *adapt* to their environment makes them useful for dealing with dynamic environments.

Essentially, these systems represent knowledge in the form of rules (classifiers). Environmental feedback enables a system to evaluate the usefulness of a rule. Useless rules are discarded and replaced by new rules (survival of the fittest) which are discovered through several rule discovery operators. Some of these operators mimic genetic operations in that they *mate* or *mutate* existing rules in order to produce new rules. The result of all this is a system which "evolves" a knowledge base that enables it to perform well in the environment to which it is applied.

This thesis discusses the author's implementation of the learning classifier system LCS. In addition, experimental results and enhancements to this system are also described. The result of this research is a system which is capable of learning from its environment in order to adapt and perform well within that environment.

Acknowledgements

I would like to thank my supervisor, Jonathan Schaeffer, for all his guidance and constructive criticism without which this thesis would not be possible. Also my parents, Lloyd and Dorothy Chai, for their support throughout this endeavor. And finally my peers, Keith Fenske, Dale Schuurmans, and Lingyan Shu, for all the interesting discussions I had with them.

Table of Contents

List of Figures

Chapter 1

Introduction

Many real world problems deal with environments which change over time. Systems which interact with such environments experience new "stimuli" (input), from time to time, with which they must deal. An example of such an environment is speech recognition in which a system might receive the sounds of a new voice; or, object recognition in which a system might "see" an object from a new angle. The success of such systems in dealing with their environments depends upon their ability to adapt to the new input. The system should try to process the new input as best it can and, if incorrectly processed, should have some sort of feedback and correction mechanism which will allow the system to adapt. Limitations in the ability of a system to adapt to new input restricts the variety of situations that it can handle. This problem is faced by many expert systems.

Expert systems have demonstrated success in a variety of areas. The problem with these systems, however, is that they tend to breakdown when they are applied to areas which are slightly different from those for which they were designed. Part of the reason for this is that many expert systems do not possess any correction mechanism that allows the system to adapt. Most expert systems consist of a rule base constructed by humans. If a system is applied to a slightly different area resulting in poor performance, then human intervention is required to modify the rule base in order to give the desired performance. This modification may involve the addition and removal of rules from the rule base. In this case, the correction mechanism of the system is provided by humans. What is needed is the ability for the system to assess its own performance and to modify its own rule base accordingly. In this way, such a system would be able to deal with the perpetual novelty of situations arising in a changing environment. One solution to this problem is to use induction.

Induction, in this context, involves the construction of categories that organize environmental input usefully. These categories allow input to be classified according to response. Inputs which require similar responses will be grouped in the same category. These categories should also be broad enough collectively to encompass the range of inputs likely to be encountered. In this way, categories provide a mechanism with which to combat the perpetual novelty of situations. A new input that arises can be classified into a category and an appropriate response can be determined for it. Of course, the key to all of this is the development of useful categories. This involves establishing some sort of environmental feedback mechanism which will allow the usefulness of a category to be determined. Also, some mechanism for discovering categories is required so that new categories can be developed. One such framework which incorporates these ideas is the *learning classifier system* [Hol86].

A learning classifier system is a rule-based system somewhat akin to an expert system. Categories are represented implicitly through the rules of the system. Thus, the task of developing categories now becomes the task of developing the rule base. Development of the rule base is accomplished through the use of the *bucket-brigade algorithm* and *genetic algorithms*. The bucket-brigade algorithm facilities the feedback mechanism in identifying useful rules in the rule base. Genetic algorithms provide the mechanism by which new rules can be discovered. As the name implies, genetic algorithms mimic genetic operations in biological organisms in order to develop new rules. In this respect, new rules might be produced from the "mating" of existing rules, or possibly by the "mutation" of existing rules. This, together with a policy that only the "fittest" rules survive, and less "fit" rules are replaced by newly discovered rules, results in a system whose knowledge base "evolves" to suit the environment. The system "learns" how to perform well in its environment. Thus, learning classifier systems are capable of adapting to gradual changes in their environment.

Learning classifier systems provide a general framework which is applicable to a variety of domains: all the internal mechanisms involved in the system are independent of the environment to which the system is connected. Thus, applying a learning classifier system to a new domain mostly involves establishing the interface between the system and the domain environment.

Much work on the development and application of learning classifier systems has been accomplished over recent years. One system, developed by Rick Riolo at the University of Michigan [Rio86a,Rio86b], involves the task of learning to predict letter sequences. Given a series of letters, the system must predict which letter will come next. Some of the features present in this system were found to be useful and the incorporated into the system that this author built. Other systems have been developed for applications such as: gas pipeline control [Gol85], visual recognition [Eng85], job shop scheduling [Dav85,HiL87], symbolic layout compaction [Fou85], tic-tac-toe [Sla86], and simple program generation [Cra85]. While the gas pipeline control and visual recognition systems have been fairly successful, success with the other systems has been limited. Further work remains to be done with those systems.

This thesis discusses LCS, the author's implementation of a learning classifier system. The goal of this research was to construct a test-bed within which to explore and investigate learning classifier systems. An initial system was implemented based on the system described in [Hol86]. Because of the lack of detail in the literature, this involved various design and implementation decisions by the author. Experiments were performed on this initial system by applying it to the domain of tic-tac-toe. Results from these experiments indicated that enhancements to the system were necessary in order to get the desired performance. With these enhancements in place, performance of the system improved dramatically giving much better results.

This thesis proceeds by first discus[illegible] learning cl[illegible]fier systems from a high level point of view in chapter 2. At the end of this chapter, the reader should have a general overall understanding of how a learning classifier system works but will not know all the finer level details required to actually construct such a system. These details are provided in chapter 3 which describes the implementation of LCS prior to the enhancements. Chapter 4 describes the enhancements made to the system as well as the justifications for those enhancements. Chapter 5 describes the procedures that one must perform in order to apply LCS to a new domain. In addition, a small detailed trace of the system (as it is applied to a trivial problem) is provided in the hopes of clearing up any questions the reader might have about the workings of the system. Chapter 6 describes the experimental results of applying the enhanced system to tic-tac-toe. Finally, the conclusion is given in chapter 7.

Chapter 2

Learning Classifier Systems

2.1. Introduction

A learning classifier system is a system within which learning can be achieved. Such a system usually displays an adaptive style of learning. This means that when it is connected to an environment (application), it learns by adapting itself to the environment in such a way as to perform better within that environment. The measurement of performance within the environment is usually related to the achievement of some goal. Thus, the closer a system gets to achieving the goal, the better its performance is. By allowing a system sufficient time to run, the system should eventually learn how to achieve the goal.

Learning in learning classifier systems can be studied by examining three issues: knowledge representation, evaluation, and discovery. The representation of knowledge refers to the scheme or format used to record the knowledge. The evaluation of knowledge refers to the methods used to evaluate the value (usefulness) of the knowledge. The discovery of knowledge refers to the methods used to discover new knowledge. The remainder of this chapter will examine each of these issues in turn; thus providing a description of the workings of a learning classifier system.

2.2. Classifier Systems

The term classifier system refers to the learning classifier system without the bucket-brigade algorithm and genetic algorithms as it is these algorithms which provide the ability to learn. A classifier system provides a format for representing knowledge. Essentially, it is a type of production system. As in a production system, a classifier system consist of classifiers, which are analogous to production rules, and messages, which are analogous to working memory elements.

Messages are used to record information related to a particular run (execution) of the classifier system. Through messages, information is passed back and forth between the system and the environment. Messages passed from the environment to the system (input messages) usually convey information about the state of the environment, while messages passed from the system to the environment (output messages) usually indicate actions to be performed in the environment. Thus, messages provide the communication link between the system and the environment. Input messages come into the system, the system decides what actions should be performed, and then relays these actions to the environment through output messages. These steps are performed repeatedly allowing the system to alter the environment. When the system performs an action (through an output message) it receives feedback as to how desirable the action was in changing the environment. This feedback can guide the system in determining its next action.

Messages also play a role internal to the system. After retrieving input messages, the system must decide what actions must be performed. This decision process may involve the development of intermediate concepts that are recorded in the form of messages. These intermediate concepts may in turn lead to the development of other intermediate concepts, and so on, before the system arrives at the actions to be performed.

Messages are represented as fixed length strings over the alphabet {0,1}. The length of a message varies from one application to another, but remains constant within an application. A message can be interpreted by breaking up its string into substrings and then interpreting each of the substrings. This breakdown and interpretation will vary from one application to another. For example, the message 1110 might breakdown into the substrings 1,11,10 which might be interpreted as "there is a block ahead (1), it is three units away (11), and it is red (10)" for some block's world application.

All messages are contained in a global structure called a message list. Input messages are placed on the message list by the environment and are subsequently processed by the system. Output messages are placed on the message list by the system and are subsequently inter[illegible]ted by the environment. Messages developed between the input and output phases of the system are also placed on the message list. The message list corresponds to working memory in a production system.

In addition to messages, classifier systems also include classifiers. The set of classifiers in a system embodies all the knowledge that system has about a particular application domain. As was mentioned prev[illegible]usly, classifiers correspond to rules in a production system. Like a production system rule, a classifier consists of a condition part and an action part. When there are messages on the message list that satisfy the con[illegible] part of a classifier, that classifier is capable of "firing". Firing results in the posting of a message, derived from the action part of the classifier, on the message list.

The condition part of a classifier specifies a set of messages to which the classifier can respond. This specification is denoted using fixed length strings over the alphabet {0,1,#}. The length of these strings depends upon the application, but the length is constant within an application and is the same as the length of a message. A message is said to satisfy a condition string when each symbol in the message string matches the symbol in the condition string that is in the same position. A "0" in the condition string will match only the symbol "0" in the message string. A "1" in the condition string will match only the symbol "1" in the message string. A "#" in the condition string acts as a don't care symbol and will match either the symbol "0" or "1" in the message string. Allowing the "#" symbol in the condition string enables one condition string to specify a set of satisfying messages. For example, the condition string 110## will match any of the following messages: 11000, 11001, 11010, and 11011. A condition string may also be prefixed with a minus sign ("-") which indicates that the condition string is satisfied if there is no message on the message list that matches the condition string. So, the condition string -110## will be satisfied if the messages 11000, 11001, 11010, and 11011 do not appear on the message list.

The condition part of a classifier may contain one or more condition strings. The condition part of a classifier is said to be satisfied if each of the condition strings is satisfied. When a condition part is satisfied, there is said to be a match between the condition part and the satisfying messages. The ability to have multiple condition strings in a condition part along with the ability to negate a condition string enables a group of classifiers to specify conditions over an arbitrary set of messages. An AND-condition can be achieved using a classifier whose condition part has many condition strings. An OR-condition can be achieved using a group of classifiers whose actions are all the same. A NOT-condition can be achieved using negated condition strings.

Once the condition part of a classifier is satisfied, it is capable of firing, producing a message from its action part. The action part of a classifier is denoted by a fixed length string over the alphabet {0,1,#}. As with a condition string, the length of an action string depends upon the application but is constant within an application and is the same as the length of a message. Unlike a condition part, an action part consists of only one action string. The message that is produced from an action part depends upon the symbols in the action string. A particular symbol in the resulting message is determined from the symbol that appears in the same position in the action string. A "0" in the action string produces the symbol "0" in the message string. A "1" in the action string produces the symbol "1" in the message string. A "#" in the action string acts as a pass through symbol, producing in the message string the same symbol as that in the same position in the message that satisfied the *first* condition string of the classifier. Thus, if a classifier had the action part 10#1# and the message 00110 satisfied the first condition string of that classifier, then the firing of that classifier would result in the message 10110.

Using a message list and a set of classifiers, the basic execution cycle of a classifier system proceeds as follows [Hol86]:

1. Place all messages from the input interface on the current message list.
2. Compare all messages to all conditions and record all matches.
3. For each match, generate a message for the new message list.
4. Replace the current message list by the new message list.
5. Process the new message list through the output interface to produce system output.
6. Return to step 1.

This execution cycle is capable of supporting parallelism since classifiers may fire in parallel. The use of parallelism can sometimes offer significant improvement in speed, especially when dealing with a classifier system that has a large set of classifiers.

A useful feature of classifiers is the ability of classifiers to couple. A classifier C2 is said to be coupled to a classifier C1 if some condition string of C2 is satisfied by the message(s) produced by C1. Coupling of classifiers occurs across several iterations of the basic execution cycle; so, in this case, C1 would fire in one iteration thus setting up C2 to fire in the next iteration. The ability of classifiers to couple enables control and sequencing of classifier firing. Explicit coupling can be achieved through the use of tags. For example, a classifier might have the condition 110##...##. In this condition, the substring 110 is called a tag. Any message with the prefix 110 will satisfy that condition. Thus, any classifiers that produce messages with the prefix 110 will be coupled to that classifier. One of the obvious uses of tags is for distinguishing input, internal, and output messages. By using different tags for each of these kinds of messages, one can create classifiers that respond only to a certain kind of message. The use of tags for distinguishing these different kinds of messages enables the realization of an arbitrary finite state machine [Hol86]. Keep in mind that tags are logical concepts that are only meani l to the environment. The system is unaware of the existence of tags and merely performs the matching and firing described previously.

2.3. Feedback and Bucket-Brigade Algorithms

Classifiers provide the means for representing knowledg . A classifier is a unit of knowledge about a particular application domain. The set of classifiers in a classifier system represents all the knowledge that the system has about a particular application domain. Some classifiers in a system represent more valuable or useful knowledge than others. Thus, there needs to be some way to evaluate and rank the classifiers in a system in order to identify the useful knowledge and weed out the useless knowledge. This knowledge evaluation is achieved in classifier systems by assigning a *strength* to each classifier. The strength of a classifier is a number which indicates how valuable that classifier is. Higher strength (stronger) classifiers are more valuable than lower strength (weaker) classifiers.

When a classifier is first created, it is usually given some predetermined initial strength since the true value of that classifier is not yet known. As that classifier gets used (fires), its strength is adjusted according to how useful or valuable it proves to be. This adjustment is achieved using the bucket-brigade algorithm (to be discussed shortly) and environmental feedback.

During the course of execution, many classifiers may be involved in achieving some goal in the environment. These classifiers may work in combination, possibly through coupling, in order to achieve that goal. The classifiers at the end of the combination or chain will be rewarded directly from the environment. Whenever the goal is achieved, the environment issues a *payoff* to each of the active classifiers. The active classifiers are just the classifiers that placed or posted messages on the message list at the time the goal was achieved. This payoff will result in an increase in the strengths of the active classifiers. However, this reward only affects the classifiers at the end of the chain; this payoff must somehow filter down the chain so as to reward those earlier "stage setting" classifiers that made it possible for the goal to be achieved. The bucket-brigade algorithm addresses this problem.

The bucket-brigade algorithm involves a modification to the basic execution cycle. Instead of allowing all satisfied classifiers to fire, classifiers must now compete for the chance to fire. Competition is introduced by having all satisfied classifiers place bids. The highest bidding classifiers are the ones most likely to fire. The bid [Hol86] of a classifier is calculated by determining the product of: a constant (e.g. 1/8 or 1/16), the strength of the classifier, and the *specificity* of the classifier. The strength of a classifier gives a measure of that classifier's past usefulness. The specificity of a classifier is calculated by dividing the number of non-#'s in its condition part by the length of its condition part. This gives a measure of how relevant that classifier is to the current situation. Classifiers with high specificity are more relevant to the current situation and consequently should outbid those of equal strength but with lower specificity. When a classifier is fired, its strength is decreased by the amount of its bid — it pays for the privilege to post its message (to fire). Alternately, the classifiers that posted the messages that enabled that classifier to fire will have their strengths increased. The amount of the increase is usually just the amount of the bid divided by the number of classifiers involved.

As one can see, the strength of a classifier acts as a kind of capital. A classifier "pays" those classifiers that enabled it to fire and receives "pay" from those classifiers that it enables to fire. Classifiers that participate in chains that lead to goal attainment will profit in this "economy"; while those that participate in "fruitless" chains will ultimately lose in this "economy". Thus the bucket-brigade algorithm is capable of rewarding those classifiers which prove to be valuable while punishing those which do not. Note that it is possible to have classifiers that, in order to fire, require messages that will never be generated. As it stands, the strengths of these useless classifiers will remain unchanged from their initial values as these classifiers will never fire. The solution to this problem is to introduce taxation. This is discussed in chapter 4.

2.4. Genetic Algorithms

Classifier systems with the bucket-brigade algorithm are capable of representing and evaluating knowledge about a particular application domain. When a classifier system first starts out, it is usually provided with a set of randomly generated classifiers that all have the same strength. As the basic execution cycle is performed, these classifiers are "exercised" (used) and their strengths are updated. Eventually, the system should reach a state where the "good" classifiers have higher strengths than the "bad" classifiers. At this stage, the bad classifiers are removed from the system and replaced by an equal number of new classifiers which are generated (discovered) using genetic algorithms. After these new classifiers have replaced the bad ones, the system resumes execution of the basic execution cycle thus "exercising" the new classifiers and updating their strengths -- and so repeats the process. This repeated process of discovering knowledge and then evaluating it, enables the system to learn.

Genetic algorithms are the tools used to discover new knowledge in classifier systems. Genetic algorithms are modeled after the biological processes that are responsible for biological evolution. In biological evolution, descendants of an evolutionary chain evolve by adapting to their environment. It is this adaptive behavior that is desired in classifier systems, the idea being that instead of the system trying to evolve biological entities, it tries to evolve knowledge (classifiers). This adaptive behavior is achieved by using genetic algorithms.

Genetic algorithms generate new classifiers by executing the following cycle:

1. Select (pairs of) classifiers according to strength -- stronger classifiers being more likely to be selected.

2. Apply genetic operators to the selected classifiers to create new offspring classifiers.

3. Replace the weakest classifiers in the system with the new offspring.

This cycle is repeated until the desired number of new classifiers has been generated.

The genetic operators used to produce the offspring work by performing operations on the condition and action strings of the selected classifiers. These operations do not alter the selected classifiers in any way, but rather are used to construct the new offspring from the selected classifiers. Three of the more commonly used genetic operators are: mutation, inversion, and crossover. Mutation is performed using a single string, and produces a string which is essentially identical but with some randomly changed symbols at randomly selected positions. Inversion is performed using a single string, and produces a string which is essentially identical but with some randomly selected substring being reversed. Crossover is performed using two strings, and produces two strings which are the same as the two originals but with some randomly selected substrings being exchanged between the two. Substrings which are exchanged are usually substrings which occupy the same positions. So, if some condition or action string was 110#00#1, then mutation might produce the string 110000##, and inversion might produce the string 1#0100#1. If two condition or action strings were 1101#1## and #0#0110#, then crossover might cross over between the third and fourth positions resulting in the strings 1100110# and #0#1#1##. The new strings that are produced from the genetic operators become part of the new offspring classifiers. Mutation and inversion both use one selected classifier producing one new offspring classifier. Crossover uses two selected classifiers and produces two new offspring classifiers.

The success of genetic algorithms in discovering valuable classifiers lies in the property of strong classifiers to contain some building blocks (substrings) which aid in the achievement of the environmental goals. Genetic operators such as mutation and inversion try to discover new useful building blocks, while operators such as crossover seek new ways of combining building blocks in order to yield more valuable classifiers. When crossover is performed on strong classifiers, it tends to be reasonably successful in producing "good" classifiers. Consequently, crossover tends to be the operator most frequently used to produce new classifiers. Mutation seems to be less successful in this respect and inversion even more-so. Thus, these operators tend to be used less frequently. By replacing the weak classifiers, genetic algorithms promote the policy of "survival of the fittest" -- only the strong classifiers survive; the weak ones die. As the set of classifiers evolve, the population of classifiers should contain more strong classifiers and hence more useful building blocks. This will result in an improvement in overall performance of the system in the environment.

2.5. Comments

Learning classifier systems provide an adaptive style of learning which is applicable to a large variety of domains. Such systems have been applied to areas such as: function optimization, visual recognition, job shop scheduling, and simple program generation. The success of learning classifier systems is largely because such systems are capable of exploiting biases in a domain which lead to improved performance within that domain. This occurs because new classifiers that are generated are similar to ones that have proved valuable in the past. Through the policy of "survival of the fittest", these systems dynamically develop a body of knowledge about a particular application domain. This "evolution" of knowledge enables these systems to learn.

Chapter 3

Implementation Description

3.1. Introduction

The system that is currently implemented is slightly different from that described in the previous chapter. Development initially involved creating a system which was as close as possible to that described in the literature (chapter 2). The paper which formed the basis for this implementation was that written by John Holland [Hol86]. This paper along with some additional literature [HHN86] provided a general description of the implementation. Some of the finer details, however, were not discussed in the literature and consequently involved some design decisions on the part of this author. These design decisions will be discussed as they arise.

Once this initial system had been developed, experiments were performed by applying this system to the domain of tic-tac-toe. The results of these experiments indicated that enhancements to the system were necessary in order to give the desired performance. These enhancements and their implementation details are discussed in the next chapter.

This chapter describes the implementation of the initial system — the system described in chapter 2. This implementation was done in C on a VAX minicomputer and involved approximately 4000 lines of code. The description of this implementation proceeds by first describing the classifier system with bucket-brigade followed by a description of the genetic operators.

3.2. Classifier System with Bucket-Brigade

This section describes the implementation of the classifier system with the bucket-brigade algorithm. The two major data structures of the implementation, namely the message and classifier data structures, are described first. Figure 3.1 provides a pictorial description of these data structures as well as supporting data structures, showing the interrelations between them. Details of this figure are given in the sections that follow. Following the data structure discussion is a description of the implementation of steps two (compare all messages to all conditions and record all matches), three (for each match generate a message for the new message list), and four (replace the current message list by the new message list) of the basic execution cycle described in chapter 2. Steps one (place all messages from the input interface on the current message list) and five (process the new message list through the output interface to produce system output) involve interacting with the environment and are discussed in chapter 5. Finally, this section describes the implementation of the environmental feedback mechanism. Keep in mind that this section discusses only the implementation details and does not address the means by which a user is to access this code. That is discussed in chapter 5.

