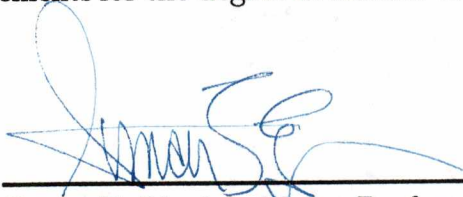


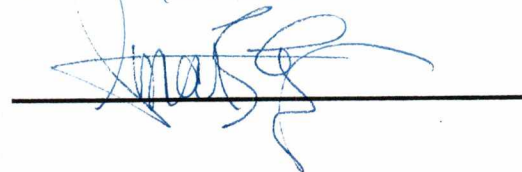
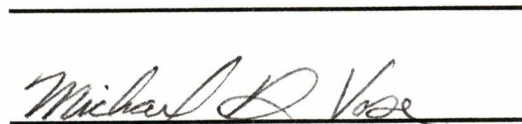
To the Graduate Council:

I am submitting herewith a thesis written by Lori A. Wang entitled "Classifier System Learning of the Boolean Multiplexer Function." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.



Gunar E. Liepins, Major Professor

We have read this thesis
and recommend its acceptance:



Accepted for the Council:



Vice Provost
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Lori A. Wang

Date November 14, 1990

**Classifier System Learning of the Boolean Multiplexer
Function**

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Lori A. Wang

December 1990

ACKNOWLEDGMENTS

I would like to thank my thesis advisor and computer science professor, Dr. Gunar Liepins, for sharing his time and ideas with me on this thesis project so that I could achieve my goals. Taking Dr. Liepins' classes, and being exposed to the areas he's interested in, has widened my interests.

I would also like to thank my other committee members, Dr. Michael Vose and Dr. David Mutchler, for their lively discussion and participation at my thesis defense. And, I especially thank Dr. Mutchler for his valuable suggestions for improvements in the text; all were incorporated into the final version.

I gratefully acknowledge the use of the excellent computing facilities of the computer science department's undergraduate and graduate laboratories and those of the university's computing center(UTCC).

ABSTRACT

Classifier systems are dynamically adapting production rule systems that modify their rules through experience. This thesis is an empirical study of classifier system learning applied to a particular Boolean reinforcement learning problem, the multiplexer function. The multiplexer function is a special case of a concept expressible in a disjunctive normal form, the $k + 2^k$ multiplexer is expressible as a $k + 1$ disjunctive normal formula. A variety of modifications to the classifier system are investigated to induce rapid learning. Some of the methods tried are dynamic population sizes, controlled initial population generality, bit flipping, relative complement, dynamic operator control, and evaluation on demand. Experimental results suggest the following: the initial population generality influences learning performance; relative complement is a more effective operator than bit flipping; and dynamic operator control, and evaluation on demand are necessary options for learning in the larger k size problems.

PREFACE

Several sets of canonical test systems were run for the $k+1$ -DNF multiplexer function, where $k = 2, 3, 4$, the address length for the multiplexer. The test systems were used as experiments to determine the usefulness of technical modifications for improving the learning speed of a classifier system. Experimental designs for the test systems as well as a tabular summary of the overall results are given in the Appendices.

First, exploratory investigations were directed at finding a reasonable performing reference system. The reference system found used the following: a symmetric reinforcement method, noisy bidding, population sizes of 400, 800, and 1600 for $k=2, 3, 4$ respectively, and initial population generalities of 33%, 50%, and 64% hash. This reference system was then used to test for the effects of several technical modification options. Reinforcement methods were tested as well as different genetic algorithm implementations. Dynamic operator control was used to dynamically allocate greater probabilities to better performing operators. Evaluation on demand was used to investigate the importance of improved accuracy in feedback. The variant systems were used as a special group of test systems using more complicated options. All the variant systems use the reduced surrogates genetic algorithm, and a generality threshold. The generality threshold limits the population to classifiers with a certain minimum specificity.

Later, a pure genetic algorithm is implemented to learn the multiplexer func-

tion using three different static evaluation functions. These three functions are analyzed for their performance, as well as, to illustrate the difficulty in choosing a good evaluation function.

TABLE OF CONTENTS

CHAPTER	PAGE
1. INTRODUCTION	1
1.1 Description of classifier systems	1
1.2 Description of genetic algorithms	7
1.3 Description of inductive inference	8
1.4 Classifier systems and concept learning	11
2. MOTIVATION	15
2.1 Can technical modifications to the classifier system algorithm improve performance?	15
2.2 Decision tree methods	16
2.3 Neural network methods	20
2.4 Classifier system methods	23
3. OBJECTIVE	25
3.1 An empirical study of classifier system performance	25
3.2 The multiplexer function	26
3.3 Rule spaces	28
3.4 Instance spaces	29
3.5 Population space	30

3.6	Examples of genetic algorithm operators	30
3.7	Vose's stable, unstable schemata theorems	31
3.8	Internal models	33
4.	A SIMPLIFIED CLASSIFIER SYSTEM	35
4.1	Introduction	35
4.2	Methods	36
4.3	Operational description	38
4.3.1	Genetic algorithm	39
4.3.2	Create	40
4.4	Exploratory systems	40
4.4.1	Dynamic population sizes	41
4.4.2	Controlled initial hash percentage	44
4.4.3	Reinforcement regimes	45
4.5	Technical modifications	49
4.6	Genetic algorithms	49
4.6.1	Mate with like	50
4.6.2	Reduced surrogates	51
4.6.3	Reduced surrogates and mate with like	52
4.7	Operators	52
4.7.1	Bit flipping	53
4.7.2	Relative complement	56
4.8	Dynamic parameter control	61

4.8.1	Dynamic operator control	62
4.9	Training set generation	65
4.9.1	Evaluation on demand	66
4.10	Dynamic operator control combined with evaluation on demand	70
4.11	Variant systems	72
4.11.1	Variant I	73
4.11.2	Variant II	73
4.11.3	Variant III	74
4.11.4	Variant IV	74
4.12	Discussion	75
5.	A PURE GENETIC ALGORITHM	77
5.1	Introduction	77
5.2	Methods	77
5.3	Operational description	78
5.3.1	Genetic algorithm	79
5.3.2	Create	80
5.4	Static evaluation functions	80
5.5	Genetic algorithms	84
5.5.1	Mate with like	84
5.6	Discussion	85
6.	SUMMARY AND CONCLUSIONS	86

BIBLIOGRAPHY	89
APPENDIXES	94
APPENDIX A. - SIX MULTIPLEXER DATA SETS	95
APPENDIX B. - ELEVEN MULTIPLEXER DATA SETS	120
APPENDIX C. - TWENTY MULTIPLEXER DATA SETS	133
VITA	147

LIST OF TABLES

TABLE		PAGE
Appendix A		
1	Experimental design for the exploratory systems	96
2	Experimental design for the technical modifications systems . . .	96
3	Experimental design for the variant systems	96
4	Summary of performance results for the six, eleven, and twenty multiplexers	97
5	Parameter set for the exploratory systems	98
6	Parameter set for the technical modifications systems	99
7	Parameter set for the variant systems	100
8	Example of a six multiplexer	101
9	The six multiplexer's rule space	101
10	The six multiplexer's instance space	101
11	The solution set classifiers for the six multiplexer	102
12	Experimental design for the pure genetic algorithm	103
13	Parameter set for the pure genetic algorithm	103

LIST OF FIGURES

FIGURE	PAGE
Appendix A	
1	Dynamic population size and hash percentage for the six multiplexer 104
2	Controlled initial hash percent for the six multiplexer 105
3	Wilson, Sen, and symmetric reinforcement methods for the six multiplexer 106
4	Correct and random actions on CREATE for the six multiplexer 107
5	Genetic algorithms for the six multiplexer 108
6	Bit flipping operator for the six multiplexer 109
7	Relative complement operator for the six multiplexer 110
8	Dynamic operator control for the six multiplexer 111
9	Dynamic operator control operator performance curves for the six multiplexer 112
10	Evaluation on demand for the six multiplexer 113
11	Dynamic operator control and evaluation on demand for the six multiplexer 114
12	Variant I, II, III, and IV for the six multiplexer 115
13	Pure-GA static evaluation functions for the six multiplexer . . . 116

14	Hierarchical rule-ordering for the three multiplexer	117
15	Pure-GA using speciation for the six multiplexer	118
16	Pure-GA macrostates for the six multiplexer	119

Appendix B

1	Dynamic population size and hash percentage for the eleven multiplexer	121
2	Controlled initial hash percent for the eleven multiplexer	122
3	Wilson, Sen, and symmetric reinforcement methods for the eleven multiplexer	123
4	Genetic algorithms for the eleven multiplexer	124
5	Bit flipping operator for the eleven multiplexer	125
6	Relative complement operator for the eleven multiplexer	126
7	Dynamic operator control for the eleven multiplexer	127
8	Dynamic operator control operator performance curves for the eleven multiplexer	128
9	Evaluation on demand for the eleven multiplexer	129
10	Dynamic operator control and evaluation on demand for the eleven multiplexer	130
11	Variant I, II, III, and IV for the eleven multiplexer	131
12	Pure-GA static evaluation functions for the eleven multiplexer	132
13	Pure-GA using speciation for the eleven multiplexer	132