3.2.1. Messages

Messages are maintained in the msgType structure which has the following declaration:

```
struct msgType {
    long id;
    int bid;
    struct bitStr *msg;
    struct clfType *clf,
    struct msgType *nxtMsg;
    };
```

The id member contains the identification number of the message. Message identification numbers start at zero and progress up to MAX_LONG - 1 at which point

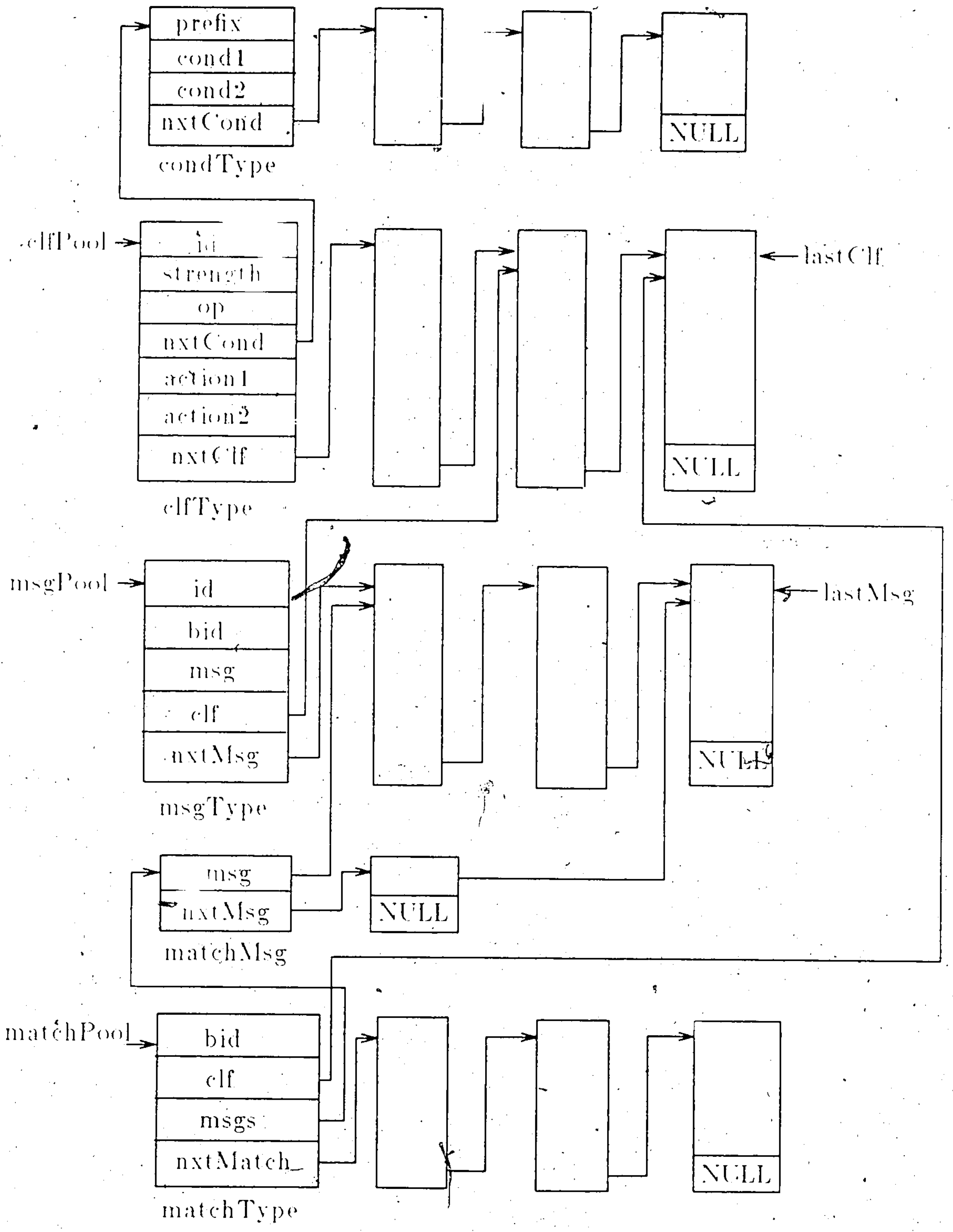


Figure 3.1. Data structures used by the classifier system.

they restart at zero (MAX_LONG is a constant which is currently set to 2147000000, approximately 2^{31}). These numbers aid in keeping track of messages.

The bid member contains the value of the bid involved in posting the message, and the clf member is a pointer to the classifier that produced the message. When a classifier is capable of firing, it places a bid which competes against bids from other classifiers which are also capable of firing. Should that classifier fire, it is the value of its bid that is stored in the bid member, and a pointer to that classifier that is stored in the clf member of the resulting message. When the message comes from the environment and therefore has no corresponding classifier that produced it, the bid and clf members are set to zero and NULL respectively.

The msg member is a pointer to the contents of the message. A message is maintained in a bitStr structure which has the following declaration:

```
struct bitStr {
   long str[MAX_MSG_SIZE/(8*sizeof(long))];
   };
```

The MAX_MSG_SIZE constant determines how large the array is. This constant indicates, in bits, the largest message allowed for any application. The sizeof(long) function is a C function that returns the number of bytes in a long integer. For the VAX machines, on which this system was developed, this number is four bytes. Thus long integers are 32 (8*sizeof(long)) bits long resulting in bit strings (strings of 1's and 0's) being stored in multiples of 32 bits, "left justified", with any extra bits being set to zero. The length of bit strings are determined by the two variables msgSize and msgBlkSize. msgSize indicates the number of bits in a message. msgBlkSize indicates the number of long integers required to store a message. This is just the minimum number of long integers required to store a bit string that is msgSize bits long. Thus if msgSize is 40 then msgBlkSize will be 2 ($\lceil 40/32 \rceil$). Memory is allocated for bit strings by issuing the call malloc(msgBlkSize*sizeof(long)). The length of this resulting bit string is usually less than that of the largest message allowed. This, however, does not present a problem in using the bitStr structure as this structure is used as a template over the allocated memory so as to be able to access that memory. A bit string which is shorter than the str array just means that only the first msgBlkSize elements are allowed to be accessed. The bitStr structure along with the two corresponding variables are also used for condition and action strings since these strings are the same length as messages. This will be discussed in more detail in the section on the classifier data structure.

Finally, the nxtMsg member is a pointer to the next message in the message pool. The message pool is a linked list of messages (msgType structures). The variable msgPool points to the first message in the pool and the variable lastMsg points to the last message. The nxtMsg member is NULL for the last message in the message pool. A pictorial description of the message pool is given in figure 3.1.

3.2.2. Classifiers

Classifiers are maintained in the clfType structure which has the following declaration:

```
struct clfType {
  long id;
  int strength;
  char op;
  struct condType *nxtCond;
  struct bitStr  action1;
  struct bitStr  action2,
  struct clfType *nxtClf;
  };
```

The id member serves a purpose similar to the equivalent in msgType. The strength member contains the strength value associated with a classifier. The op member is a character that indicates which genetic operation created the classifier. Classifiers can be created from the various genetic operations such as crossover, mutation, and so on. Knowing which operator created a classifier can aid in tracking and debugging the genetic operators. The various values that this member can assume will be discussed in the section on the genetic operations.

The condition part of a classifier is maintained in the `nxtCond` member. This member is a pointer to a linked list of condition strings which make up the condition part of the classifier. Each condition string is maintained in a `condType` structure, which has the declaration:

```
struct condType {
   int prefix;
   struct bitStr *cond1;
   struct bitStr *cond2;
   struct condType *nxtCond;
   };
```

The `prefix` member is an integer which indicates whether or not the condition string is negated. It assumes one of the two constant values `POSPFX` (if the condition string is un-negated) or `NEGPFX` (if the condition string is negated).

The actual condition string itself is maintained in the two members `cond1` and `cond2` each of which contains a bit string. A condition string is encoded into these two bit strings by encoding each symbol into the corresponding position in each bit string. The encoding scheme is as follows:

Symbol	cond1	cond2
"0"	0	0
"1"	0	1
"#"	1	1

Thus, the condition string 1101##0# would be encoded as 00001101 for `cond1` and 11011101 for `cond2`. As with messages, the bit strings are stored in multiples of 32 bits and any extra bits are set to zero. This encoding scheme allows for a simple test to determine if a message satisfies a condition string:

result = cond2 XOR (cond1 OR message)

If the result is a bit string which is all zeros then the message satisfies the condition string, otherwise it does not. So, for example, if the message 11011000 was compared against the condition string previously given, the result would be:

```
result = 11011101 XOR (00001101 OR 11011000)
       = 11011101 XOR 11011101
       = 00000000
```

which indicates that the message satisfies the condition string.

Finally, the nxtCond member is a pointer to the next condition string in the list. This member has the value NULL for the last condition string.

Turning back to the clfType structure, the action1 and action2 members each contain bit strings which specify the action string. This string is encoded in a fashion similar to the condition string. The encoding scheme in this case is:

Symbol	action1	action2
"0"	0	0
"1"	0	1
"#"	1	0

So, the action string 00#1##11 would be encoded as 00101100 for action1 and 00010011 for action2. Again, these bit strings are stored in multiples of 32 bits with any extra bits being set to zero. This encoding scheme facilitates a simple method for determining the message resulting from firing a classifier:

resulting_message = action2 OR (action1 AND message)

where message is the message that satisfied the first condition string of the classifier. Thus, if the satisfying message was 1 100001 and the action string was that given previously, then the resulting message would be:

```
resulting_message = 00010011 OR (00101100 AND 10100001)
                  = 00010011 OR 00100000
                  = 00110011
```

which is as expected.

Finally, the nxtClf member is a pointer to the next classifier in the classifier pool. The classifier pool is a linked list of classifiers (clfType structures). The variable clfPool points to the first classifier in the pool and the variable lastClf points to the last classifier. The nxtClf member has the value NULL for the last classifier in the pool. A pictorial description of the classifier data structure and the classifier pool is given in figure 3.1.

3.2.3. Basic Execution Cycle

This section describes the implementation of steps two through four of the basic execution cycle presented in chapter 2. These steps form the heart of this cycle as they are the steps which produce the new message list from the current one. All these steps are performed by the function executeOneCycle. This function essentially consists of three function calls to the functions getMatches, detMatchesToFire, and createNewMsgPool.

3.2.3.1. Constructing Matches

Step two of the basic execution cycle involves comparing all messages to all conditions and recording all matches. This is accomplished by the function getMatches. The purpose of this function is to build up a match pool (matchPool) which is a linked list of matches (matchType structures).

The matchType structure contains all the information related to a match between a particular classifier and a particular set of messages. Its declaration is as follows:

```
struct matchType {
   int bid;
   struct clfType *clf;
   struct matchMsg *msgs;
   struct matchType *nxtMatch;
   };
```

The bid member contains the bid for the match; the clf member indicates the classifier involved in the match; the msgs member points to the list of messages that satisfy the condition part of the classifier; and the nxtMatch member points to the next match in the match pool. A pictorial description of the match pool is given in figure 3.1.

3.2.3.2. Finding Matches

Construction of the match pool first involves determining a possible match between a message and the first condition string of a classifier. This means that each message is compared to the first condition string of each classifier to determine if the message satisfies the condition string (note that the first condition string of a classifier must be un-negated as it is the message that satisfies this string that is used in the firing of that classifier). For each comparison that results in satisfiability, a possible match exists and the remaining condition strings in the classifier are examined to determine if they too can be satisfied. This involves constructing, for each remaining condition string, a list of messages which satisfy that particular condition string. This message list should be empty for negated condition strings and non-empty for un-negated condition strings. If this is the case for all the remaining condition strings then the condition part of the classifier is satisfiable and a match can be constructed. Actually, several matches can be constructed and are done so through the function buildMatches.

Since for each un-negated condition string there is a list of messages which satisfy that condition string (in the case of the first condition string this list consists of only one message), a set of matches can be constructed by constructing different sets of messages which satisfy the condition part of the classifier. Each set is constructed by selecting one message from the list of messages for each un-negated condition string. By selecting different combinations of messages, different sets of satisfying messages can be formed. buildMatches constructs all possible combinations resulting in the addition of several matches to the match pool.

3.2.3.3. Bid Calculation

A bid must be calculated for each match in the match pool. This is performed by the function calcBid which is invoked for each match at the time the match is being constructed. The value of the bid is calculated as follows:

bid = bid_coefficient * strength * specificity

The bid_coefficient is contained in the variable bidCoeff, the strength is the strength of the classifier involved in the match, and the specificity is the specificity of the classifier involved in the match.

Specificity is calculated by dividing the total number of non-#'s in all the un-negated condition strings by the product of the number of bits in a message (msgSize) and the maximum number of condition strings allowed in a condition part (maxNumConds). This calculation permits the comparison of specificity amongst classifiers whose condition parts have differing numbers of condition strings. With this calculation, classifiers with more un-negated condition strings will likely have higher specificities. This stands to reason as a classifier with more un-negated condition strings will only be applicable to more specialized situations and thus should have a higher specificity.

If the value of the bid turns out to be negative (because the classifier strength is negative) then a bid of zero is used.

3.2.3.4. Determining Which Matches Should Fire

Because of the bidding competition between the matches, it may be the case that not all the matches should fire. The function that determines which matches should fire is detMatchesToFire. This function reduces the size of the match pool by removing those matches which shouldn't fire. The matches to be removed depend upon two variables: clfPostLvl and fireProb. clfPostLvl indicates the maximum number of matches that are allowed to fire. A value of zero indicates that an infinite number of matches can fire. This function proceeds by first establishing a new match pool consisting of the first clfPostLvl matches in the old match pool. If the old match pool has less than clfPostLvl matches then all the matches are used. Of the remaining matches in the old match pool, approximately fireProb percent of them are examined to determine if their bids are ~~higher~~ than the lowest bid in the new match pool. If this is the case for a particular match, then that match will be inserted in the new match pool displacing the lowest bidding match. The actual method used to achieve a probabilistic examination of the remaining matches is simply to generate a random number for each of the remaining matches; if this number is less than or equal to fireProb then the corresponding match is examined. When the old match pool has been exhausted, the new match pool replaces it, resulting in a reduced match pool containing only those matches that should fire.

3.2.3.5. Creating The New Message Pool

Creation of the new message pool is performed by the function createNewMsgPool. This function fires all the matches in the match pool creating a new message pool which replaces the old message pool. A match is fired by first creating a new message from the classifier and the message, which satisfies the first condition string of that classifier, stored in the match. This new message is placed in the new message pool. Following this, the strength of the firing classifier is decreased by the amount of the bid. Then, the "supplying" classifiers' strengths (the classifiers who posted the messages that satisfied the condition strings of the firing classifier) are updated. This involves incrementing the strength of each supplying classifier by the amount of the bid divided by the number of supplying classifiers. After all the matches have fired, the old message pool is destroyed and the new message pool replaces it.

3.2.4. Environmental Feedback

Environmental feedback involves issuing payoff (numerical feedback) to classifiers in the system. This is accomplished by calling the function `payoff`. The payoff issued may either be a positive or negative integer depending upon whether the system is being rewarded or punished for its actions.

When issuing payoff, the group of classifiers to receive the payoff must be identified. Classifier groups are identified using group numbers (0 and up). The group number 0 is a special group which consists of all the active classifiers (classifiers which have posted a message to the current message list). Other group numbers refer to groups which have been constructed by the application. Construction of groups is performed through the function `addToPOGrp`. When called, this function adds all the currently active classifiers to a specified group number. If the group did not previously exist, it is created, and the active classifiers are placed in it. Otherwise, the active classifiers are just added to the existing group. The number of groups that can be created is limited by the amount of available memory.

Group information is maintained as a linked list of groups where each group is a linked list of classifiers. The data structures involved in this are:

```
struct payoffClf {
   struct clfType *clf;
   struct payoffClf *nxtClf;
   };


struct payoffGrp {
   int num;
   struct payoffClf *clfs;
   struct payoffGrp *nxtGrp;
   };
```

The payoffGrp structure contains information about a group, namely its number (num) and the classifiers in the group (clfs). The payoffClf structure is used to identify the classifiers (clf) in a group. Group information can be accessed through the variable payoffGrps which is a pointer to the linked list of groups.

In addition to constructing a group, an application can also destroy a group through the function rmvPOGrp. Destroying a group disassociates the classifiers in the group from that group, and frees up the group number so that it can be used again in the construction of another group. Only group numbers greater than 0 can be destroyed.

With the ability to construct and destroy groups, an application has the capability to group together the sets of active classifiers, that appear over the iterations of the basic execution cycle, in any fashion desired. This way, various groups of classifiers can be identified for different amounts of payoff. When payoff is issued to a group, the amount of the payoff is divided equally amongst the classifiers in that group. This results in the strengths of those classifiers being either increased or decreased depending upon whether the payoff is positive or negative.

3.3. Producing A New Generation

As expected, execution of this system proceeds in two stages. The first involves the evaluation of the current knowledge as the basic execution cycle is performed and feedback is received from the environment (application). After performing the basic execution cycle for some number of iterations, it is halted and the second stage is performed. This involves the discovery of new knowledge as the next generation of classifiers is produced. This two stage process, which is known as a generation, then repeats. Execution of the system usually proceeds for some specified number of generations. The number of iterations of the basic execution cycle that is performed in a generation is determined by the application. Usually, the application might perform a fixed number of iterations per generation or it might perform the iterations until a certain number of "weak" classifiers have been identified. At any rate, the criteria for determining the number of iterations to be performed rests with the application code.

This section describes the details involved in producing a new generation. The function which accomplishes this is `produceNewGen`. Briefly, this process involves first identifying the weak classifiers which are to be replaced. Then, the new classifiers are generated using genetic operators. Finally, the weak classifiers are replaced by the new ones. This completes the production of a new generation. Note that if there are no weak classifiers at the time `produceNewGen` is called then no new classifiers are generated.

3.3.1. Identifying Weak Classifiers

The first step in producing a new generation is to identify the weak classifiers in the system. This raises the obvious question concerning the definition of a weak classifier. In this system, it was decided that a weak classifier is any classifier whose strength is below a certain value. This threshold value is stored in the variable `rplcThres` which is set by the application at the start of the execution.

When it is time to produce a new generation, the classifier pool is first sorted according to strength. This is done using a quick sort and involves first loading the array `sortedClfPool` with the classifier pool and then sorting that array. The elements of this array are of the type `clfMarker` which has the following declaration:

```
struct clfMarker {
   struct clfType *clf;
   int use;
   };
```

In this structure, the `clf` member points to a classifier in the classifier pool. The `use` member is used by the genetic operators and hence shall be discussed in that section.

After the classifier pool has been sorted, the weak classifiers can easily be identified. If there are any weak classifiers in the classifier pool then they will be replaced by new classifiers produced from the genetic operators. The workings of these operators are discussed in the next section.

3.3.2. Genetic Operators

The genetic operators that are used in this system are mutation and crossover. Inversion was not used because it tends to be one of the weaker operators, having less success at producing useful knowledge. Both mutation and crossover work by selecting classifiers from the classifier pool and then performing some genetic operation on them to produce new classifiers. These new classifiers are then placed, temporarily, in a list of new classifiers. This avoids having new classifiers being produced from new classifiers, a problem which could result if the new classifiers were placed directly into the classifier pool. After all the new classifiers have been g[illegible]ated, the list of new classifiers is then put into the classifier pool replacing the weak classifiers.

This section describes the implementation of the mutation and crossover operators. This description proceeds by first discussing some basic data structures that are required by the operators. Following this, a description of the operators themselves is given.

3.3.2.1. Data Structures

Often it is the case that messages are the composition of several fields where each field is more than one bit long. It is also common for a field not to use all the possible states that can be represented by its corresponding bits. For example, a message might consist of two fields, speed and direction. There might be three different possible speeds (speed1, speed2, and speed3) and three different possible directions (left, right, and forward). Encoding each of these fields requires two bits. One possible representation of these fields is:

Speed	Bits	Dir.	Bits'
speed1	"01"	left	"10"
speed2	"10"	right	"01"
speed3	"11"	forward	"11"

Thus, the message "1011" would indicate traveling at speed2 in the forward direction. Note that the bits "00" are not used by either field. Any message produced by the system in this application should not have the bits "00" present in any field. This restriction has a direct effect upon the types of classifiers the system is allowed to produce. For instance, the system is allowed to produce the classifier

##11 / 0111

but not the classifier

##11 / 0011.

It is also possible that the fields for the condition strings of a classifier might be different than that for the action string. For example, the first bit of a message might indicate whether it is an input message ("1") or an output message ("0"). This means that all the condition strings should start with a "1" and all the action strings should start with a "0". The system should not produce a classifier which has a condition string that starts with a "0", or an action string that starts with a "1".

In order to prevent these problems from arising, the system maintains information about the position and permissible values for the various fields in the condition and action strings. This information is provided to the system at run-time from the application. Information about fields for the condition string is maintained in the variable condBndryList, and for the action string, in the variable actBndryList. Both of these variables are pointers to the structure bndryType which has the following declaration:

```
struct bndryType {
   int start;
   int length;
   int lglValCnt;
   struct bndryValues *lglValues;
   struct bndryType *nxtBndry;
   };
```

The start member indicates the position where the field starts. Positions are numbered from 0 to msgSize-1 corresponding to each of the symbol positions in a condition or action string. Thus a field starting at the beginning of a condition or action string would have a start value of 0. The length member indicates the length of the field in symbols. The lglValCnt member indicates the number of permissible or legal values that the field can assume. The nxtBndry member points to the next field specification. Finally, the lglValues member points to a linked list of the legal values for the field. Each element in this list is contained in a bndryValues structure which has the following declaration:

```
struct bndryValues {
   long val1;
   long val2;
   struct bndryValues *nxtVal;
   };
```

The val1 and val2 members specify a legal value for the field. For condition string fields, these members correspond to the cond1 and cond2 members of the condType structure. For action string fields, these members correspond to the action1 and action2 members of the clfType structure. A legal value is encoded into these two members in a fashion similar to that of condition strings for a condition string field and to that of action strings for an action string field. This, the legal value "110##" would have a val1 of "00011" and a val2 of "11011" if it was for a condition string field, and a val1 of "00011" and a val2 of "11000" if it was for an action string field. From the declaration, the maximum length that a field can be is the bit length of a long integer, which for VAX machines is 32. Fields shorter then this maximum have their legal values stored "left-justified" in the val1 and val2 members. The nxtVal member is a pointer to the next legal value in the linked list. A pictorial description of the bndryType and bndryValues structures is provided in figure 3.2.

3.3.2.2. Classifier Selection for Mutation

After the number of weak classifiers has been determined, crossover and mutation are applied to the classifier pool in order to generate an equal number of new classifiers. Crossover is the first operator applied and, as will be discussed in the section on crossover, it generates new classifiers which will replace some of the weak classifiers. The remainder of the new classifiers to be generated comes from the mutation operator.

The selection of classifiers to be mutated basically involves probabilistically selecting classifiers, where higher strength classifiers are more likely to be selected.

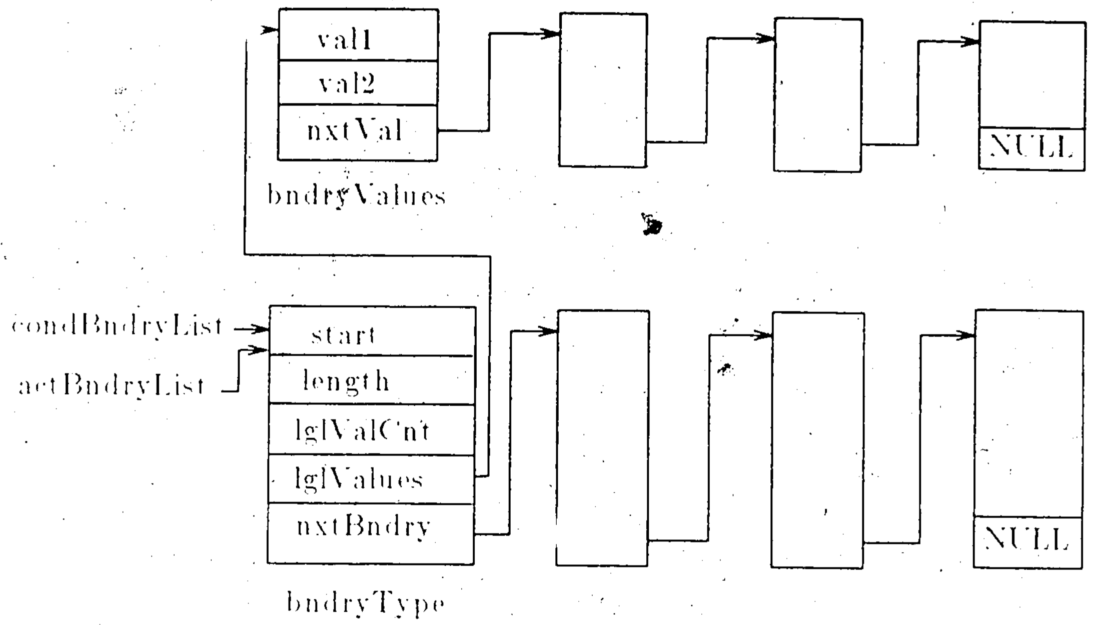


Figure 3.2 Data structures used by genetic operations.

This effect is achieved with the function detClfToUse and the variable mutationProb.

The variable mutationProb, which is set by the application, indicates roughly the chances that high strength classifiers will be used. This variable can assume integer values of 100 or less. A value of 100 indicates that only the highest strength classifiers will be used, while values less than that increase the chances that lower strength classifiers will be used.

The function detClfToUse is given mutationProb as a parameter. This function runs through the sortedClfPool array marking the use member of each classifier as either TRUE or FALSE; TRUE indicating that the classifier can be used for mutation. The use member of a classifier is marked TRUE if a randomly generated number (between 0 and 100) is less than or equal to mutationProb, otherwise it is marked FALSE. detClfToUse returns a number which indicates the number of classifiers which were marked as TRUE. Mutation is then performed on these classifiers, processing them in descending order of strength, until either the number of new classifiers required has been generated or until all the classifiers marked as TRUE have been used up, whichever comes first. Note that should the latter come first, then not all the weak classifiers will be replaced by new ones. This just means that the new generation will contain some weak classifiers which will probably be replaced in the next generation.

3.3.2.3. Mutation Operator

Mutation is performed by the function `mutate`. This function proceeds by first creating a new classifier which is a duplicate of the classifier to be mutated. Then, the `strength` and `op` members of the new classifier are set to `initStrength` and 'M' (for mutate) respectively. Finally, the new classifier is mutated. This involves first identifying which condition or action string is to be mutated. A string is randomly chosen, with equal probability, amongst all the condition strings and the action string. Once chosen, the field to be mutated and the legal value that it is to be mutated to are chosen at random. If the string is a condition string, this involves using `condBndryL-ist`; if it is an action string, this involves using `actBndryList`. After the field and legal value are chosen, the mutation is performed by replacing the c   ent value in the field with the chosen legal value.

3.3.2.4. Classifier Selection for Crossover

The first step in the selection of classifiers for crossover is to determine the number of new classifiers that crossover is produce. This is accomplished using the variable `crossoverRate`. This variable is set by the application and indicates what percentage of the new classifiers produced should be produced by crossover. It can assume integer values in the range of 0 to 100, where 0 indicates that no new classifiers are to be produced by crossover and 100 indicates that all the new classifiers are to be produced by crossover. The number of new classifiers to be produced by crossover is calculated by taking the percentage, indicated by `crossoverRate`, of the total number of new classifiers required. If this figure is odd, it is taken to be the next lowest even number as crossover can only be performed on an even number of classifiers. The resulting figure is then compared against the value returned by `detClfToUse` (which also, if necessary, is reduced to the next lowest even number) and the lower of the two becomes the actual number of classifiers that crossover is to produce.

As with mutation, crossover uses the function `detClfToUse` to probabilistically select the classifiers that it's going to use. In the case of crossover, the variable `crossoverProb` is used as the parameter to `detClfToUse`. This variable is set by the application and has the same range of values and meaning as `mutationProb`, but with regards to crossover.

After the actual number of classifiers to be produced (call it `n`) has been determined, crossover is applied to randomly selected pairs of classifiers. Each classifier in a pair is randomly chosen from amongst the `n` highest strength classifiers which have their `use` member set to TRUE. Once a classifier has been chosen for a pair, it will not be chosen again for any other pair. The result is the generation of `n` new classifiers produced from crossover. Mutation is then responsible for supplying the rest of the required new classifiers.

3.3.2.5. Crossover Operator

Crossover is performed by the function `crossover`. This function proceeds by first examining the two classifiers involved in the crossover. It may be the case that the two classifiers have a differing number of condition strings. For this case, the classifier with the least number of condition strings is temporarily given additional condition strings so as to have both classifiers containing an equal number. The additional condition strings given are the most general condition strings possible (i.e.

"####...#"). In this way, the meaning of the classifier is not altered by the additional condition strings since they are satisfied by any message, and at least one message must exist on the message list to satisfy the first condition string if the classifier is capable of firing.

In actuality, the way crossover proceeds is to first identify which of the two classifiers has the most number of condition strings. Then, two new "empty" classifiers are created and their `strength` and `op` members are set to `initStrength` and 'X' (for crossover) respectively. This is followed by performing crossover on each pair of condition strings. Crossover is first performed on the first condition string from each of the two classifiers giving the first condition string for each of the two new classifiers (see figure 3.3). Then it is performed on the second condition string from each of the two classifiers giving the second condition string for each of the two new classifiers, and so on. If one classifier has more condition strings than the other, then the classifier with the least number is treated as if it had additional condition strings ("####...#") when it runs out of condition strings.

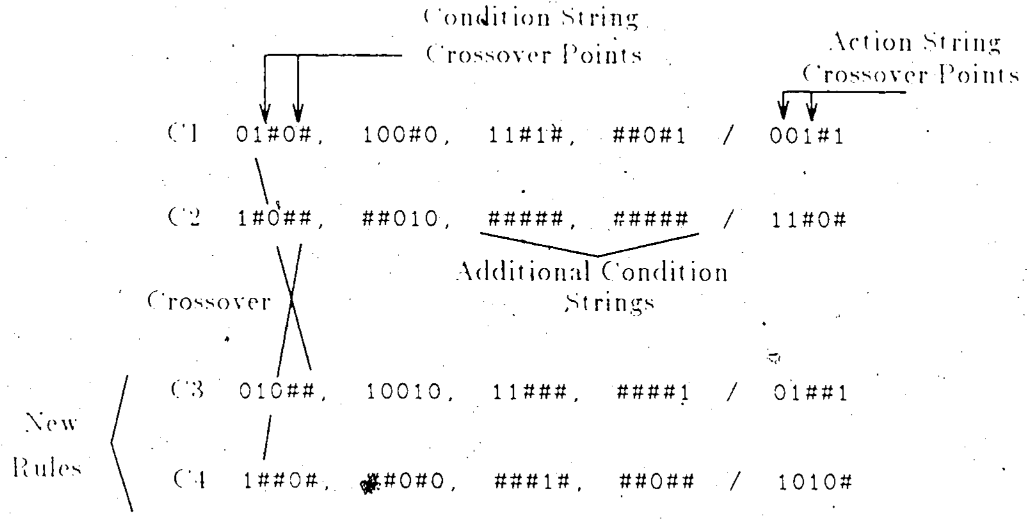

Condition String Crossover Points
Action String Crossover Points
C1 01#0#, 100#0, 11#1#, ##0#1 / 001#1
C2 1#0##, ##010, #####, ##### / 11#0#
Additional Condition Strings
Crossover
New Rules
C3 010##, 10010, 11###, ####1 / 01##1
C4 1##0#, ##0#0, ###1#, ##0## / 1010#

Figure 3.3 An example of crossover.

The crossover of a pair of condition strings first involves randomly determining the starting and ending points of the crossover. These crossover points are always "between" fields so that entire fields are involved in the crossover. Once determined, these points are used for the crossover of all the condition strings — condition strings all crossover at the same points. The crossover of a pair of condition strings just involves exchanging corresponding fields between the two condition strings. Only fields between the two crossover points are involved in the exchange.

crossover also performs crossover on the action strings. In this case, two new starting and ending crossover points are randomly determined since the field specification for the action string may be different from that of the condition string. Again, these points are "between" fields. Crossover of the pair of action strings proceeds in a fashion similar to that of the condition strings.

Figure 3.3 gives an example of the crossover of the two classifiers C1 and C2 producing the new classifiers C3 and C4. In this example, the crossover points of the condition strings occur between symbol positions two and three, and symbol positions four and five. Thus, the crossover of the first condition strings 01#0# for C1 and 1#0## for C2 result in the first condition strings 010## for C3 and 1##0# for C4. Similarly, the crossover of the second condition strings 100#0 for C1 and ##010 for C2 result in the second condition strings 10010 for C3 and ##0#0 for C4. Crossover proceeds in the same fashion for the rest of the condition strings. Notice that classifier C2 originally had only two condition strings (C2 originally was 1#0##, ##010 / 11#0#) but, because C1 has four condition strings, two additional condition strings were added to C2. These condition strings are of the most general form possible (i.e. #####). With these additional condition strings, the condition parts of C1 and C2 have the same length and thus crossover can be performed on them. Crossover of the action strings involves the selection of two new crossover points. In this example, these points occur between symbol positions one and two, and symbol positions three and four. Thus, the crossover of the action strings 001#1 for C1 and 11#0# for C2 result in the action strings 01##1 for C3 and 1010# for C4.

3.3.3. Replacing Weak Classifiers

After crossover and mutation have produced the list of new classifiers, `replaceClfs` is called to place them into the classifier pool replacing the weak classifiers. Each new classifier to be placed in the pool is first examined to determine if a duplicate already exists in the classifier pool. A duplicate classifier is any classifier which has the same condition and action parts. If a duplicate exists, then the new classifier is discarded and the classifier pool remains unchanged. Otherwise, the new classifier replaces the current weakest classifier in the pool. Note that prior to replacing a classifier, all the references to that classifier must be removed. This is accomplished by the function `removeClfRefs` which will remove any references to a classifier that are contained in the message pool or the payoff groups (`payoffGrps`). The production of a new generation is completed when the list of new classifiers has been exhausted. Note that since new duplicate classifiers are discarded, it is possible to have weak classifiers present in the new generation as not all the weak classifiers may be replaced. This just means that the new generation will start out with some weak classifiers which will probably be replaced in the next generation.

Chapter 4

System Enhancements

4.1. Introduction

Once the initial system had been developed, experiments were performed by applying it to the domain of tic-tac-toe. The results of these experiments indicated that enhancements to the system were necessary in order to achieve the desired level of performance. These enhancements and their implementation details are discussed in this chapter.

4.2. Taxation

One of the problems faced by the initial system is that classifiers which are not used are not removed from the classifier pool because their strength never falls below the threshold value. If a classifier is not used, then it does not receive any feedback (positive or negative) and its strength remains unchanged. This problem can be solved by the introduction of taxation [Rio86b].

In this system, taxation refers to the taxation of classifiers. The idea here is that classifiers should have to pay tax (a small portion of their strength) for their existence in the classifier pool. The amount of tax to be paid by a classifier is calculated by just taking some fraction of its strength. Taxation occurs at each iteration of the basic execution cycle. Through taxation, classifiers which are never used and classifiers which no longer prove us[illegible]ul (po[illegible]ibly because some better classifiers were discovered) will decrease in strength and even[illegible]ally fall below the threshold value.

A slight improvement on this idea is the introduction of different tax rates for different classifiers. Classifiers which have demonstrated more usefulness should be taxed at a lower rate. In this way, useless classifiers can be more quickly removed from the classifier pool due to higher taxation. One approximate measure of the usefulness of a classifier is the number of times it has fired. The more it has fired, the more useful it is likely to be. Given the fire count for a classifier, the tax rate can be simply calculated by dividing a fixed tax rate (`taxRate`) by the fire count. Actually, the fixed tax rate is divided by the fire count plus one since the fire count can have a value of zero. To avoid having too small a tax rate for classifiers which have fired a lot, a ceiling figure (`maxFireTax`) can be established for the fire count value used in the calculation. If the fire count is greater than the ceiling figure, then the ceiling figure is used in the calculation, otherwise the fire count is used. This establishes a minimum value for the tax rate of a classifier. The reason that one would want to avoid a very small tax rate for a classifier is that if a better classifier is later discovered, then, because of the small tax rate, it would take a very long time to remove the former classifier from the pool. Establishing a minimum tax rate avoids this problem.

The first step in implementing taxation is to implement the fire count. This is easily achieved by adding the member `fireCnt` to the `clfType` structure. Each time a new classifier is created, its `fireCnt` member is set to zero. Each time a classifier is fired, its `fireCnt` member is incremented by one.

With the fire count in place, taxation can now be implemented. This just involves reducing the strength of each classifier in the classifier pool by some multiplicative factor. Thus,

$$new\_strength = old\_strength * tax\_factor$$

where

$$tax\_factor = 1.0 - \texttt{taxRate} / (fire\_factor + 1)$$

and fire_factor is either the `fireCnt` for the classifier or `maxFireTax`, whichever is smaller. `taxRate` is a float which is set by the application and has a range of values from 0 to 1 where 0 indicates no tax. Usually, `taxRate` is set to some value close to zero, for example 0.002. `maxFireTax` is an integer which is set by the application and has a range of values from zero up. Higher values for maxFireTax results in lower taxation for classifiers which have higher fire counts. Note that since classifier strength is an integer and taxRate is a float, the smallest decrease in strength that can occur for any taxRate greater than zero is a decrease of one. This ensures that every classifier will be reduced in strength, due to taxation, for any taxRate greater than zero. Taxation is performed by the function taxClfs. This function is invoked in the basic execution cycle (executeOneCycle) just after the new message pool has been created (createNewMsgPool).

4.3. Bid Modification

Experimenting with the initial system revealed an interesting problem that arose between established classifiers (classifiers which had proven themselves useful) and new classifiers which competed (for firing) against the established ones. Essentially the problem was that in order to try to retain the best classifiers, new classifiers must be given the opportunity to prove themselves better than existing established classifiers. This means that when a new classifier and an established classifier compete for firing, the new classifier should initially outbid the established classifier so that it can fire and thus be given the chance to prove itself better. If the new classifier proves to be superior, then it should continue to outbid the established classifier. This will eventually result in the death of the established classifier as taxation will drive its strength below the threshold value and it will be replaced. If the new classifier proves to be inferior, then at some point it should underbid the established classifier. When this occurs, the established classifier should continue to outbid the new classifier and the new classifier will eventually die due to taxation.

To facilitate this scheme, the system was set up so that new classifiers were given an initial strength which was higher than that of any established classifiers in the system. This initial strength can be determined as the strength of established classifiers usually stabilize into some range of values. This stabilization occurs for a classifier when the amount that the classifier bids begins to balance off with the amount of feedback (payoff) it receives. At this stage, the classifier's strength will stabilize. Having a high initial strength for new classifiers ensures that they will initially outbid any of their established competitors. As a new classifier is used (fired), its strength will begin to stabilize. If it stabilizes into a range of values which are high enough that it continues to outbid its competitors then its competitors will eventually die, otherwise it will eventually die.

Now while this scheme for trying to retain the best classifiers seems to work quite elegantly, it makes the assumption that the strengths of the established classifiers remains unchanged during the interim it takes the new classifiers to stabilize. This, however, is not the case as the stabilization period requires some time, and during that period the established classifiers are experiencing taxation. This taxation can be large enough to cause a significant decrease in the strengths of the established classifiers resulting in the possible loss of those classifiers even if they are the better ones. This case occurs when the established classifiers would have outbid the stabilized new classifiers if there were no taxation but, because of taxation, fail to do so.

To solve this problem, one could set the `taxRate` and the `maxFireTax` so that established classifiers would experience small taxation. However, to do so would mean that it would take a long time to remove an established classifier should a better classifier be discovered.

Another solution to this problem is to decrease the amount of time it takes for new classifiers to stabilize. This is where the modification to the bid calculation comes in. The idea here is to get new classifiers to bid larger amounts so that they can drop quicker to their stabilizing values. A simple way to achieve this is to increase the bid_coefficient and divide the bid by the `fireCnt` or some maximum v ue (`maxFire-`

Bid), whichever is smaller. Thus, the new bid calculation is:

$$\text{bid} = \text{bid\_coefficient} * \text{strength} * \text{specificity} / (\text{fire\_factor} + 1)$$

where fire_factor is either the fireCnt for the classifier or maxFireBid, whichever is smaller. maxFireBid is an integer which is set by the application and has a range of values from zero up. The effect that is achieved with this new bid calculation is that new classifiers will bid larger amounts and thus experience more dramatic changes in strength resulting in quicker stabilizing times.

4.4. Cover Operator

When the initial system was set up and applied to tic-tac-toe, countered difficulties in acquiring knowledge about legal moves and consequently was unable to complete most of its games. This was due partly to the inability of the genetic operators to quickly produce classifiers that "covered" input messages to which the system was unable to respond. Such an input message fails to satisfy the condition part of any of the classifiers in the classifier pool resulting in a situation where none of the classifiers can respond (fire) to that message. Genetic operators should eventually produce a classifier which can respond to that message, however, there is usually a long interim between the time that the message first appears and the time that the responding classifier is produced. To combat this problem, a new discovery operator was introduced.

This operator, called cover [Rio86b], produces classifiers which respond to input messages to which the system was previously unable to respond. It "covers" those input messages. Cover proceeds in two stages: first, messages which need to be covered are collected, then, classifiers which cover those messages are generated.

Message collection is performed once in every iteration of the basic execution cycle (executeOneCycle). This occurs after all the possible matches between messages and classifiers have been determined (after getMatches has been executed). At this point, coverCheck is called to add any new messages to the list of messages to be covered.

This function proceeds by first identifying those messages on the current message list that are allowed to be covered. Normally, only input messages are covered. However the system does not know how to tell input messages apart from other messages. To rectify this the system is initially given a condition string (coverCond) from the application which indicates the set of messages that are allowed to be covered. Any message which satisfies the condition string belongs to this set. Using the condition string, coverCheck ~~first~~ examines the current message list and identifies those messages which satisfy the condition string. Then, of those messages, coverCheck discards any messages which participate in a match. These are messages which appear in any of the matches in the match pool. Finally, coverCheck adds the remaining messages to the list of messages to be covered (coverMsgs). Note that a message is added to this list only if it does not already appear in this list.

New classifiers are generated each time a new generation is produced. The cover operator takes precedence over the two genetic operators as it gets first crack at producing new classifiers. As with crossover, cover uses a variable (coverRate) to determine how many classifiers it is allowed to produce. coverRate is a variable which is set by the application and indicates what percentage of the new classifiers are to be produced by cover. This variable can assume integer values in the range of 0 to 100, where 0 indicates that no classifiers are to be produced by cover. Once the number of classifiers to produce has been determined, cover is repeatedly called until either that many classifiers have been produced or until coverMsgs has been exhausted, whichever comes first. The remaining number of classifiers to be produced is then passed on to crossover and mut: [illegible] ich proceed as before.

Each time `cover` is called, it first checks to see whether there are any messages left that need to be covered (is `coverMsgs` empty). If there are, `cover` removes the first message from `coverMsgs` and generates a classifier which covers that message. It then returns TRUE indicating that it was successful in producing a new classifier. If there are no messages left to be covered, then `cover` just returns FALSE. When `cover` produces a new class[illegible]er, it does so by first creating the new classifier and setting that classifier's `strength` and `op` members to `initStrength` and 'C' (for cover) respectively. The condition part of the classifier is then set so as to contain one condition string which matches, exactly, the message to be covered. This means that the condition string will contain only "1"s and "0"s, no "#"s. Then, the action part of the classifier is constructed randomly. This is done by mutating each field in the action string to some value randomly chosen from amongst all the legal values for that field. This completes the construction of the new classifier.