Appendix C

1	Dynamic population size and hash percentage for the twenty multiplexer	134
2	Controlled initial hash percent for the twenty multiplexer	135
3	Wilson, Sen, and symmetric reinforcement methods for the twenty multiplexer	136
4	Genetic algorithms for the twenty multiplexer	137
5	Bit flipping operator for the twenty multiplexer	138
6	Relative complement operator for the twenty multiplexer	139
7	Dynamic operator control for the twenty multiplexer	140
8	Dynamic operator control operator performance curves for the twenty multiplexer	141
9	Evaluation on demand for the twenty multiplexer	142
10	Dynamic operator control and evaluation on demand for the twenty multiplexer	143
11	Variant I, II, III, and IV for the twenty multiplexer	144
12	Long runs(maxtrials=70,000) for the twenty multiplexer	145
13	Pure-GA static evaluation functions for the twenty multiplexer	146
14	Pure-GA using speciation for the twenty multiplexer	146

CHAPTER 1

INTRODUCTION

1.1 Description of classifier systems

Classifier systems are rule-based production systems that adapt and evolve to model critical aspects of their problem environment.

Production systems are computer programs that use a data base of rules to represent their knowledge. The data base of rules is simply a list of rules: rule 1, rule 2, rule 3, etc. Production rules are commonly used in expert systems [34, Rich,1983]. Expert systems are computer systems, or programs, that can solve very specific, narrowly defined problems at, or near, the level of a human expert [19, Harmon and King, 1985]. For example, **MYCIN** is a rule-based expert system that is used for diagnosing bacteremia and meningitis infection. **XCON** is a rule-based expert system that is used to configure computer hardware systems. **DRILLING ADVISOR** is another rule-based expert system that is used to advise oil drilling crews. **PUFF** is a rule-based production system used to diagnose obstructive airway diseases in humans. There are many more expert systems that use production rules to represent knowledge [4, Barr and Feigenbaum, 1981].

The rules in a rule-based production system are generally *IF-THEN* type rules which contain two parts: a premise, and a conclusion. The rule's premise is tested against an activating agent for the premise's truth or falsity. If the premise tests true then the rule's conclusion is made, or *fired*, otherwise the next rule's premise is tested, and so on. In traditional expert systems, each rule would be tested and fired in a serial, sequential fashion.

Rule-based production systems can be used to represent knowledge for many different types of problems. But production systems suffer from their own set of problems. For instance, serially testing and firing rules is a slow process. Complex problems often require fast solutions. Production systems need to be able to test and fire rules in parallel.

Another problem concerns updating the rule base. Updating requires an on-going, expensive effort by the expert and the knowledge programmer to keep the knowledge base current with available information in the field.

Another problem concerns how to get the knowledge into the data base in the first place. This is what Feigenbaum has called the "*knowledge acquisition bottleneck*" [4, Barr and Feigenbaum, 1981]. The human expert must perform the difficult task of taking their knowledge and transforming it into a set of production rules. This is difficult because the expert must be able to express what they know in rules, and they must also be able to assign a figure of merit to each rule as it is added. This is not only difficult and time-consuming, but sometimes impossible because the expert often is not aware of the heuristics

she/he is using to solve problems. To input what they know into an expert system, the human expert must transform their expertise into high level symbolic knowledge. But their expertise may be at a lower more intuitive level in the form of an *internal model*[10, Caudill and Butler].

Another problem concerns the fact that an expert system built for a particular problem domain, say, diagnosing infectious diseases, can only be used to diagnose infectious diseases. Traditional expert systems are not easily, nor successfully, transferable from one problem domain to another. (A problem domain is a class, or a group, of problems. For instance, the class of diseases that are infectious is one problem domain. Another, smaller problem domain, would be the class of diseases that are infectious menengitis.) Traditional expert systems are not robust. They do not transfer well from one domain to another.

In summary, there are four major problem areas with traditional expert systems:

- Serial versus parallel rule testing - serial rule testing and firing is a slow, time-consuming method for rule activation [15, Goldberg, 1989].
- Knowledge acquisition bottleneck - the human expert and knowledge engineer entering data into the knowledge data base is a slow and tedious process.
- Maintaining a current knowledge base - updates to the rule base can be costly, or sometimes impossible.
- Brittle software solutions - traditional expert systems are not easily transferable between problem domains. They are not robust [21, Holland,1981].

Classifier systems are dynamic, adaptive, evolutionary learning programs which *learn* from their experiences. Classifier systems are rule-based production sys-

tems which maintain a list of production rules called a population. This is usually a small sample from the universe of all possible rules for that problem.