4.5. Generalization Operator

While the cover operator covers input messages to which the system cannot respond, the classifiers it produces are very specific, being applicable to only the input messages they were designed to cover. Crossover and mutation can produce more general classifiers (classifiers which contain more "#"s in their condition part) from these specific ones, but they tend to be quite slow in doing so. In an effort to speed up this process, the generalization operator was introduced.

This operator proceeds by organizing classifiers in the classifier pool into generalization groups and then creating generalizations from these groups. This organization of classifiers is actually an ongoing process as these groups are incrementally updated when classifiers are added or removed from the pool. When it is time to produce a new generation, generalization will [illegible]lled upon to create some of the new classifiers. These new classifiers will [illegible]ted from the generalization groups, each classifier being the product of a single group. Groups will be selectively chosen to be generalized since there will likely be more groups than the number of new classifiers to be created.

Generalization groups are maintained in the data structures:

```
struct genrlClf {
   struct clfType *clf;
   struct genrlClf *nxtClf;
   };


struct genrlGrp {
   int clfCnt;
   int gen;
   int condMtch;
   struct genrlClf *clfs;
   struct genrlGrp *nxtGrp;
   };
```

The genrlClf structure is used to maintain a linked list of classifiers which form a generalization group. Its clf member points to a classifier in the group and its nxtClf member points to the next element in the linked list. The genrlGrp structure is used to maintain a linked list of all the generalization groups. For this structure, the clfCnt member indicates the number of classifiers in a particular group, the gen member indicates whether or not that group has been generalized, the condMtch member indicates the similarity of the classifiers in that group, the clfs member points to the classifiers in that group, and the nxtGrp member points to the next group in the list. The variable generalizations points to the first group in the list.

The addition of classifiers to the classifier pool also requires updating the generalization groups. This is performed by the function addToGen, which adds a classifier to one of the generalization groups. If an appropriate group cannot be found for the classifier, then a new group is created and the classifier is placed there.

A classifier can only be added to a group if in doing so the resulting group has the same action part for all its classifiers, the same prefix for the [illegible] condition string for all its classifiers, and a similarity greater than or equal to genCondMtch. The similarity of a group is determined from the first condition strings of all of its classifiers, and is computed by determining the number of positions where the symbols are the same for all of these condition strings. So, for example, if a group had two classifiers which had the first condition strings "1#0##" and "##00#" then the group would have a similarity of 3 since the symbols in positions 2, 3, and 5 are the same. If another classifier having the first condition string "1#011" was added to the group, then the symbols in the fifth position would not all be the same anymore, and the similarity would drop to 2. The similarity of a group must always be greater than or equal to genCondMtch. This is a variable which is set by the application and determines how bold or weak the generalizations created will be. It assumes integer values in the range of 0 to msgSize where higher values indicate the creation of weaker generalizations. Exactly how generalizations are created will be discussed shortly.

When addToGen is given a classifier to add to a group, it must decide what group, if any, it is to add it to. The only groups in which it is allowed to be added are those which satisfy the three criteria given previously. If more than one of these groups exist, then addToGen will add it to the group which has the highest similarity after the classifier has been added. If none of these groups exist, then a new group is created and the classifier is placed in there.

Removing a classifier from the classifier pool also requires updating the generalization groups. This is performed by the function rmvFromGen which locates the group that contains the classifier, removes the classifier from that group, and, if the resulting group is empty, removes the group from the list of groups. ot that both the addition and removal of a classifier to a generalization group also involves appropriate updating of the genrlGrp members: clfCnt, gen, and condMtch.

When it is time to produce a new generation, some of the new classifiers to be created may be created by generalization. The number of new classifiers to be produced by generalization is determined by generalizeRate. This variable is set by the application and serves the same purpose as coverRate, but with respect to generalization. Generalization takes precedence over cover as it gets first crack at producing its percentage of new classifiers. The remainder of new classifiers to produce then falls into the hands of cover, then crossover, and finally mutation (just as before). As with cover, generalization may be unable to fulfill its quota of new classifiers. This is handled in the same way that cover handles it.

Generalization is performed by the function generalize. This function produces a single generalization (classifier) and returns TRUE if it was successful, FALSE otherwise. It proceeds by first determining which group to generalize. Only groups which have more than one classifier and have not been generalized yet (gen is FALSE) are considered. Of those groups, a score is computed for each, and the group with the highest score gets generalized.

The score for a group is computed by taking the sum of the strengths of the classifiers in the group and multiplying that by the similarity (condMtch) of the group. This means that groups containing more classifiers, higher strength classifiers, and having higher similarities will be favored. Notice that there is a tradeoff between the number of classifiers in a group and its similarity — the more classifiers it has, the lower its similarity will likely be; and visa versa. In effect, what this score calculation says is that a large group of good classifiers which say roughly the same thing will likely form a good (though weaker) generalization. The reason for the bias towards higher similarities is that although it may produce a weaker generalization, it is more likely to be a "safe" generalization. Also, later on, the weaker generalizations can in turn be generalized to produce more bold generalizations. This gives a more gradual, but more successful development of bold generalizations.

After examining all the groups to determine the one to generalize, it may be possible that none of the groups are able to satisfy the criteria of having more than one classifier and having not been generalized yet. In this case, `generalization` just returns FALSE. Otherwise, it generalizes the highest scoring group. This involves first creating the condition string of the generalizing classifier. To do this, the first condition strings of all the classifiers in the group are examined and positions which have the same symbol throughout these strings results in that symbol being placed in the same position in the new condition string. All other positions in the new condition string have the symbol "#" placed in them. This gives the condition string for the generalizing classifier. The action string for this classifier is simply the same action string that is present in all the classifiers in the group. The strength that this classifier is assigned is determined from the variable `genLife`. This variable is set by the application and results in the strength being set to a value which will allow this classifier to outbid the classifiers from which it was formed, `genLife` times, assuming that it gets no feedback (zero payoff) each time it fires. This strength setting gives the application more control over determining the length of the "trial period" that the generalizing classifier has to prove itself than if the strength were set to `initStrength`. Finally, the `op` member of the generalizing classifier is set to "G" (for generalization).

Before returning, `generalize` updates the generalization group by setting `gen` to TRUE. Note that the addition or removal of classifiers from this group can result in `gen` being set back to FALSE. After updating the group, `generalize` then exits returning a value of TRUE.

At present, generalization only works with classifiers whose condition part contains only one condition string. Future development of this system might extend this to work for multiple condition strings.

Chapter 5

Writing Application Software

5.1. Introduction

This chapter describes the steps involved in writing an application for LCS. Currently, writing application software involves the use of an interpretive programming language written by Keith Fenske, called face [Fen88]. This language has special features which allow it to "run" LCS. In addition, it provides the user with high level programming constructs and the ability to symbolically manipulate strings (of "0"s, "1"s, and "#"s). These and other features make face a useful language for writing application software.

A user who wishes to apply LCS to a particular application would start by writing the application software in face. Having done so the user would then invoke face, entering the face programming environment, load his application code, and execute it. Execution of the code will in turn automatically cause execution of LCS. Communication between the application and LCS is done completely through face. A description of the face programming language is given in the face Language Description manual [Fen88].

This chapter focuses on the elements that must be present in the application software in order for it to correctly invoke and use LCS. An example of a simple application is provided at the end giving the face code involved as well as a detailed trace of the execution.

5.2. Code Essentials

Any application software must perform some essential tasks. It must initialize LCS, perform the basic execution cycle, and produce new generations. This section describes what is required on behalf of the application software in order to perform these tasks. The code for the face system functions described in this section (the functions which communicate with LCS) is written in face and can be found in the file user.f. Note that this file must be loaded into the face programming environment prior to loading the application software.

5.2.1. Initialization

Before LCS can be used, it must first be initialized. Initialization usually occurs at the start of the application program. This tailors the system to suit the application. Initialization usually involves the following sequence of calls (in face):

```
clear();
seed(seedNum);
msgSize(size);
clfPostLvl(postLvl);
maxNumConds(numConds);
bidCoeff(coeff);
maxFireBid(fireCnt);
fireProb(prob);
taxRate(rate);
maxFireTax(fireCnt);
condBndry(start, length, lglValCnt, "lglVal1, lglVal2, .."); 

actBndry(start, length, lglValCnt, "lglVal1, lglVal2, ..");

initStrength(strength);
rplcThres(thresh);
genCondMtch(mtch);
genLife(length);
coverCond(cond);
crossoverProb(prob);
mutationProb(prob);
```

Explanation of these calls and their parameters can be found in appendix A1.

After performing these calls, LCS is almost ready to execute. The only thing missing is the initial rule base. Rules are specified to LCS by performing a series of the call:

rule({{*condStr1*, *condStr2* ..}, *actionStr*, *strength*, *fireCnt*});

In this call, the condition and action strings are strings in the face language consisting of the characters "0", "1", and "#". The *strength* and *fireCnt* are both integers which indicate the strength of the rule and the number of times that the rule has fired, respectively. *strength* is usually set to the value specified in initStrength and *fireCnt* is usually set to 0; except in the case where the initial rule base is the same as some rule base produced in a previous run. So, for example, an initial rule might be:

```
rule({{"100#101"},"011#001",1000,0});
```

Please note that it is the responsibility of the application program to ensure that the parameters of the rule call are consistent with the other initialization calls made previously, as no error checking is performed.

Since the parameters of the rule call are just sets of strings and integers in face, the construction of an initial rule involves constructing these strings and integers using face. So, for example, if an initial rule with a random condition and action string is desired, then face code must be written which constructs a random condition and action string that are, in turn, used in the rule call. Note that face makes this easy since it allows these strings to be symbolically constructed.

Repeatedly constructing rules and relaying them to LCS through the rule call establishes the initial rule base. Once done, the system is ready to execute.

5.2.2. The Basic Execution Cycle

After initialization has been performed, the application program typically proceeds by iterating for some number of generations. In each generation it performs the basic execution cycle for some number of iterations and then turns out the next generation of classifiers. The number of iterations performed for the basic execution cycle and for the generations is determined by the application code. What LCS provides are the functions required to perform a basic execution cycle and to produce a new generation. The organization as to when a basic execution cycle is performed or when a new generation is produced is completely determined by the application code as it is this code that determines when these functions are called. This section describes the functions related to the basic execution cycle. The next section describes those related to producing new generations.

The first step in the basic execution cycle is to construct the input messages. The application program is responsible for constructing each input message (which is a string of "0"s and "1"s). Again, face makes this task easier by allowing strings to be manipulated symbolically. After an input message string has been constructed, it is placed on the message list by calling:

```
message(message string);
```

So, the call:

```
message("100101");
```

would place the message "100101" on the message list.

After all the input messages have been placed on the message list, the heart of the basic execution cycle is performed by calling:

```
nextcycle();
```

This will result in the generation of the new message list.

The messages in the message list can then be accessed through the `face` variable `messlist`. This variable contains a set where each member is a set containing the message string followed by the id number of the message. So, for example, `messlist` might look like:

```
{{"011010", 10}, {"001001", 11}, {"011100", 12}}
```

By accessing the elements in this set (as described in the `face` Language Description manual), one can retrieve the output messages and then processes them appropriately. Note that `messlist` will be an empty set when the message list is empty. This can occur when no rules are capable of firing at the time `nextcycle` is called.

After processing the output messages, the next set of input messages are constructed and the cycle repeats. Optionally, payoff may be issued to the active rules (payoff group number 0) prior to the placement of the next set of input messages on the message list. This is done by calling:

`payoff(0,`*amt*`);`

As an example, one might issue the payoff:

```
payoff(0,100),
```

which would payoff 100 to the active rules. Also, at this stage, the active rules can be added to one or more payoff groups by calling

`addToPOGrp(`*group_num*`);`

for each group. So, if the active rules were to be added to group numbers 2 and 3, the calls

```
addToPOGrp(2);
addToPOGrp(3);
```

would be performed. After payoff and group addition of the active rules has c pleted, the cycle continues with the next set of input messages. Note that issuing payoff to any group other than group number 0, or removing a payoff group can be performed at any place in the cycle. Payoff is issued to a group by calling:

```
payoff(group_num, amt);
```

The removal of a group is accomplished by calling:

```
rmvPOGrp(group_num);
```

5.2.3. Producing New Generations

Before a new generation can be produced, the degree to which each of the discovery operations are involved in the production of new rules must be set. This is done by performing the calls:

```
generalizeRate(rate);
coverRate(rate);
crossoverRate(rate);
```

Once set, a rate remains in effect until the appropriate call is reissued at which point it assumes a new rate. Thus, to change the cover rate, the call coverRate would be performed with the new rate value as its parameter. As was discussed in chapter 3, mutation is responsible for producing any new rules left to be produced after generalization, cover, and crossover have completed. Thus, the rate of mutation does not need to be set as it is always effectively 100 percent since it produces all the new rules left to be produced when its turn (to produce new rules) comes up. Further explanation of generalizeRate, coverRate, and crossoverRate can be found in appendix A1.

After the rates have been established, the next generation of rules is produced by calling:

```
nextgeneration();
```

This will result in the generation of new rules which replace all those rules whose strength is below the threshold. If all the rules have their strength above the threshold then no new rules will be generated.

At any time, the rule base can be accessed through the face variable `rulelist`. This works in a fashion similar to `messlist`. In this case, each member of the `rulelist` set looks like:

{{*condStr1, condStr2, ..*}, *actionStr, strength, fireCnt, discovOp, id*} .

Through this variable the rule base can be examined to see how it is evolving as the program proceeds.

5.2.4. Additional Functions

LCS also provides two additional functions which can aid the application program. The first of these functions is `clfcntbelowthres`. It requires no parameters and returns an integer which indicates the number of rules that currently have a strength below the threshold. The second of these is `trackmsg`. It takes one parameter which is the id number of a message that is currently on the message list. It returns a printable string which represents the rule that placed that message on the message list. This function is mainly used for debugging purposes when it is desired to see which rules have fired.

5.2.5. Exiting

When the application program has finished with LCS, it may perform the call:

```
close();
```

This closes the connection to LCS. This is also done implicitly upon exiting `face`.

5.3. A Simple Problem

The following is a simple example of applying LCS to the light switch problem [Fen88]. This is the problem of getting the system to learn when a light is on/off. There are two switches to the light. When both switches are up or both are down, the light is on; otherwise the light is off. The system is presented with the state of the two switches (randomly generated) from which it tries to determine the state of the light. If it determines the light state correctly then it is given positive feedback (rightpo), otherwise it is given negative feedback (wrongpo).

Messages for this application are three bits long. The first bit of a message indicates whether it is an input message ("1") or an output message ("0"). For input messages, the next two bits indicate the state of the switches. A bit value of "0" indicates that a switch is down, "1" indicates that it is up. For output messages, the second bit is always "0" and the third bit indicates the state of the light ("0" for off, "1" for on). Thus, the system might be given the message "101" which says that switch one is down and switch two is up. The correct response that the system should return is the message "000" which says that the light is off.

As the system iterates through the basic execution cycle, it will identify the classifiers which correctly determine the state of the light. Incorrect classifiers will be replaced by new classifiers produced by the discovery operators. At the end, the system should contain the classifiers:

```
100 / 001
101 / 000
110 / 000
111 / 001
```

Note that if "#"s are allowed in the action part then the last two rules could be replaced by the single rule 11# / 00#. However, in order to keep the system trace (which follows) simple, this was not allowed.

Following is the face code for the light switch ~~problem~~. Although use of the generalization operation would be counterproductive for this application, it has nevertheless been set up to be used for the sake of completeness. Any classifiers produced by generalization should eventually be replaced as they will produce an incorrect response for at least one of the light switch combinations. (Generalization will produce classifiers which contain a "#" as either the second or third symbol in their condition string. Such classifiers attempt to determine the state of the light knowing only the value of one of the switches. This will obviously lead to a wrong answer at some point in time.)

This program proceeds by first performing some initialization. Then, an initial random rule base is constructed. Finally, the generation cycles are performed. In each generation, the basic execution cycle is iterated until a specified number of classifiers (rplcSize) fall below the strength threshold or until the maximum number of cycles has been executed; whichever comes first. This maximum ensures that the program will not run on forever.

```
#
# Light Switch Program
#

#
# This function performs initialization.
#
#
function initialize()
do
   #
   # Set the right and wrong payoffs.
   #
   rightpo:=999;
   wrongpo:=-99999;   # large value ensures quick removal of bad rules
   #
   # Set the number of generations to be performed and the maximum
   # cycles per generation.
   #
   genCnt:=40;
   maxCycCnt:=10;
```

```
    #
    # Clear the learning classifier system.
    #
    clear();
    #
    # Set the parameters.
    #
    seed(125);
    msgSize(3);
    clfPostLvl(1);
    maxNumConds(1);
    bidCoeff(0.3);
    maxFireBid(2);
    fireProb(100);
    taxRate(0.3);
    maxFireTax(3);
    rplcThres(300);
    condBndry(0,1,1,"1");
    condBndry(1,1,3,"0 1 #");
    condBndry(2,1,3,"0 1 #");
    actBndry(0,2,1,"00");
    actBndry(2,1,2,"0 1");
    initStrength(10000);
    genCondMtch(2);
    genLife(2);
    coverCond("1##");
    crossoverProb(100);
    mutationProb(100);
end;

#
# This function generates the initial rule base which consists of
# four rules.
#
#
function getInitRules(i,          # counter (local)
                      cond,       # condition string (local)
                      act,        # action string (local)
                      symb)       # a symbol (local)
do
   for i from 1 to 4 do
      #
      # Construct a random condition string.
      #
      cond:="1";
      symb:="01#"[random(3)+1];          # get a random symbol
      cond:=cond+symb;
      symb:="01#"[random(3)+1];          # get a random symbol
      cond:=cond+symb;
      #
      # Construct a random action string.
      #
      act:="00"+"01"[random(2)+1];
```

```
        #
        # Add the rule to the rule base, setting the initial strength to
        # 10000 and the initial fire count to 0.
        #
        rule({{cond},act,10000,0}),
     end;
  end;

  #
  # This is the main function.
  #
  #
  function light(rplcSize,          # replace size (local)
                 g,                 # generation counter (local)
                 cycCnt,            # cycle count counter (local)
                 right,             # num. of correct answers (local)
                 wrong,             # num. of wrong answers (local)
                 nrc,               # num. of no responses (local)
                 mess,              # a message (local)
                 first,             # first switch (local)
                 second)            # second switch (local)
  do
     initialize();
     getInitRules();
     rplcSize:=3;         # set the replace size
     for g from 1 to genCnt do
        write("\n--- Generation #",g," ---\n\n");
        displayset(rulelist);
        #
        # Zero the stat counters.
        #
        right:=0;
        wrong:=0;
        nrc:=0;
        cycCnt:=0;
        #
        # Print header.
        #
        write("\nInput  Output  Correctness  Classifier\n");
        #
        # Perform the basic execution cycle.
        #
        while ((clfcntbelowthres()<rplcSize) and (cycCnt<maxCycCnt)) do
           #
           # Get two random switch settings.
           #
           first:="01"[random(2)+1];
           second:="01"[random(2)+1];
           #
           # Display the switch settings.
           #
           write("  ",first+second,"       ");
           #
```

```
# Construct an input message from the switch settings.
#
mess:="1"+first+second;
message(mess);
#
# Call nextcycle.
#
nextcycle();
#
# Process the output message (if there is one)
#
if (size(messlist)>0) then
   #
   # The system had a response - an output message exists.
   #
   write(messlist[1][1][3],"        ");   # Display the output message.
   if (messlist[1][1]=="000") then
      if (first==second) then
         #
         # Both switches are in the same position, but the system
         # says that the light is off - wrong.
         #
         payoff(0,wrongpo);
         wrong:=wrong+1;
         write("wrong     ");
      else
         #
         # The switches are in opposite positions and the system
         # says that the light is off - right.
         #
         payoff(0,rightpo);
         right:=right+1;
         write("right     ");
      end;
   else
      if (first==second) then
         #
         # Both switches are in the same position and the system
         # says that the light is on - right.
         #
         payoff(0,rightpo);
         right:=right+1;
         write("right     ");
      else
         #
         # The switches are in opposite positions, but the system
         # says that the light is on - wrong
         #
         payoff(0,wrongpo);
         wrong:=wrong+1;
         write("wrong     ");
      end;
   end;
```

```
            #
            # Display the rule which produced the output message.
            #
            write(trackmsg(messlist[1][2]),"\n");
         else
            #
            # The system had no response - no output message exists.
            #
            nrc:=nrc+1,
            write("no response\n");
         end;
         cycCnt:=cycCnt+1;
      end;
      #
      # Display the stats.
      #
      write("\nright ",right,", wrong ",wrong,", no response ",nrc,"\n");
      write("cycle count ",cycCnt,"; #clfs below thres ",
         clfcntbelowthres(),"\n");
      #
      # Produce the next generation.
      #
      if (g==1) then
         #
         # This is the first generation.
         #
         generalizeRate(0);
         coverRate(100);
         crossoverRate(67);
      elif (nrc>0) then
         #
         # There are still some "no responses".
         #
         generalizeRate(0);
         coverRate(100);
         crossoverRate(67);
      else
         #
         # The system was able to respond to all the light switch
         # configurations produced in this generation
         #
         generalizeRate(50);
         coverRate(100);
         crossoverRate(67);
      end;
      #
      # Call nextgeneration.
      #
      nextgeneration();
   end;
   write("\n*** Done ***\n");
end;
```

For this program, the tax rate was calculated based on how many cycles an unfired rule should survive before falling below threshold. The idea is that an unfired rule should survive at least long enough to "see" any input to which it can respond. That is, a rule should at least survive long enough to be given the chance to prove its worthiness. In this case, there are four different types of input to the system (four possible light switch combinations). Doubling that, and adding on a bit more for safe measure, gives 10 cycles as the survival length of an unfired rule. This is more than adequate for an unfired rule to be given the chance to prove itself. Thus, with an initial strength of 10000 and a threshold strength of 300, the tax rate is calculated as follows:

$$300 = 10000(1 - taxRate)^{10}$$

giving a tax rate of approximately 0.3.

The generalization, cover, and crossover rates for the first generation and any generation with inputs which resulted in "no response" are set to 0, 100, and 67 respectively. For these generations, the rule base probably does not have enough good rules from which to form generalizations, thus that rate is set to 0. However, there are most likely situations which had "no response" and consequently the cover rate is set to 100. After the cover operation, if there are three or more rules left to be replaced, then some of them will be replaced by crossover. For all the other generations, the generalization, cover, and crossover rates are set to 50, 100, and 67. This gives generalization a chance to produce some rules and cover the opportunity to cover any situations left on its queue to be covered. Again, if three or more rules remain to be replaced after these two operations, then some of them will be replaced by crossover.

Following are the results of a run of the light switch program. At the beginning of each generation, the current rule base is displayed. Each rule is displayed in the format:

{{condition}, action, strength, fire count, discovery op., id}

This is followed by a trace of the basic execution cycle which displays the input light switches state, the output light state, the correctness of the output, and the rule (after firing) which produced the output. At the end of each generation, statistics for that generation are displayed.

Trace (user input is in italics):

```
face da class: an interactive classifier programming language

ready for input
load("lightrun.f");
loading file 'lightrun.f'
loading file 'pretty.f'

end-of-file on file 'pretty.f'
loading file 'user.f'

end-of-file on file 'user.f'
loading file 'light.f'

end-of-file on file 'light.f'

end-of-file on file 'lightrun.f'
light();

--- Generation #1 ---

{
{{"10#"}, "000", 10000, 0, "I", 0}
{{"1##"}, "001", 10000, 0, "I", 1}
{{"100"}, "001", 10000, 0, "I", 2}
{{"111"}, "000", 10000, 0, "I", 3}
}.

Input  Output  Correctness  Classifier
  11     0       wrong      {{"111"},"000",-94050,1,"I",3}
  10     1       wrong      {{"1##"},"001",-94645,1,"I",1}
  11     1       right      {{"1##"},"001",-84181,2,"I",1}
  11     1       right      {{"1##"},"001",-76868,3,"I",1}
  10     1       wrong      {{"1##"},"001",-171101,4,"I",1}
  00     1       right      {{"100"},"001",1998,1,"I",2}
  10     1       wrong      {{"1##"},"001",-246396,5,"I",1}
  10     1       wrong      {{"1##"},"001",-327915,6,"I",1}
  00     1       right      {{"100"},"001",2103,2,"I",2}
  01     0       right      {{"10#"},"000",1272,1,"I",0}
```

```
right 5, wrong 5, no response 0
cycle count 10, #clfs below thres 2

--- Generation #2 ---

{
{{"10#"}, "000", 1272, 1, "I", 0}
{{"1##"}, "001", -280571, 6, "I", 1}
{{"100"}, "001", 1892, 2, "I", 2}
{{"111"}, "000", -21781, 1, "I", 3}
}

Input  Output  Correctness  Classifier
  10      1       wrong     {{"1##"},"001",-359527,7,"I",1}
  01      0       right     {{"10#"},"000",1874,2,"I",0}
  00      1       right     {{"100"},"001",2273,3,"I",2}
  10      1       wrong     {{"1##"},"001",-384546,8,"I",1}
  00      1       right     {{"100"},"001",2749,4,"I",2}
  01      0       right     {{"10#"},"000",2177,3,"I",0}
  01      0       right     {{"10#"},"000",2878,4,"I",0}
  01      0       right     {{"10#"},"000",3484,5,"I",0}
  01      0       right     {{"10#"},"000",4007,6,"I",0}
  11      1       right     {{"1##"},"001",-239877,9,"I",1}

right 8, wrong 2, no response 0
cycle count 10, #clfs below thres 2

--- Generation #3 ---

{
{{"10#"}, "000", 3706, 6, "I", 0}
{{"100"}, "000", 10000, 0, "M", 7}
{{"100"}, "001", 1859, 4, "I", 2}
{{"111"}, "000", -4285, 1, "I", 3}
}

Input  Output  Correctness  Classifier
  10      no response
  01      0       right     {{"10#"},"000",3958,7,"I",0}
  01      0       right     {{"10#"},"000",4416,8,"I",0}
  11      0       wrong     {{"111"},"000",-102365,2,"I",3}
  01      0       right     {{"10#"},"000",4525,9,"I",0}
  10      no response
  00      0       wrong     {{"100"},"000",-99300,1,"M",7}
  00      0       wrong     {{"10#"},"000",-96657,10,"I",0}

right 3, wrong 3, no response 2
cycle count 8, #clfs below thres 3

--- Generation #4 ---

{
{{"110"}, "001", 10000, 0, "C", 8}
{{"100"}, "000", -84404, 1, "M", 7}
{{"100"}, "001", 993, 4, "I"; 2}
{{"111"}, "000", -67160, 2, "I", 3}
}

Input  Output  Correctness  Classifier
  00      1        right     {{"100"},"001",1825,5,"I",2}
  11      0        wrong     {{"111"},"000",-155908,3,"I",3}
  01      no response
  10      1        wrong     {{"110"},"001",-97959,1,"C",8}

right 1, wrong 2, no response 1
cycle count 4, #clfs below thres 3

--- Generation #5 ---

{
{{"110"}, "001", -97959, 1, "C", 8}
{{"100"}, "000", -44058, 1, "M", 7}
{{"100"}, "001", 1443, 5, "I", 2}
{{"101"}, "000", 10000, 0, "C", 11}
}

Input  Output  Correctness  Classifier
  01      0        right     {{"101"},"000",6948,1,"C",11}
  00      1        right     {{"100"},"001",2109,6,"I",2}
  10      1        wrong     {{"110"},"001",-163696,2,"C",8}
  00      1        right     {{"100"},"001",2622,7,"I",2}
  11      no response
  01      0        right     {{"101"},"000",3773,2,"C",11}
  00      1        right     {{"100"},"001",2866,8,"I",2}
  11      no response
  11      no response
  00      1        right     {{"100"},"001",3040,9,"I",2}

right 6, wrong 1, no response 3
cycle count 10, #clfs below thres 2

--- Generation #6 ---

{
{{"111"}, "000", 10000, 0, "C", 14}
{{"100"}, "000", -8670, 1, "M", 7}
{{"100"}, "001", 3040, 9, "I", 2}
{{"101"}, "000", 2474, 2, "C", 11}
}

Input  Output  Correctness  Classifier
  10      no response
  11      0        wrong     {{"111"},"000",-95835,1,"C",14}
  11      0        wrong     {{"111"},"000",-186250,2,"C",14}
  00      1        right     {{"100"},"001",3000,10,"I",2}
  00      1        right     {{"100"},"001",3496,11,"I",2}
  11       0        wrong     {{"111"},"000",-239545,3,"C",14}
  11       0        wrong     {{"111"},"000",-321578,4,"C",14}
  01       0        right     {{"101"},"000",1981,3,"C",11}
  11       0        wrong     {{"111"},"000",-375148,5,"C",14}
  11       0        wrong     {{"111"},"000",-447010,6,"C",14}

right 3, wrong 6, no response 1
cycle count 10, #clfs below thres 2

--- Generation #7 ---

{
{{"110"}, "001", 10000, 0, "C", 16}
{{"100"}, "000", -1704, 1, "M", 7}
{{"100"}, "001", 2365, 11, "I", 2}
{{"101"}, "000", 694, 3, "C", 11}
}

Input  Output  Correctness  Classifier
  10      1       wrong      {{"110"},"001",-94050,1,"C",16}
  01      0       right      {{"101"},"000",2303,4,"C",11}
  00      1       right      {{"100"},"001",2682,12,"I",2}
  11      no response
  10      1       wrong      {{"110"},"001",-151980,2,"C",16}
  01      0       right      {{"101"},"000",2515,5,"C",11}
  11      no response
  00      1       right      {{"100"},"001",2631,13,"I",2}
  01      0       right      {{"101"},"000",2789,6,"C",11}
  10      1       wrong      {{"110"},"001",-192231,3,"C",16}

right 5, wrong 3, no response 2
cycle count 10, #clfs below thres 2

--- Generation #8 ---

{
{{"111"}, "000", 10000, 0, "C", 18}
{{"100"}, "000", -332, 1, "M", 7}
{{"100"}, "001", 2250, 13, "I", 2}
{{"101"}, "000", 2579, 6, "C", 11}
}

Input  Output  Correctness  Classifier
  11      0       wrong      {{"111"},"000",-94050,1,"C",18}
  10      no response
  00      1       right      {{"100"},"001",2601,14,"I",2}
  10      no response
  10      no response
  00      1       right      {{"100"},"001",2850,15,"I",2}
  01      0       right      {{"101"},"000",2342,7,"C",11}
  00      1       right      {{"100"},"001",3194,16,"I",2}
  10      no response
  10      no response
```

```
right 4, wrong 1, no response 5
cycle count 10, #clfs below thres 2

--- Generation #9 ---

{
{{"110"}, "000", 10000, 0, "C", 20}
{{"100"}, "000", -63, 1, "M", 7}
{{"100"}, "001", 2732, 16, "I", 2}
{{"101"}, "000", 1852, 7, "C", 11}
}

Input  Output  Correctness  Classifier
  10      0        right     {{"110"},"000",6948,1,"C",20}
  11      no response
  00      1        right     {{"100"},"001",2945,17,"I",2}
  00      1        right     {{"100"},"001",3451,18,"I",2}
  01      0        right     {{"101"},"000",2127,8,"C",11}
  11      no response
  10      0        right     {{"110"},"000",3356,2,"C",20}
  00      1        right     {{"100"},"001",3271,19,"I",2}
  00      1        right     {{"100"},"001",3722,20,"I",2}
  01      0        right     {{"101"},"000",2293,9,"C",11}

right 8, wrong 0, no response 2
cycle count 10, #clfs below thres 1

--- Generation #10 ---

{
{{"110"}, "000", 2445, 2, "C", 20}
{{"111"}, "000", 10000, 0, "C", 22}
{{"100"}, "001", 3442, 20, "I", 2}
{{"101"}, "000", 2293, 9, "C", 11}
}

Input  Output  Correctness  Classifier
  01      0        right     {{"101"},"000",2908,10,"C",11}
  10      0        right     {{"110"},"000",2830,3,"C",20}
  11      0        wrong     {{"111"},"000",-97084,1,"C",22}
  00      1        right     {{"100"},"001",3266,21,"I",2}
  01      0        right     {{"101"},"000",2913,11,"C",11}
  01      0        right     {{"101"},"000",3424,12,"C",11}
  10      0        right     {{"110"},"000",2722,4,"C",20}
  10      0        right     {{"110"},"000",3265,5,"C",20}
  11      0        wrong     {{"111"},"000",-138765,2,"C",22}
  00      1        right     {{"100"},"001",2838,22,"I",2}

right 8, wrong 2, no response 0
cycle count 10, #clfs below thres 1

--- Generation #11 ---
```

```
{
{{"110"}, "000", 2793, 5, "C", 20}
{{"101"}, "00.", 10000, 0, "M", 23}
{{"100"}, "001", 2838, 22, "I", 2}
{{"101"}, "000", 2505, 12, "C", 11}
}

Input  Output  Correctness  Classifier
  10     0        right     {{"110"},"000",3324,  "C",20}
  10     0        right     {{"110"},"000",3766,  "C",20}
  10     0        right     {{"110"},"000",4134,  "C",20}
  01     1        wrong     {{"101"},"001",-9795 1,"M",23}
  10     0        right     {{"110"},"000",4181,   C",20}
  11     no response
  10     0        right     {{"110"},"000",4218,   C",20}
  11     no response
  00     1        right     {{"100"},"001",2262,23,"I",2}
  00     1        right     {{"100"},"001",2882,24,"I",2}

right 7, wrong 1, no response 2
cycle count 10, #clfs below thres 1

--- Generation #12 ---

{
{{"110"}, "000", 3337, 10, "C", 20}
{{"111"}, "001", 10000, 0, "C", 24}
{{"100"}, "001", 2882, 24, "I", 2}
{{"101"}, "000", 1146, 12, "C", 11}
}

Input  Output  Correctness  Classifier
  00     1        right     {{"100"},"001",3398,25,"I",2}
  11     1        right     {{"111"},"001",5163,1,"C",24}
  11     1        right     {{"111"},"001",4949,2,"C",24}
  11     1        right     {{"111"},"001",5119,3,"C",24}
  10     0        right     {{"110"},"000",3031,11,"C",20}
  00     1        right     {{"100"},"001",3069,26,"I",2}
  00     1        right     {{"100"},"001",3554,27,"I",2}
  01     0        right     {{"101"},"000",1550,13,"C",11}
  00     1        right     {{"100"},"001",3736,28,"I",2}
  01     0        right     {{"101"},"000",2192,14,"C",11}

right 10, wrong 0, no response 0
cycle count 10, #clfs below thres 0

--- Generation #13 ---

{
{{"110"}, "000", 2050, 11, "C", 20}
{{"111"}, "001", 3205, 3, "C", 24}
{{"100"}, "001", 3455, 28, "I", 2}
{{"101"}, "000", 2192, 14, "C", 11}
}

Input  Output  Correctness  Classifier
  11      1       right     {{"111"},"001",3667,4,"C",24}
  11      1       right     {{"111"},"001",4052,5,"C",24}
  01      0       right     {{"101"},"000",2559,15,"C",11}
  11      1       right     {{"111"},"001",4119,6,"C",24}
  01      0       right     {{"101"},"000",2970,16,"C",11}
  00      1       right     {{"100"},"001",2946,29,"I",2}
  00      1       right     {{"100"},"001",3452,30,"I",2}
  11      1       right     {{"111"},"001",3712,7,"C",24}
  10      0       right     {{"110"},"000",1911,12,'C",20}
  00      1       right     {{"100"},"001",3457,31,"I",2}

right 10, wrong 0, no response 0
cycle count 10, #clfs below thres 0
```

```
--- Generation #39 ---

{
{{"110"}, "000", 2496, 80, "C", 20}
{{"111"}, "001", 3524, 65, "C", 24}
{{"100"}, "001", 2487, 88, "I", 2}
{{"101"}, "000", 1710, 83, "C", 11}
}

Input  Output  Correctness  Classifier
  11      1       right     {{"111"},"001",3933,66,"C",24}
  01      0       right     {{"101"},"000",2315,84,"C",11}
  10      0       right     {{"110"},"000",2775,81,"C",20}
  11      1       right     {{"111"},"001",3800,67,"C",24}
  11      1       right     {{"111"},"001",4162,68,"C",24}
  10      0       right     {{"110"},"000",2974,82,"C",20}
  10      0       right     {{"110"},"000",3475,83,"C",20}
  00      1       right     {{"100"},"001",2196,89,"I",2}
  01      0       right     {{"101"},"000",2205,85,"C",11}
  01      0       right     {{"101"},"000",2835,86,"C",11}

right 10, wrong 0, no response 0
cycle count 10, #clfs below thres 0

--- Generation #40 ---

{
{{"110"}, "000", 2749, 83, "C", 20}
{{"111"}, "001", 2816, 68, "C", 24}
{{"100"}, "001", 1878, 89, "I", 2}
{{"101"}, "000", 2835, 86, "C", 11}
}

Input  Output  Correctness  Classifier
  10      0       right     {{"110"},"000",3288,84,"C",20}
  00      1       right     {{"100"},"001",2445,90,"I",2}
  00      1       right     {{"100"},"001",3034,91,"I",2}
  01      0       right     {{"101"},"000",2866,87,"C",11}
  11      1       right     {{"111"},"001",2713,69,"C",24}
  01      0       right     {{"101"},"000",3206,88,"C",11}
  01      0       right     {{"101"},"000",3668,89,"C",11}
  01      0       right     {{"101"},"000",4053,90,"C",11}
  10      0       right     {{"110"},"000",2582,85,"C",20}
  10      0       right     {{"110"},"000",3148,86,"C",20}

right 10, wrong 0, no response 0
cycle count 10, #clfs below thres 0

*** Done ***
```

As can be seen by the trace, the system picked up the solution in generation #12. Other runs of this program have resulted in the system being unable to obtain the solution until generation #20. However, by generation #20 the system usually has the solution. This run ended with the resulting rule base:

```
{{"110"}, "000", 3148, 86, "C", 20}
{{"111"}, "001", 1835, 69, "C", 24}
{{"100"}, "001", 1755, 91, "I", 2}
{{"101"}, "000", 3467, 90, "C", 11}
```

which is as desired.

Chapter 6

The Tic-Tac-Toe Application

6.1. Introduction

The main application to which LCS was applied for experimentation was the game of tic-tac-toe. In this application, LCS (which played X) always played second against a random player (which played O). Payoff was set up so to reward it positively for making legal moves (placing its token in an unoccupied square) and to reward it negatively for making illegal moves (placing its token in an occupied square). Also, LCS was rewarded positively for winning or drawing games and rewarded negatively for losing games. Thus, the goal of the experiment was to start the system off with a randomly constructed rule base, producing in the end a rule base which enabled the system to win the majority of games against the random player. Hopefully, this rule base would also [illegible] the system to play an optimal game - it would always make a correct [illegible] never losing against a perfect opponent.

This chapter describes the tic-tac-toe application giving a breakdown of the implementation as well as explanation for the various design decisions made. This is followed with a discussion of the results from the experimentation with tic-tac-toe. Pseudo code for the tic-tac-toe application can be found in appendix A2.

6.2. Representation

In designing the tic-tac-toe application, one of the first problems to resolve is to determine what a message and a classifier are to represent.

6.2.1. Messages

For this application, it was decided that an input message would represent a board configuration and an output message would represent a move. Thus, when it is the system's turn to move it will be given a message which indicates the current board configuration and will return a message which indicates its move.

This dual representational property of messages means that input messages are of a different form than output messages. The first symbol of a message is reserved to facilitate the distinction between the two kinds of messages. If this symbol is a "1", then the message is an input message, otherwise the symbol must be a "0" and the message is an output message.

For an input message, the remainder of the symbols in the message must encode a board configuration. To accomplish this, it was decided that a board would be encoded linearly (square one, square two, ... , square nine) into the message. The enumeration of the squares of the board are as follows:

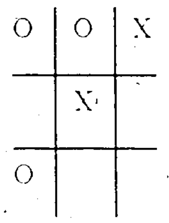
1	2	3
4	5	6
7	8	9

Since a square can have three states (blank, X, or O), two symbols are required to represent a square. The encoding scheme chosen for this representation is:

State	Symbol 1	Symbol 2
blank	0	0
X	1	0
O	1	1

Notice that with this encoding scheme, the concept of an occupied square can be represented within a condition string by the symbols "1#" (since that will match either an X or O but not a blank square). If the encoding scheme had been such that X was 10 and O was 01, then there would be no way to represent the concept of an occupied square. Thus, one can see the importance of carefully chosing the encoding scheme as a poor choice can lead to the inability to represent certain concepts and consequently can affect the learning potential of the system. This is one of the representational problems inherent in learning classifier systems.

Having nine squares with each square being represented by two symbols gives a total of 18 symbols required to encode a board configuration. Add to that one additional symbol to indicate that it is an input message and the result is an input message that is 19 symbols long. Thus, the board configuration



would result in the input message

1111110001000110000.

Since all messages must be of the same length, output messages are also 19 symbols long. After the initial "0" symbol (indicating that it is an output message), the remaining 18 symbols of an output message have to indicate a move. Since there are only nine possible moves that can be made (squares one to nine), only four symbols are required to represent a move if a move is just encoded into its binary equivalent (i.e. a move to square 5 would be encoded into 0101). Thus, out of the 18 symbols available to indicate a move, only the last four are used - the other symbols are set to "0". Thus, a move to square 5 would result in the output message 0000000000000000101.

and a move to square 8 would result in the output message 0000000000000001000.

6.2.2. Classifiers

The structure of the input and output messages influenced the structure of the classifiers. Classifiers were designed so that their condition part matches a board configuration and their action part produces a move. In this design, the condition part of a classifier contains only one condition string. This condition string conforms to the structure of an input message in that it is 19 symbols long with the first symbol always being the symbol "1". The remaining symbols are grouped together in consecutive pairs with each pair corresponding to a square on the board (just as in input messages). A pair is allowed to assume one of the following values: "00", "10", "11", "#0", "1#", or "##". A pair is not allowed to assume "01" because that combination is not used in the encoding of a board configuration into an input message. Consequently, the appearance of that combination in a condition string would result in a classifier that would not match any input messages — hence it is disallowed. The combination "#1" is disallowed because it is redundant with the combination "11". "#1" matches the encodings "01" and "11", but "01" is not used so it effectively matches only "11". Thus whenever the pair "#1" occurs in a condition string, it could be replaced by the pair "11" without making any difference. Hence "#1" is redundant and consequently disallowed. The combination "0#" is also disallowed for similar reasons.

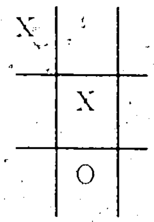
Some examples of condition strings are: 110################ which will match any board configuration that has an X in the first square, 11111############## which will match any board configuration that has an O in the first and second squares, and 1######111100###### which will match any board configuration that has an O in the fourth and fifth squares and a blank in the sixth square.

Action strings were designed to conform to the structure of output messages. As such, the first 15 symbols of an action string are always all "0"s and the last four symbols of an action string contain the move to be made. These four symbols can assume the same set of values that the last four symbols on output messages can. Some examples of action strings are 0000000000000000101 which produces an output message indicating a move to square 5, and 0000000000000000111 which produces an output message indicating a move to square 7.

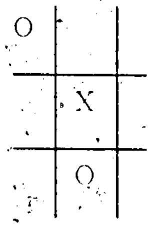
A classifier contains a single condition string in its condition part and, of course, a single action string in its action part. Thus, a classifier indicates a move to be made in response to the presence of any member of a set of board configurations. If a board configuration appears that is a member of the set of configurations defined by the condition string (the board configurations which match the condition string) then the action string indicates the move to be made in response to that board configuration. [illegible]mple, the classifier

11#0000001000001100 / 0000000000000000011

indicates a move to square 3 for either of the boards



or



Note, however, that the first board would not appear in a real game since the opponent always moves first.

6.2.3. Initialization

Initialization of tic-tac-toe basically involves setting the system parameters and constructing the initial rule base. The system parameters are set through the following sequence of calls:

```
seed(125);
msgSize(19);
clfPostLvl(1);
maxNumConds(1);
bidCoeff(0.5);
maxFireBid(4);
fireProb(95);
taxRate(0.002);
maxFireTax(19);
condBndry(0,1,1,"1");
condBndry(1,2,6,"00 10 11 #0 1# ##");
condBndry(3,2,6,"00 10 11 #0 1# ##");
condBndry(5,2,6,"00 10 11 #0 1# ##");
condBndry(7,2,6,"00 10 11 #0 1# ##");
condBndry(9,2,6,"00 10 11 #0 1# ##");
condBndry(11,2,6,"00 10 11 #0 1# ##");
condBndry(13,2,6,"00 10 11 #0 1# ##");
condBndry(15,2,6,"00 10 11 #0 1# ##");
condBndry(17,2,6,"00 10 11 #0 1# ##");
actBndry(0,15,1,"000000000000000");
actBndry(15,4,9,"0001 0010 0011 0100 0101 0110 0111 1000 1001");
initStrength(100000);
rplcThres(300);
genCondMtch(13);
genLife(5);
coverCond("1##################");
crossoverProb(95);
mutationProb(95);
```

The values for some of these parameters have already been discussed in the previous sections on messages and classifiers. Of the remaining parameters, some had values that were easily determined while others had values that were determined by experimental feedback. Those requiring this feedback will be discussed in the section on experimental results. The others are discussed now.

The seed parameter is just an initial random number for the random number generator. The random number 125 was chosen. The `clfPostLvl` is set to 1 so that the system will only generate one move in response to a board. `fireProb` is set to 95 so that, for the most part, the "best" classifier will be chosen to fire. `rplcThres` is arbitrarily set to 300. `coverCond` is set to 1################## so that only input messages will be covered. Finally, `crossoverProb` and `mutationProb` are set to 95 so that, for the most part, the highest strength classifiers will be used for crossover and mutation.

In addition to setting system parameters, initialization also involves setting some application parameters. For tic-tac-toe, these include some payoff parameters, a rule base size parameter, and some parameters that provide run information. The payoff parameters are as follows:

Payoff Item	Amount
Legal Move	100
Illegal Move	-1000000
Win Game	10800
Lose Game	-16400
Draw Game	6000

As expected, the legal move payoff is the payoff issued when a legal move is made; the win game payoff is the payoff issued when the system wins a game; and so on. Admittedly, some of these parameter values look a bit odd, but the explanation as to how these values were obtained can be found in the section on experimental results. The manner in which payoff is issued for each of these payoff items (what payoff group receives the payoff) is discussed in the next section.

The rule base size parameter indicates the number of rules in the rule base. The size of the rule base remains constant throughout the run. This parameter is set to a value of 120. Explanation as to how this value was obtained is provided in the section on experimental results.

The parameters that provide run information consist of the generation count parameter, the min cycles/generation parameter, the max cycles/generation parameter, and the replace size parameter. The generation count parameter indicates the number of generations to be produced in the run. The run terminates after the last generation has been produced. Usually this parameter is set to a value of 300 or higher. The min and max cycles/generation parameters set boundaries on the minimum number of cycles that must be performed, and the maximum number of cycles that can be performed, in a generation. One cycle refers to one iteration of the basic execution cycle. These parameters are set to 1150 for the min and 2700 for the max. The replace size parameter indicates the minimum number of replaceable rules that must be present in order to produce the next generation. A replaceable rule is a rule whose strength is below the replacement threshold. This parameter is set to a value of 10.

So, in a generation, the basic execution cycle is performed for at least 1150 iterations. When these iterations have completed, if the number of replaceable rules is below 10 then further iterations are performed until at least 10 replaceable rules are present. At this point, the basic execution cycle is stopped and the next generation of rules are produced. If, in the process of iterating, the number of cycles performed should exceed 2700 then the basic execution cycle will be stopped and the next generation will be produced. This ensures that the run will complete in a finite amount of time.

Explanation as to how the values were obtained for all these run parameters is given in the section on experimental results. As well, further discussion about some of these parameters is also given.

Construction of the initial rule base is accomplished by randomly generating a condition and action string for each rule. The condition string is initially set to the symbol "1". Then, 9 pairs of symbols are appended to it, where each pair is randomly chosen from amongst the pairs: "00", "10", "11", "#0", "1#", and "##". The action string is initially set to "000000000000000" and one of the symbol combinations "0001", "0010", ... , "1001" is randomly chosen to be appended to it. The rule is given an initial strength of 100000 and an initial fire count of 0. 120 rules are generated constituting the rule base.

6.2.4. The Basic Execution Cycle

After initialization has been performed, the tic-tac-toe application begins iterating for the number of generations indicated by the generation count parameter. In each iteration, the basic execution cycle is first performed for some number of iterations (as discussed in the previous section) followed by the production of the next generation. This section discusses the basic execution cycle. Production of a new generation is discussed in the next section.

Each iteration of the basic execution cycle begins by first retrieving the opponent's move. This involves either generating a random legal move or prompting a human opponent for the move. After the move has been retrieved and the board has been updated, the board is checked to determine whether the opponent has won or whether a draw game has been reached. If either of these is the case, then the appropriate payoff is issued (either lose or draw game payoff) and a new game is started thus completing the current iteration. Otherwise, it is the system's turn to move.

Retrieving the system's move involves first constructing an input message that represents the current board. This involves first standardizing the board and then encoding this standardized board into an input message as was described in the section on messages. The reason the board is standardized is so that the system will not have to learn about board symmetry. Two boards in which one can be derived through flips and rotations of the other, will appear as the same board to the system (i.e. they will both be encoded into the same input message). Standardization of a board is accomplished by rotating and flipping the board through the 8 various positions (rotate 4 times, flip, and rotate another 4 times; where each rotation is by 90 degrees clockwise), computing a key for each position, and then encoding the position with the highest key into the input message. A key is computed for a board position by treating each square as a base 3 digit (blank=0, X=1, O=2) and thus the entire board as a base 3 number (with square 9 as the most significant digit and square 1 as the least significant digit).

Once the input message has been constructed, it is passed to the system through the face system function `message`. Next, the face system function `nextcycle` is called, producing the output message. This output message is then retrieved from the face variable `messlist` and used to update the standardized board. After updating, the standardized board is destandardized bringing it back to the configuration with which the opponent is familiar. The board is then checked to see if the system has made a legal move. If it has not, then illegal move payoff is issued and a new game is started thus completing the current iteration. Otherwise, legal move payoff is issued and the board is checked to determine if the system has won or if a draw game has been reached. If either of these have occurred, then the appropriate payoff is issued (either win or draw game payoff) and a new game is started thus completing the current iteration. Otherwise, the system has finished making its move and the current iteration is completed.

Note that after `nextcycle` has been called, it is possible that no output message may exist. This occurs when the input message does not satisfy the condition part of any rule. In other words, the system has no response to the current board. When this occurs, a new game is started thus completing the current iteration.

When payoff is issued, it is issued to either the group of currently active rules (payoff group number zero), or the group of rules which were active at some point during the current game. The latter group is known in the application as payoff group number one. This group is emptied (rmvPOGrp) at the start of each game and active rules are added to this group (addToPOGrp) in each iteration of the basic execution cycle. Payoff for legal and illegal moves are issued to group number zero. Payoff for won, lost, or drawn games are issued to group number one.

6.2.5. Producing New Generations

After the iterations of the basic execution cycle have completed, the next generation of rules are produced. This involves first setting the generalization, cover, and crossover rates and then calling the face system function nextgeneration. When this function has completed, a new generation of rules will exist in the rule base and the next iteration of the generation cycle can begin.

The values that the generalization, cover, and crossover rates are set to are dependent upon the state of the execution. At the start of the run, there does not exist any good rules from which to form generalizations. Thus, the generalization rate is set to 0. There are, however, many board configurations that the system has never seen before and most likely is unable to respond to. Consequently, the cover rate is set to 100. Of the remaining new rules to be produced after cover has completed, 40 percent of them are produced by crossover (crossover rate is set to 40) and the rest by mutation. Note that the important criteria in determining these values for the rates was not based so much on the fact that it was the start of the run, but rather on the fact that there was a high number of "no responses". Thus, these rates come into effect whenever the "no response" count is high or, as is expressed in the code, whenever the "no response" count for a generation is greater than 37. This figure was arrived at by examining the results of previous runs and observing the "no response" count when the system had begun to acquire some good rules. This occurred around a "no response" count of 37.

When the "no response" count is 37 or less, the rule base will likely contain some good rules from which generalizations can be formed. Consequently, the generalization rate is set to 30 so that some generalizations will be created while still leaving the opportunity for the other discovery operators to create new rules (70 percent of the new rules will come from the other operators). Again, the cover rate is set to 100 in order to cover any board configurations to which there was "no response". Also, the crossover rate remains at 40.

6.3. Experimental Results

This section describes the results that were obtained from the experiments conducted on tic-tac-toe. The setting of some of the parameters are first discussed followed by a discussion of the results that were obtained using those settings. Finally, some of the problems encountered in the course of obtaining those results are discussed.

6.3.1. Setting Parameters

The setting of some of the parameters has already been discussed in previous sections. What remains to be discussed is the setting of those parameters that required experimental feedback to determine their values. Again these parameters can be categorized into system and application parameters. The system parameters are discussed first followed by the application parameters.

The first system parameter to discuss is `initStrength`. As §4.3 indicates, the initial strength of new rules must be high enough to outbid established rules in order to give the new rules the opportunity to prove themselves. In the initial runs of the system, this problem was not realized and consequently the initial strength was set too low. This resulted in the system retaining only mediocre rules. When this problem was discovered, a new higher initial strength had to be determined. This new initial strength was based on the highest strength good rule.

Roughly speaking, a good rule's strength will stabilize when the average payoff it receives equals the average bid it makes; or in other words:

bid_coeff * specificity * strength / (max_fire_bid + 1) = avg_payoff

A rule that wins a game every time it is used will have the highest avg_payoff (average payoff). This will be a payoff of 100 for the legal move, plus approximately 2700 for winning the game (this figure will be discussed in the application parameters discussion). With a bid_coeff of 0.5 and a max_fire_bid of 4 (these values will be discussed shortly), this gives:

0.5 * specificity * strength / 5 = 2800

The highest strength good rule will have the lowest specificity, according to the equation. The rule with the lowest specificity that always wins the game it is used in will have a condition part that is only concerned with three squares on the board. These three squares will form a line (either diagonally, horizontally, or vertically) and the condition part will indicate that two of these squares have to contain X's and the third can contain either an X or be blank. Note that this third square has to be blank because if an X were there then there would be three X's in a line and the game would have been over. Since an X is encoded as "10", blank or X is encoded as "#0", and all condition strings start with a "1", the specificity of such a rule is 6/19. Putting this figure into the previous equation gives a strength of 88667. Thus, the `initStrength` is set to the rounded up value of 100000.

The next system parameters to discuss are `taxRate` and `maxFireTax`. The value for the `taxRate` should be such that new rules placed into the system will survive long enough to come across any board configurations to which they are applicable. The question is, how many basic execution cycles does this represent? To answer this, the total number of different board configurations that the system could possibly "see" was first determined. If new rules survived for that many cycles, then there would be a good chance that they would come across any boards to which they are applicable. The figure calculated was 2781 but the calculations involved did not account for the fact that the boards presented to the system are standardized. Accounting for this would reduce this figure. However, it is also true that board configurations do not appear with equal frequency. Start game configurations appear more frequently than end game configurations. This suggests an increase in the number of cycles that new rules should survive for. Taking both of these facts into consideration, the nun [illegible] f cycles that new rules should survive for was ballparked at 2700. Initially his fi[illegible]e was ballparked at lower values but experimental results (which shall be discussed later) proved them to be inadequate. Note that, for this figure, it is better to err on the side of being too large, as it is better that new rules come across too many ~~boards~~ than too few. This was also taken into consideration. Since new rules start out with an initial strength of 100000 and can be replaced when their strength falls below 300, the `taxRate` can be calculated as follows:

$$300 = 100000(1 - taxRate)^{2700}$$

This gives a `taxRate` of approximately 0.002 which is the current setting.

The value used for the `maxFireTax` was also determined from experimental feedback. Established rules should experience smaller taxation so that when new rules initially outbid them, taxation does not decrease their strength by much. This was discussed in §4.3. Exactly how small a taxation they should experience, though, was determined experimentally. Initially the maxFireTax was set to 9, but that proved to be too small as some established rules experienced too much taxation during the period that new rules were outbidding them. Consequently, those rules were incorrectly lost. After trying a few more values, the current value of 19 was arrived at. Note that, for this parameter, it is better to err on the side of being too large, as a larger value just means that it takes longer to get rid of useless rules while a smaller value might mean the unjustified loss of some established rules.

The system parameters bidCoeff and maxFireBid are set to 0.5 and 4 respectively. maxFireBid was determined experimentally. Initially smaller values were used but this resulted in too long of a stabilization period for new rules, and consequently the unjustified loss of some established rules. Eventually the value of 4 was arrived at. The bidCoeff was always set such that

bidCoeff / (maxFireBid + 1) = 0.1

The 0.1 value was just arbitrarily chosen so that rules that fired more than maxFireBid times would effectively use a bid coefficient of 0.1 in their bid calculations. This equation sets the bidCoeff at 0.5.

Finally, the last system parameters to discuss are genCondM-ch and genLife. genCondMtch was determined experimentally. If this parameter is too small then over generalizations result. If it is too large then weak generalizations result (generalizations that are not that much more general than the rules they were formed from). Initially a value of 11 was used but the rules which resulted tended to be a bit over generalized. Consequently this led to the current value of 13. genLife is currently set to 5. Initially, higher values were tried, but a poor generalization would just take too long to stabilize resulting in the loss of some of the rules from which it was formed. This led to the current setting.

Turning to the application parameters, the first ones to be discussed are the payoff parameters. The legal move parameter was chosen to be a small value so that a rule receives some reward for making a legal move but will not be able to flourish by making legal moves alone. Consequently this parameter was arbitrarily set to 100. The illegal move parameter was set to -1000000 so that any rule which makes an illegal move will have its strength dropped below the replacement threshold.

The win game payoff was determined by first deciding the strength range within which rules produced by cover should stabilize. To do so, only rules that always win games were examined, as those rules will have the highest stabilizing strengths. It was decided that these rules should stabilize within the 25000 to 30000 range as this was sufficiently high enough so that it would take a fair number of cycles for taxation to drop their strengths below the replacement threshold of 300. Within this range, a strength of 28000 was arbitrarily chosen to be the stabilizing strength. Because rules produced by cover have a specificity of one, and the `bidCoeff/(maxFireTax+1)` ratio is 0.1, the bid placed by a winning cover rule will be

$$0.1 * 1 * 28000 = 2800.$$

As was discussed previously, the payoff that this rule receives should equal this figure. The rule receives a payoff of 100 for making a legal move. Thus, it should receive the difference of 2700 for winning the game. Since the system can make up to 4 moves to win a game, the payoff for winning a game is set to 4 * 2700 which is 10800.

The payoff for drawing a game was chosen to be approximately half that of winning. This led to the rounded up figure of 1500 for each system move in a draw game which gives 4 * 1500 or 6000 as the draw game payoff.

The payoff for losing a game was determined experimentally. This payoff is always set to a negative value indicating that a loss is undesirable. If this parameter is set to a value that is too large (in magnitude) then some good rules may be incorrectly lost. Payoff for losing a game (as with winning or drawing) is distributed equally amongst all the rules that participated in the game. The loss of a game means that at least one of the rules produced an incorrect move, but which rule is at fault is unknown. Therefore all the rules receive a small negative feedback. As these rules participate in other games, the rule which was at fault should receive this negative feedback more frequently as it is more likely to be involved in lost games. This negative feedback should accumulate, eventually causing the rule's strength to fall below the replacement threshold. If the payoff for losing a game is too large, then the frequency with which some good rules receive positive feedback (from winning or drawing) may not be enough to combat the accumulative effect of the negative feedback they receive when they are involved in games that the faulty rule is involved in. Consequently these rules will be wrongly lost. On the other hand, if the payoff for losing is too small, then an opposite effect occurs. A faulty rule does not necessarily lose every game it is involved in as the opponent plays randomly. Thus, a faulty rule can and most likely does receive some positive feedback. If the payoff for losing is too small, then this positive feedback may be enough to cancel the effect of the negative feedback and subsequently enable the rule to survive. The value of this payoff parameter was determined by initially issuing a payoff of -3100 to any rule that was involved in a lost game. This gave a lose game payoff of 4 * -3100 which is -12400. This value, however, proved to be too small, and after some more trials the current value of -16400 was arrived at.

Of the remaining application parameters to discuss, the generation count parameter shall be first. This parameter is currently set to 300. Initially smaller values were [illegible] but they did not let the system run long enough to give the desired results. The [illegible] ax cycles/generation parameters are set to 1150 and 2700 respectively.

Th[illegible] purpose of the min cycles/generation parameter is to force the basic execution cycle to be performed for a minimum number of iterations. This allows the strengths of new rules to stabilize so that when it is time to produce a new generation, the strengths of the rules will more accurately reflect their usefulness. The calculation of this parameter was done by considering a new rule which is not applicable to any legal board configurations (a legal board configuration always has the number of X's and O's differing by at most one). Such a rule would start out with a strength of 100000 and should have a low strength when it comes time to produce a new generation. This low strength was chosen to be 10000. Thus, the number of cycles involved in going from 100000 to 10000 is calculated by:

$$10000 = 100000(1 - 0.002)^{cycles}$$

This gives a cycle count of 1150 which is the current setting for the min cycles/generation parameter.

The purpose of the max cycles/generation parameter is to ensure that the basic execution cycle is performed for a finite number of iterations in each generation. This parameter was just set to the number of cycles that new rules should survive for, which is 2700 as was described in the discussion on the tax parameters. Note that this maximum was never reached in any of the generations for any of the runs.

The next application parameter to discuss is the rule base size parameter. This parameter was determined experimentally. Initially, a value of 200 was used but this proved to be too large as too many useless rules would be identified and replaced in each generation resulting in the creation of too many new rules. When a rule base has too many new rules, an established rule in the base has to contend with a lot of new competitors who initially take away that rule's ability to fire and consequently the positive feedback that it would normally receive. A lot of new competitors means that the established rule is "starved" for a long period of time during which it is also experiencing taxation. This leads to the problem discussed in §1.3 which could result in the loss of the established rule. This behavior was observed when the rule base size was set to 200. Consequently lower values were tried eventually giving the current value of 120.

Finally, the last application parameter to discuss is the replace size parameter. As was mentioned in §6.2.3, this parameter indicates the minimum number of replaceable rules that must be present in order to produce the next generation. As was just discussed, it is undesirable to have too many new rules present in the rule base. Thus, this parameter is set to a low value of 10.

As the reader has probably noticed, setting all the parameters can be a tedious and time consuming process. This is one of the drawbacks to using this system. Also, the adjusting of these parameters, as well as the choice for the domain representation, introduces some domain dependent knowledge into the system. This affects, to some extent, the domain-independent property of this system. However, this introduction of domain dependent knowledge is restricted to only the parameters and the representation choice and does not enter into the internal workings of this system. Thus, the code for this system is completely domain-independent.

6.3.2. Results

Setting the system and application parameters to the values described above gives the results shown in figures 6.1 and 6.2. Figure 6.1 shows, for each generation, the percentage of the games won, lost and drawn of all the games that were completed in that generation. Figure 6.2 shows, for each generation, the percentage of the board configurations "seen" by the system, in that generation, that resulted in an illegal move or a "no response".

By examining the graph in figure 6.1, one can see that in about the first 50 generations the system is in more or less of a chaotic state as it is acquiring knowledge about legal moves and any games that it completes is effectively due to random playing on its part. This can also be observed in the first 50 generations of figure 6.2 in which the system is unable to respond to a large portion of the boards presented to it. This, however, is a diminishing trend as by generation 80 the percentage of "no responses" has dropped to a few percent where it remains. The illegal move plot is similar although the percentage of illegal moves in the initial generations is significantly smaller compared to the "no responses". This is to be expected as initially the system is dealing with mostly opening board configurations as opposed to end game board configurations. In opening board configurations the majority of the squares on the board are unoccupied and thus a square randomly chosen to move to is likely to be unoccupied resulting in a legal move. Consequently, the biggest stumbling block initially is coming across mid game board configurations that the system has never seen before and to which it is unable to respond.

After about generation 50, the system begins completing more of its games. The positive feedback it receives for winning begins to take effect as the system quickly learns to win the majority of the games it completes (see figure 6.1). Learning to draw a game rather than lose it is, however, a bit more complicated as the system does not firmly acquire this capability until about generation 190. By examining figure 6.1, one can see that shortly after generation 50 the draws are relatively high but quickly diminish from there. However, at the same time, the wins are increasing as the draws are decreasing. What is happening here is the system is learning that winning is better than drawing and some of the games which were previously being drawn are now being won. As the system proceeds from there, figure 6.1 shows that the draws and losses remain low, but intertwine around each other before finally separating with the draws above the losses around generation 190. The reason it takes so long for the system to learn to draw over losing is because of the lack of feedback for rules which address this. Of all the games played after the initial generations, only 15 to 20 percent of them result in either draws or losses. This means that new rules which are applicable to such games have a much smaller chance of being tried as the chances are slimmer that the board configurations to which they are applicable, will appear. Consequently, such rules receive feedback less often resulting in slower learning and hence the long period before draws separate above losses. Beyond generation 190, the system has firmly established the wins over the draws and the draws over the losses. This remains in effect to the end of the run.

Although the system performed quite well, there are still some anomalies that arose. The fact that the plots of the illegal moves and games lost didn't go down and remain at zero percent can be explained by the system giving preference to trying new rules over established ones. A new rule always has a chance of making an illegal move, or making an incorrect move which results in the loss of a game. Thus these plots will never go down and remain at zero percent as long as new rules keep appearing.

Another interesting phenomena is the inability of the system to complete a game after having completed several games in the past. This only occurs in the initial generations. Figure 6.1 displays this behavior at around generation 30 at which point the total number of games won, lost, and drawn (i.e. the total completed) is zero. This also corresponds with high "no response" which occurs around generation 30 in figure 6.2. The reason this occurs is because the system has lost all the rules which respond to the opening boards. Essentially, the system learns to play by first acquiring knowledge about winning end games. It learns to place the third X down that will give it three X's in a line. This knowledge is more easily retained as the corresponding rules receive high positive feedback (for winning). As the system acquires this end game knowledge, it begins to retain mid game knowledge as it now knows how to win (end game knowledge) given certain mid game configurations. Thus, these configurations also receive high positive feedback and consequently are retained. Eventually, this trend will continue to the opening game configurations and the system will retain knowledge about opening moves. Prior to this, though, it is possible to lose knowledge about opening moves, and does occur as can be witnessed by the graphs.

While the anomalies discussed so far have been explainable, there were some less explainable phenomena which did occur.

More detailed examinations of the runs revealed that occasionally a good rule would be lost which should not have been. In most of the cases this was due to taxation which dropped the rule's strength below the replacement threshold. The rule, however, should have been used frequently enough to receive enough positive feedback to keep it alive. Possibly, further fine tuning of the parameters could solve this.

Another related anomaly is the fact that the "no response" plot does not go down and remain at zero percent. The system should eventually be able to respond to all legal board configurations, however there always seems to be a small percentage of the boards to which it is unable to respond. This is due partly, if not entirely, to the previous problem of rules that are wrongly lost. When a good rule is lost, some of the board configurations to which it would normally respond become "no responses". This would account for some, if not all, of the "no responses".

The run which produced the graphs in figures 6.1 and 6.2 required a fair bit of computation time. This indicates that tic-tac-toe was not an easy problem for LCS. One of the reasons for this is that in tic-tac-toe, the feedback for wins, losses, and draws is delayed until the end of the game. When the system loses a game, at least one of the moves made during that game is responsible for the loss. Exactly which moves were the bad moves is not revealed to the system, the system must determine that by itself. If the feedback was more direct such that the system would receive some negative payoff when it makes a bad move, then the learning time (and hence the computation time) would be significantly reduced.

Playing against a random opponent also increases the learning time. The random opponent may let the system win or draw games that should have been lost. This leads the system to believe that some moves are good when in actuality they are not.

Thus the system must now "unlearn" these moves — a process that takes time.

In addition to this, the search space of possible classifiers for tic-tac-toe is quite large resulting in a fair amount of time to search though that space and arrive at the set of classifiers (classifier pool) which give the desired performance. Examining the number of legal values for each of the fields in the condition and action strings gives a total of approximately 90.7 million ($6^9 * 9$) possible classifiers.

Finally, the representation chosen for tic-tac-toe does not permit knowledge learned about one board pattern to be carried across to a similar version of the same pattern. For example, suppose the system had a classifier that stated if squares 1 and 2 contain an X and square 3 is blank then the system should place an X in square 3. There is no discovery operation which could produce a new classifier from that one which states the equivalent in terms of squares 4, 5, and 6. The system would have to relearn the same pattern in terms of squares 4, 5, and 6. The fact that it already has that knowledge in terms of squares 1, 2, and 3 would in no way speed up this process. This "redundant" learning increases the learning time of the system.

6.3.3. Some Problems Encountered

During the course of obtaining the results given previously, several problems were encountered and solved, some of which led to the enhancements described in chapter 4. Following is a description of some of these problems.

When the system was first constructed and tried, the only discovery operators in the system were crossover and mutation. Given an initial rule base that was randomly constructed, it was hoped that the system would be able to discover legal moves followed by the ability to win and draw games. However, when the system actually ran, it had trouble acquiring legal moves and completing games. Unfortunately, it was never able to overcome these problems and consequently never "got off the ground". This led to the incorporation of the cover operator.

When taxation was introduced into the system, it brought along the difficulty of determining the `taxRate` (and also the `maxFireTax`). As was mentioned in the discussion on the tax parameters, determining the number of cycles that new rules should survive for was a trial and error process. Before the value of 2700 was arrived at, lower values were tried. This resulted in higher tax rates. The effect of too high a tax rate can be observed in figures 6.3 and 6.4. The run that produced those graphs, had its `taxRate` increased to 0.01 and its `maxFireTax` reduced to 9. As can be seen by the graphs, the system never achieves any level of performance as the positive feedback that good rules receive is not enough to combat the high taxation. Consequently good rules are eventually lost and the system fails to retain most of what it has learned.

Another problem that was encountered was the long stabilization period of new rules as was described in §4.3. As was previously discussed, this could sometimes lead to the unjustified loss of some good rules. The bid modification enhancement solved this problem, but prior to it, runs produced graphs similar to those in figures 6.5 and 6.6. The run that produced these graphs had an identical setup to the run that produced figures 6.1 and 6.2 except that the bid modification was "turned off" (`bidCoeff` set to 0.1, `maxFireBid` set to 0). Comparing these graphs to those in figures 6.1 and 6.2, one can observe the slower learning that occurs without the enhancement. Figure 6.5 shows that the wins don't firmly established themselves in a high position until about generation 130, and by generation 300, the draws have still not separated above the losses. Figure 6.6 shows that the "no responses" do not drop down until about generation 130 and at generation 175 there is a significant peak in the "no responses". Closer examination of the run revealed that this peak is due to the loss of some good rules. Thus, as one can see, the bid modification enhancement provides a definite improvement in system performance.

6.4. Other Systems

To the author's knowledge, the only other learning classifier system to be constructed and applied to tic-tac-toe is that developed by David Slate [Sla86]. Although this work is currently unpublished, word of it was obtained through personal communication with Slate. In his system, he uses internal messages for tic-tac-toe. Thus, after his system receives an input message giving the current board configuration, several classifiers may fire in succession, each producing an internal message, before the final classifi[illegible]s and produces the output message. The action string of this final class[illegible]ontains a list of moves as opposed to a single move. Thus, an output message for his system contains a list of moves, and the output interface is "smart" enough to run through that list and choose the first legal move it encounters. Results from his experiments indicates that his system achieves fairly good performance in tic-tac-toe. It seems that his system develops a strategy for choosing squares. As execution proceeds, his system begins to produce output messages which have their list of moves ordered so as to try sequences of squares which form rows, columns, and diagonals. Thus, an output message might indicate to first try square 1, then 2, then 3, and so on. The development of this square choosing strategy is an interesting result.

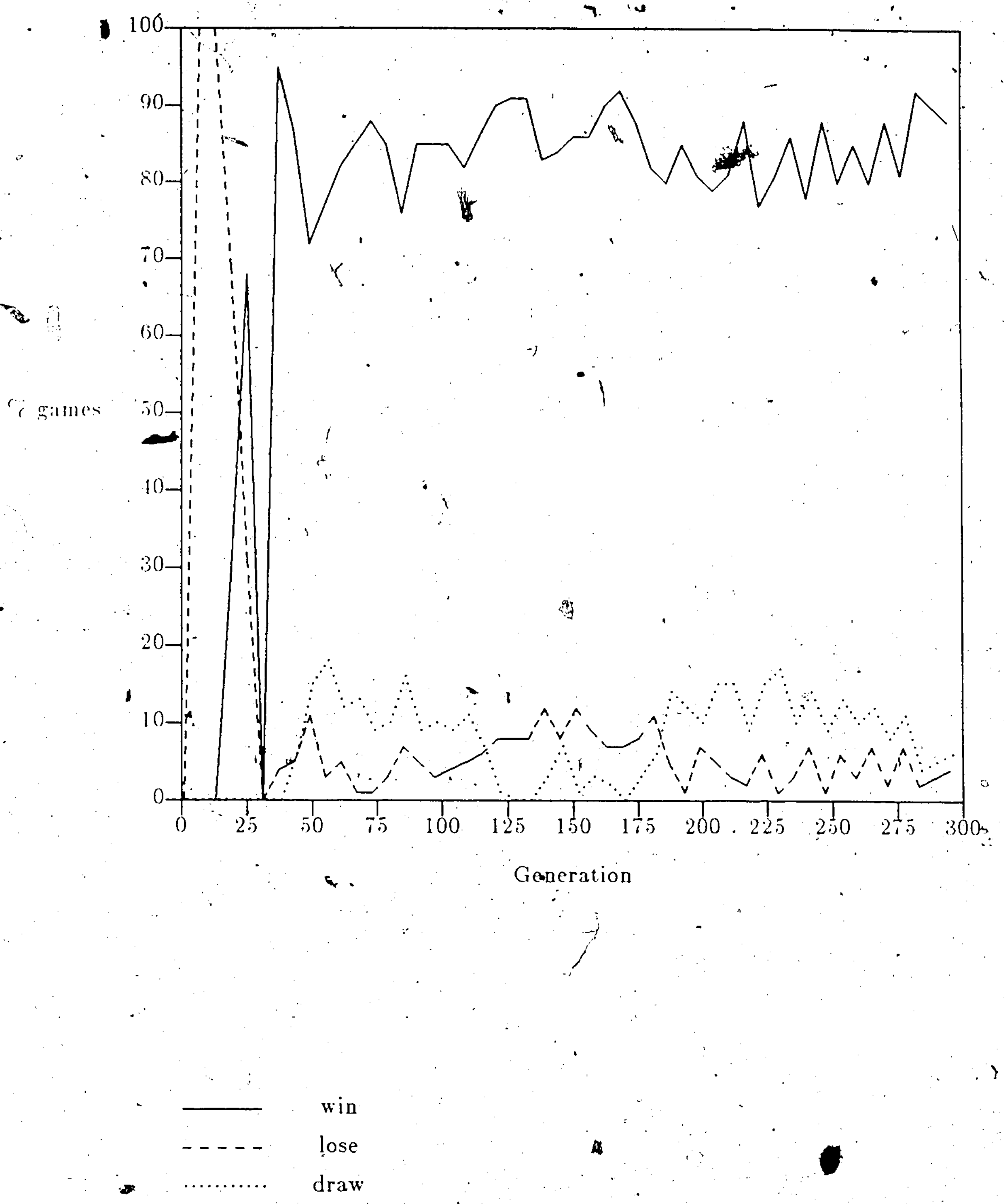
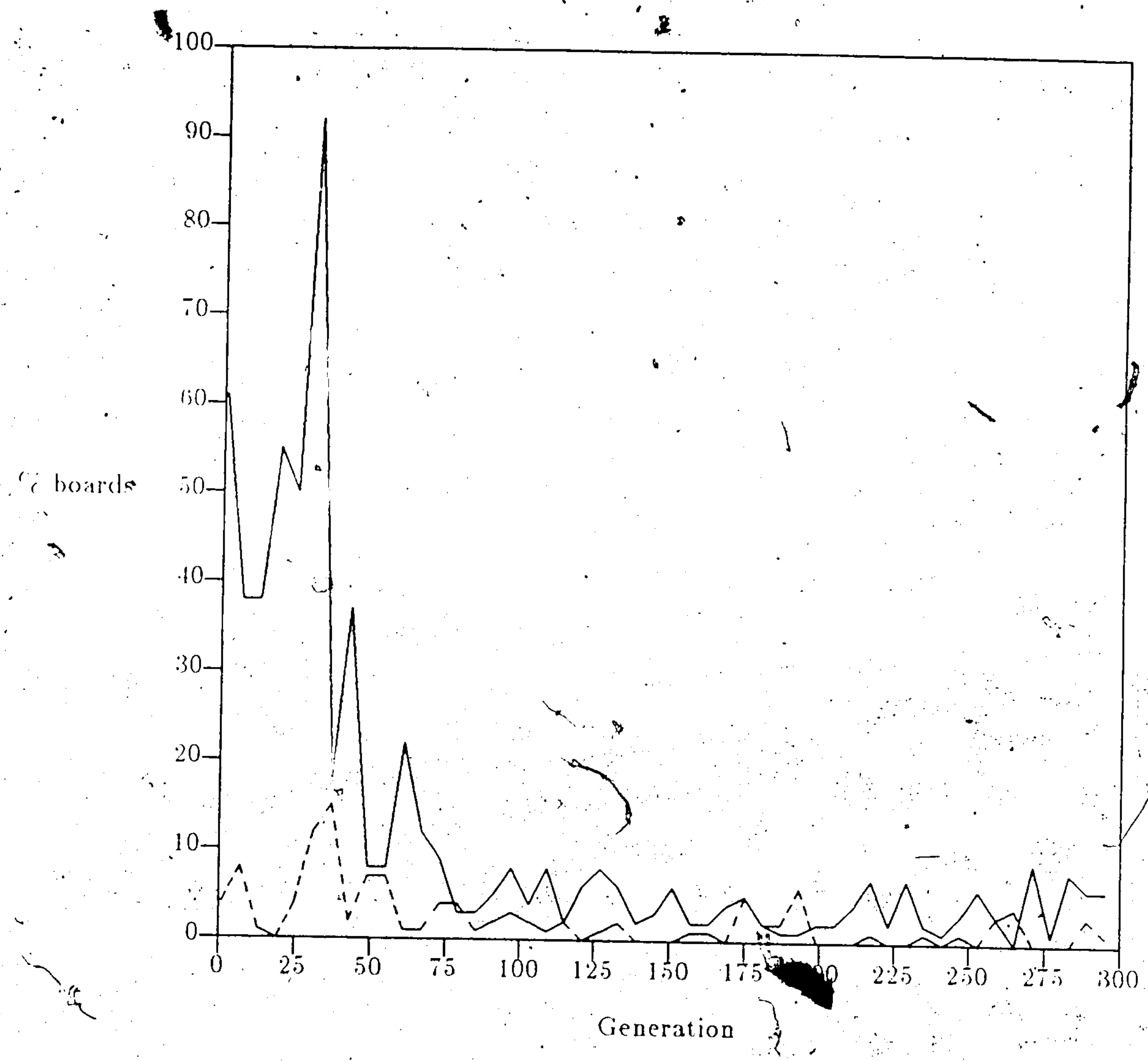


Figure 6.1 Wins, losses, and draws for good run.



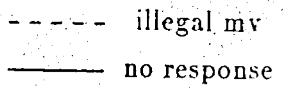


Figure 6.2 Illegal moves and no responses for good run.

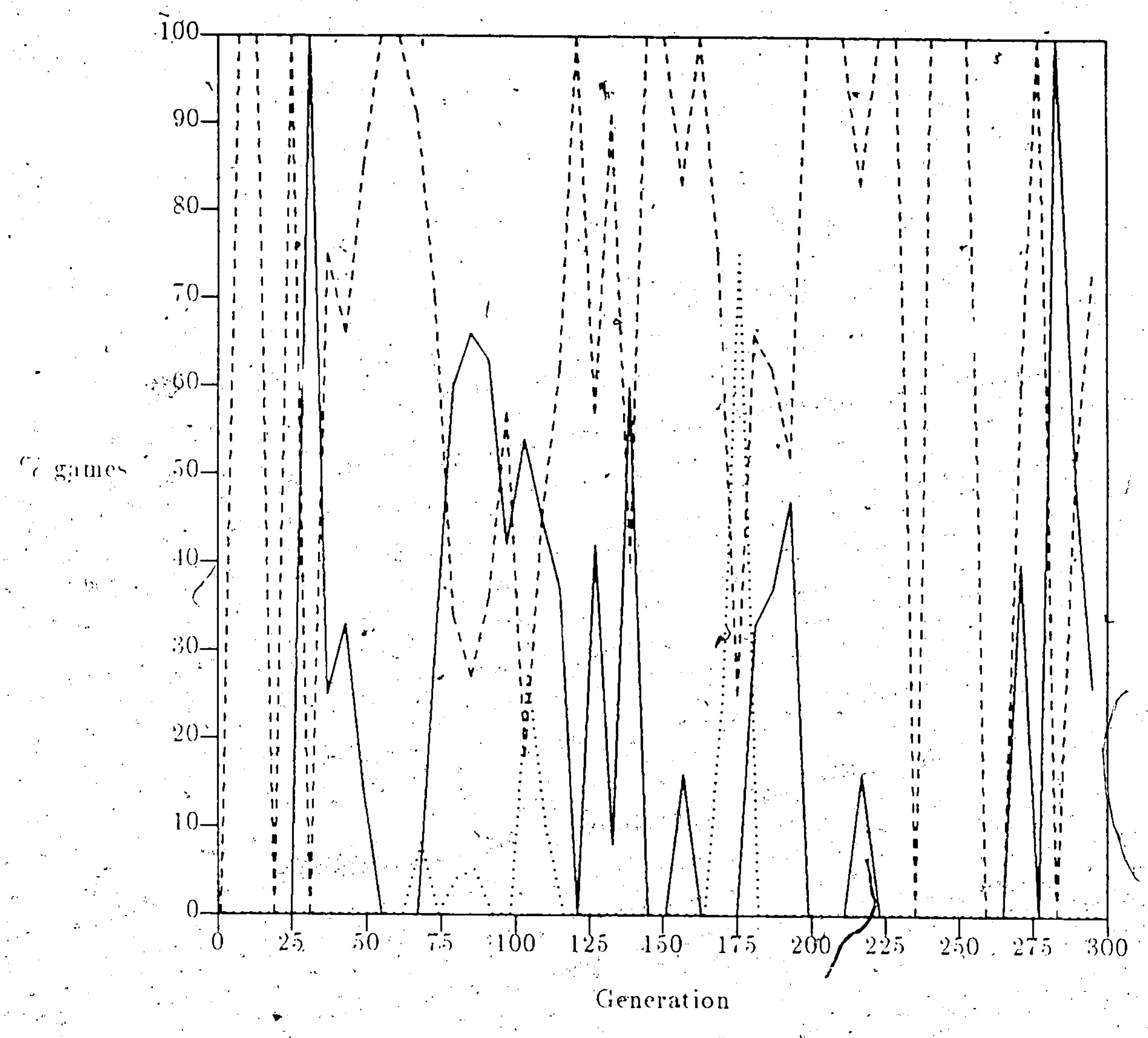
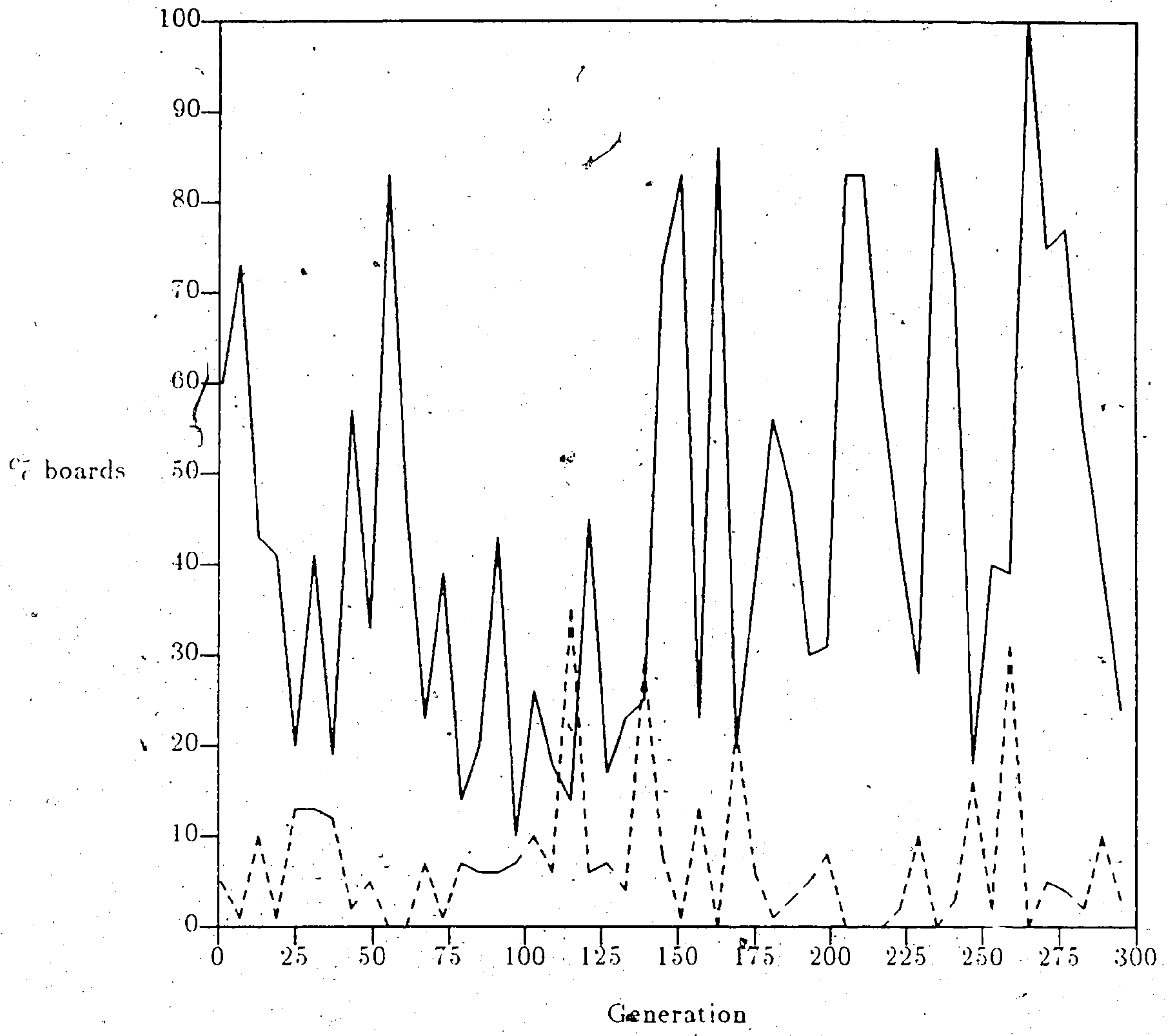


Figure 6.3 Wins, losses, and draws with high taxation.



- - - - - illegal mv
——— no response

Figure 6.4 Illegal moves and no responses with high taxation.

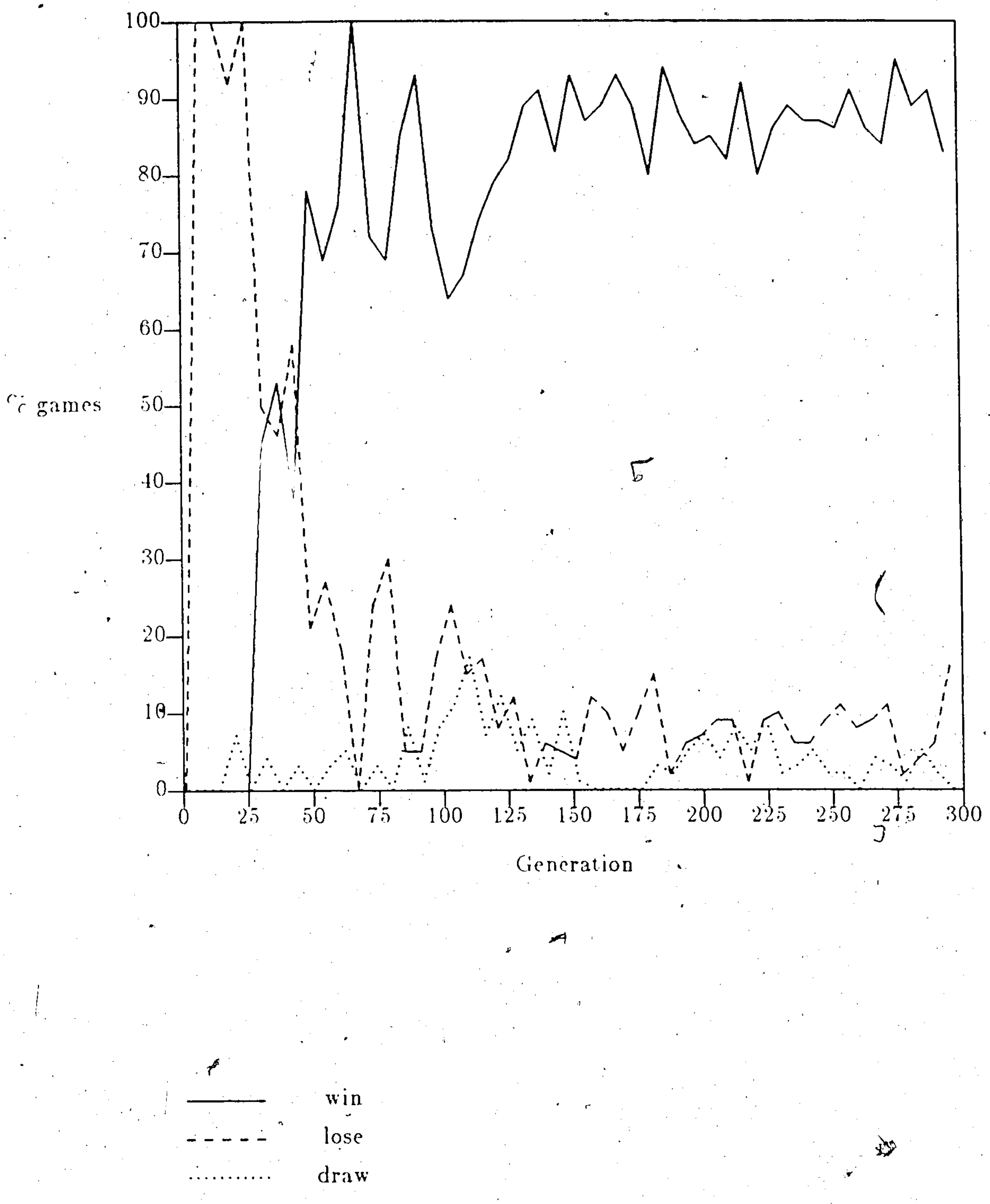


Figure 6.5 Wins, losses, and draws with no maxFireBid.

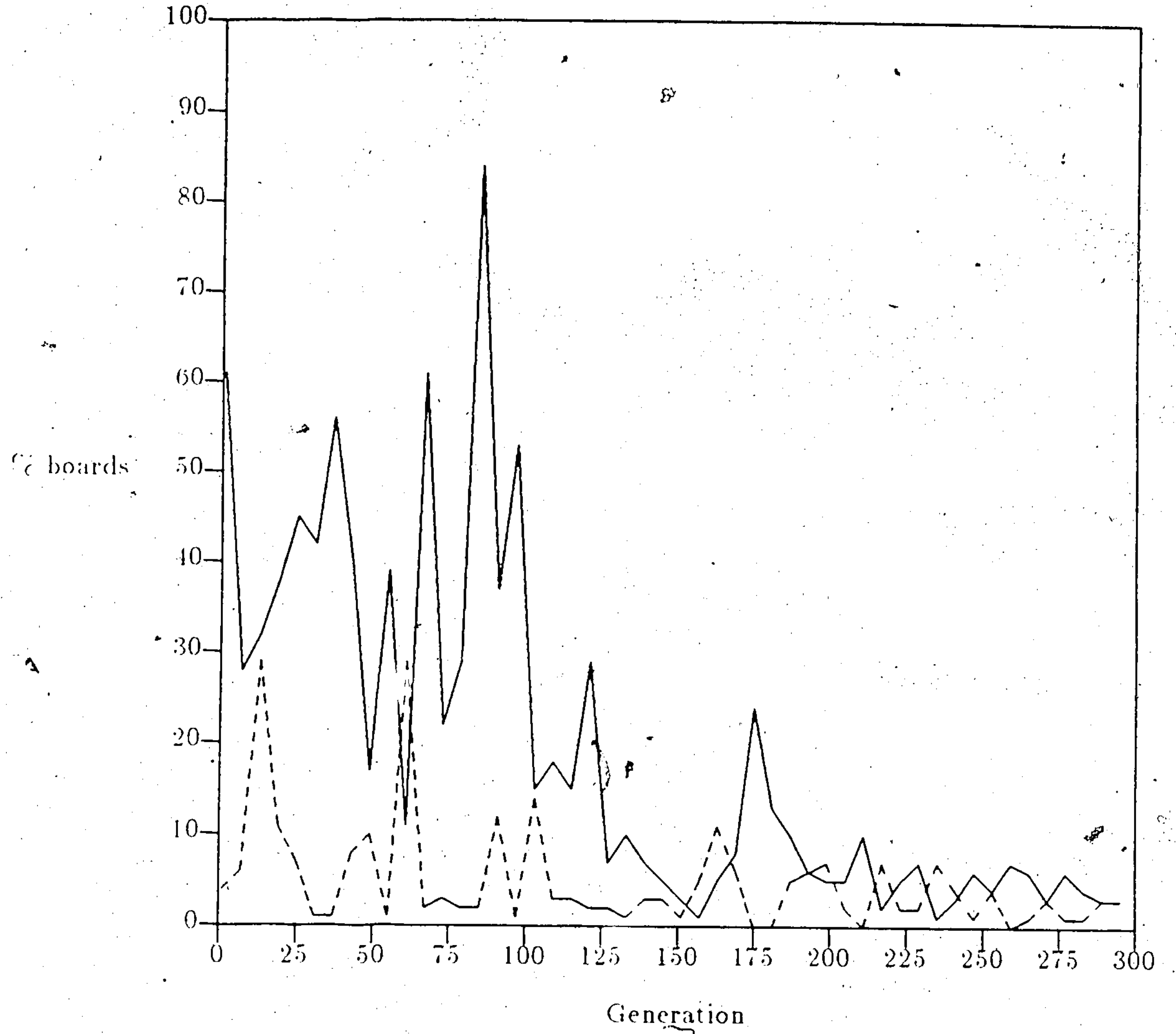


Figure 6.6 Illegal moves and no responses with no maxFireBid.

Chapter 7

Conclusion

The purpose of this thesis was to investigate learning classifier systems through the construction of, and experimentation with, such a system. To this end, the system LCS was developed and applied to the domain of tic-tac-toe. Results from initial experiments indicated a need for some enhancements to the system, which included the incorporation of taxation and the cover operator. Later experiments revealed the need for other enhancements such as the bid modification and the generalization operator in order to improve system performance. With these enhancements in place, the system proved to be quite successful in acquiring good performance in the tic-tac-toe domain.

Experimentation with LCS did reveal some limitations in these type of systems. First, the length of a message in a real world application would most likely be quite longer then that used for tic-tac-toe. Longer messages means longer condition and action strings in classifiers. Thus, the space of possible classifiers for a real world application would be considerably larger then that for tic-tac-toe resulting in significantly longer run times before the system "evolves" a classifier pool that is acceptable in terms of performance. Second, having a fixed size classifier pool limits the adaptive range of the system. Suppose the system performs well in an environment using a certain number of classifiers. If that environment should change such that adapting to it involves the addition or removal of a significant number of classifiers, then the system would fail to adapt properly due to its fixed size classifier pool. Allowing the size of the classifier pool to grow or shrink dynamically would obviously solve this problem. This is one possible future enhancement that could be made to this system.

Another drawback to using this system is the setting of all the parameters. For the most part, this is a relatively easy process — there are, however, a few exceptions. The tax parameters (`taxRate` and `maxFireTax`) required a fair bit of adjustment, although the adjustments were always in the direction so as to lessen the effect of taxation. In addition the payoff parameters, in particular the lose game payoff, and the rule base size parameter also required adjustments. Determining the current settings of these parameters required several runs and post-run analyses — a time consuming process. Also, it is still not certain that the current settings give the best performance possible. Further fine tuning could produce better results. As was mentioned before, the adjusting of these parameters introduces domain-specific knowledge into the system. This gives rise to the possible future enhancement of self-adjusting parameters in order to combat this.

Parameters which regulated themselves would be a step towards producing a more ideal learning system. The system might start out with some default parameter settings and, as it executes, it would adjust these settings according to run diagnostics and performance information of previous generations. Removing this task from the user and placing it into the system results in a more automated learning mechanism.

Another possible enhancement is the automated determination of the best possible assignment of string representations to application components. Currently it is the user's job to determine the best string representations to use — this also means that the user is introducing domain-specific knowledge into the system. For tic-tac-toe, that involved comparing the pros and cons of various representations before finally arriving at the choice of "10" for X, "11" for O, and "00" for a blank square. As was previously discussed, choosing other representations could lead to the inability to represent certain concepts and consequently can affect the learning potential of the system. While this problem is inherent in the classifier system formalism, there are nevertheless better and worse choices for the representation. Obviously, the best choice maximizes the learning potential of the system. Finding this choice, however, could be a nontrivial task. Thus, the automation of this choosing process would ease the task of application design.

The area of learning classifier systems is relatively new and is still in its infancy. Other than LCS, only a handful of general purpose learning classifier systems currently exist. These include systems developed by Riolo [Rio86b], Goldberg [Gol85], and Slate [Sla86]. As is indicated by the problems mentioned above, much work remains to be done before learning classifier systems can be applied to the difficult problems encountered in the real world. Hopefully, future research will make this possible.

References

[Cra85] Nichael Lynn Cramer, A Representation for the Adaptive Generation of Simple Sequential Programs, *International Conference On Genetic Algorithms And Their Applications*, 1985, 183-187.

[Dav85] Lawrence Davis, Job Shop Scheduling with Genetic Algorithms, *International Conference On Genetic Algorithms And Their Applications*, 1985, 136-140.

[Eng85] Arnold C. Englander, Machine Learning of Visual Recognition using Genetic Algorithms, *International Conference On Genetic Algorithms And Their Applications*, 1985, 197-201.

[Fen88] Keith Fenske, *An Interactive Classifier Programming Language*, M. Sc. Thesis, University of Alberta, 1988.

[Fou85] Michael P. Fourman, Compaction of Symbolic Layout using Genetic Algorithms, *International Conference On Genetic Algorithms And Their Applications*, 1985, 141-153.

[Gol85] David E. Goldberg, Genetic Algorithms and Rule Learning in Dynamic System Control, *International Conference On Genetic Algorithms And Their Applications*, 1985, 8-15.

[HiL87] M. R. Hilliard and G. E. Liepins, A Classifier-Based System for Discovering Scheduling Heuristics, *Second International Conference on Genetic Algorithms And Their Applications*, 1987, 231-235.

[Hol86] John Holland, Escaping Brittleness: The Possibilities of General Purpose Learning Algorithms applied to Parallel Rule-Based Systems, in *Machine Learning: Volume II*, 1986, 593-623.

[HHN86] John H. Holland, Keith J. Holyoak, Richard E. Nisbett and Paul R. Thagard, in *Induction: Process of Inference, Learning, and Discovery*, 1986, 68-150.

[Rio86a] Rick L. Riolo, *LETSEQ: An Implementation of the CFS-C Classifier System in a Task-Domain that Involves Learning to Predict Letter Sequences*, Technical Report, Logic of Computers Group, Division of Computer Science and Engineering, University of Michigan, Ann Arbor, Michigan, January 1986.

[Rio86b] Rick L. Riolo, *CFS-C: A Package of Domain Independent Subroutines for Implementing Classifier Systems in Arbitrary, User-Defined Environments*, Technical Report, Logic of Computers Group, Division of Computer Science and Engineering, University of Michigan, Ann Arbor, Michigan, January 1986.

[Sla86] David Slate, personal communication, *Los Alamos, NM*, 1986.

Appendix A1

LCS Command Summary

The following is a summary of the commands accepted by LCS:

`actBndry` *start length lglValCnt {lglVal}*

Defines a boundary (field) in the action string and establishes the set of legal values for that boundary. *start* indicates the bit position where the boundary starts (the first bit position in an action string is zero); *length* indicates the length of the boundary in bits; *lglValCnt* indicates the number of legal values for that boundary; *lglVal* indicates a legal value (there should be *lglValCnt* legal values). A legal value is a string from the alphabet {0,1,#} that is *length* symbols long.

`addToPOGrp` *grp*

Adds the currently active classifiers to the group number *grp*. *grp* should be a positive integer from 1 up.

`bidCoeff` *coeff*

Sets the bid coefficient, used for bid calculations, to *coeff*. This parameter should be a float between zero and one (e.g. 0.3).

`clear`

Performs system initialization.

`clfCntBelowThres`

Displays a number indicating the number of classifiers whose strength is currently below the threshold.

`clfPostLvl` *postLvl*

Sets the classifier post level to *postLvl*. The *postLvl* is a positive integer that indicates the maximum number of classifiers that are allowed to fire in one execution cycle. A value of zero indicates that an infinite number of classifiers are allowed to fire.

`close`

Exits the system.

`condBndry` *start length lglValCnt {lglVal}*

Defines a boundary (field) in the condition string and establishes the set of legal values for that boundary. *start* indicates the bit position where the boundary starts (the first bit position in a condition string is zero); *length* indicates the length of the boundary in bits; *lglValCnt* indicates the number of legal values for that boundary; *lglVal* indicates a legal value (there should be *lglValCnt* legal values). A legal value is a string from the alphabet {0,1,#} that is *length* symbols long.

`coverCond` *cond.*

Defines the set of messages that the cover operator will cover. All those messages that satisfy *cond* are messages which can be covered. *cond* is a string from the alphabet {0,1,#} that is the same length as condition strings.

`coverRate` *rate*

Sets the cover rate to *rate*. This rate controls the amount of new classifiers created through the cover operation. This parameter should be an integer between 0 and 100 where a value of 100 indicates that all the classifiers left to be replaced will be replaced using the cover operation.

`crossoverProb` *prob*

Sets the probability that the highest strength classifiers will be used in the crossover operation. *prob* should be an integer value between 0 and 100 (e.g. 95) where a value of 100 indicates that only the highest strength classifiers will be used for crossover.

`crossoverRate` *rate*

Sets the crossover rate to *rate*. This rate controls the amount of new classifiers created through the crossover operation. This parameter should be an integer between 0 and 100 where a value of 100 indicates that all the classifiers left to be replaced will be replaced using the crossover operation.

`fireProb` *prob*

Sets the fire probability to *prob*. This parameter should be an integer between 0 and 100 (e.g. 95). The fire probability indicates roughly what percentage of the matches are examined in determining the highest bidding matches to fire.

`genCondMtch` *mtch*

Defines how much the condition strings of a group of classifiers must match in order for that group to be generalized. A group of classifiers whose condition strings have *mtch* corresponding bits or more that match is a potential group for generalization.

`generalizeRate` *rate*

Sets the generalization rate to *rate*. This rate controls the amount of new classifiers created through the generalization operation. This parameter should be an integer between 0 and 100 where a value of 100 indicates that all the classifiers to be replaced will be replaced using the generalization operation.

`genLife` *length*

Defines how many times (*length*) a newly created generalization will outbid the classifiers that it was formed from assuming that the generalization gets no feedback each time it fires. This parameter affects the initial strength given to generalizations. It should be set to a positive integer (e.g. 4).

`initStrength` *strength*

Sets the value of the initial strength to *strength*. Classifiers that are created from cover, crossover, and mutation operations will be given the initial strength *strength*. This parameter should be a positive integer (e.g. 100000).

`maxFireBid` *fireCnt*

Sets the maximum fire count used in bid calculations to *fireCnt*. This parameter can be any positive integer (e.g. 2).

`maxFireTax` *fireCnt*

Sets the maximum fire count used in the taxation of classifiers to *fireCnt*. This parameter can be any positive integer (e.g. 19).

`maxNumConds` *numConds*

Sets the maximum number of condition strings allowed in the condition part of a classifier to *numConds*. This parameter can be any positive integer.

`message` *msg*.

Adds a message to the message pool. *msg* is a string drawn from the alphabet {0,1} which is *msgSize* symbols long.

`messlist`

Displays the message pool.

`msgSize` *size*

Sets the size of messages to be *size* bits long. This parameter can be any valid positive integer.

`mutationProb` *prob*

Sets the probability that the highest strength classifiers will be used in the mutation operation. *prob* should be an integer value between 0 and 100 (e.g. 95) where a value of 100 indicates that only the highest strength classifiers will be used for mutation.

`nextcycle`

Performs one basic execution cycle.

`nextgeneration`

Produces the next generation of classifiers. This involves replacing only those classifiers whose strength is below the threshold with classifiers produced from the various discovery operations.

`payoff` *grp amt*

Issues payoff of the amount *amt* to the group number *grp*. *grp* should be a positive integer from 0 up (where 0 indicates the group of active classifiers) and *amt* should be an integer.

`rmvPOGrp` *grp*

Destroys the group number *grp*. This involves disassociating the classifiers in the group from that group and then destroying it. *grp* should be the group number of an existing group other than group number 0.

`rplcThres` *thresh*

Sets the replacement threshold to *thresh*. Classifiers whose strength are below *thresh* at the time a new generation is being produced, will be replaced. This parameter should be any positive integer (e.g. 500).

`rule` *condition/action;strength;fireCnt*

Adds a classifier (rule) to the classifier pool. *condition* is the condition part which is a list of condition strings separated by commas, where each condition string is a string from the alphabet {0,1,#} which is equal in length to messages and may optionally be prefixed with a minus sign ("-"). *action* is the action part which is an action string drawn from the alphabet {0,1,#} which is equal in length to messages. *strength* is an integer value indicating the initial strength of the classifier. *fireCnt* is an integer value indicating the initial fire count for the classifier.

`rulelist`

Displays the classifier pool.

`seed` *seedNum*

Sets the seed number used for generating random numbers to *seedNum*. This parameter can be any valid positive integer.

`taxRate` *rate*

Sets the tax rate to *rate*. This parameter should be a float between zero and one. Usually, it is near zero (e.g. 0.003).

`trackmsg` *msgId*

Displays the classifier that placed the message, with id number *msgId*, on the message list. *msgId* is the id number of a message that is currently on the message list.

Appendix A2

Tic-tac-toe Pseudo Code

Following is the pseudo code for the tic-tac-toe program (note that BEC stands for basic execution cycle):

```
set system parameters;
construct initial rule base;
for each generation cycle do
   while ((the minimum BECs have not been performed) or
      ((there are not enough replacable rules) and
      (the maximum BECs have not been reached))) do
      get the opponent's move;
      if the opponent wins then
         issue "lose game" payoff to payoff group #1;
         remove payoff group #1;
         start a new game;
      else if the board is full then
         issue "draw game" payoff to payoff group #1;
         remove payoff group #1;
         start a new game;
      else
         standardize the board;
         encode the board into an input message;
         call nextcycle();
         if there is an output message then
            add active rules to payoff group #1;
            update board according to output message;
            destandardize the board;
            if the system move is legal then
               issue "legal move" payoff to payoff group #0;
               if the system wins then
                  issue "win game" payoff to payoff group #1;
                  remove payoff group #1;
                  start a new game;
               else if the board is full then
                  issue "draw game" payoff to payoff group #1;
                  remove payoff group #1;
                  start a new game;
               end;
            else
               issue "illegal move" payoff to payoff group #0;
               remove payoff group #1;
               start a new game;
            end;
         else
            remove payoff group #1;
            start a new game;
         end;
      end;
   end;
   if ((this is the first generation) or
      (the "no response" count is greater than 37)) then
      set the generalization rate to 0;
      set the cover rate to 100;
      set the crossover rate to 40;
   else
      set the generalization rate to 30;
      set the cover rate to 100;
      set the crossover rate to 40;
   end;
   call nextgeneration();
end;
```