In classifier systems, rules are activated, or matched and fired, in parallel. Each rule is an *IF-THEN* rule that encodes a premise and a conclusion. Rule premises are matched against working memory consisting of *environmental and internal messages*. Premises can be general insofar as they have wild cards in them and can match multiple memory entries. This is what allows for parallel rule activation. One rule can represent many instances of the problem domain. Matching rules compete for the right to have their conclusion made. Classifier systems using parallel rule activation do not suffer the same time constraint problems of systems using serial rule activation.

In classifier systems, rules are rewarded or punished according to the utility of their premises and conclusions. In traditional expert systems a rule's merit is set by the human expert when the rule is originally placed into the data base. Classifier systems allow a rule's utility to evolve. Classifier systems do not start with a pre-arranged set of rules as in traditional expert systems but rather they start with a randomly generated small sample of rules. Through adaptation with the environment a set of plausible rules emerge within the population.

We humans model our environments by developing and testing an internal model of what we perceive to be true and false about the world. We do not have perfect information about our environment, nor are our attitudes and perceptions necessarily correct or even approximate. Nonetheless, we use our internal

model to anticipate future outcomes.

Holland describes something that he calls a *default hierarchy*, or *q-morphism*[23, Holland, Holyoak, Nisbett, and Thagard, 1986]. A q-morphism is a quasi-homorphism which is a many-to-one function. A homomorphism is a function which maps many instances, in our case this would be environmental messages, onto one result, this would be a classifier, or rule. Default hierarchies consist of a hierarchical rule set in which a general rule takes care of the most common situations while specific rules handle the exceptional cases. In principle, default rule hierarchies can develop in classifier systems that are provided with delayed reinforcement, as well as stimulus-response systems [22, Holland, 1985]. In our system we use immediate reinforcement in a stimulus-response method. Theory to the contrary, we have not seen default hierarchies evolve when using immediate payoff and penalty as in stimulus response systems. Instead the internal model of the environment still evolves but it may not be hierarchical. This internal model, which becomes the default rule set, should be able to handle most, if not all, situations. It should, at the very least, be able to handle those cases that were used to build the model. One particular internal model that perfectly handles the multiplexer function has been called the *solution set* by [39, Wilson, 1987]. The solution set classifiers make up the set of conjunctive rules that when disjoined together form the minimal representation of the logic circuit function for the multiplexer.

Because default hierarchies and other perfect, or nearly perfect, rule sets

can be automatically evolved using classifier systems, they do not suffer from the *knowledge acquisition bottleneck* problem as do traditional expert systems. Classifier systems do not require a human expert, nor a knowledge engineer, to create the rule base. Rather, classifier systems generate their own rule base.

Classifier systems are adaptive learning systems. To maintain a current up-to-date population of rules would only require updating the information in the environmental message. Classifier systems generate their own rule base. If new information pertaining to the field became available, then the rule base would be recreated using that new information in the classifier system environment. Classifier systems do not suffer from problems in updating the rule base, because they generate their own knowledge.

Traditional expert systems also suffer from something known as *brittleness*, which means lack of robustness [23, Holland, Holyoak, Nisbett, and Thagard, 1986]. They are not easily, nor successfully, transferred from one problem domain to another. In classifier systems the problem domain is expressed as an environmental message. A rule base is generated using the classifier system's *experiences* with the environmental message along with its rewards and punishments. Classifier systems are general learning systems. To change the problem domain would only require changing the environmental message. Because classifier systems interact with an environment they are more easily transferable among different problem domains. Adaptive, evolutionary rule-based production systems are more robust than traditional expert systems. Ideally, the clas-

sifier system would be a general problem solver; it could provide solutions to a variety of problems.

1.2 Description of genetic algorithms

Genetic algorithms are the primary rule search mechanism for classifier systems. Because of this, classifier systems are often called genetics based machine learning algorithms [15, Goldberg, 1989]. Genetic algorithms are an optimization technique, their goal is to find a minimum or a maximum. They are also called *evolutionary* optimization techniques because they are based on the evolutionary metaphor. Traditional search methods are calculus based algorithms. These are man-made methods as opposed to algorithms borrowed from nature. Genetic algorithms use natural selection, or survival-of-the-fittest, to find a minimum or a maximum. Genetic algorithms, as developed by Holland, use simple bit string representations, and simple operators on the bit strings to search a space of bit strings [20, Holland, 1975]. The bit strings represent points in the problem domain. The primary genetic operators are selection, crossover, and mutation on the bit string patterns. Each string is assigned a figure of merit, or utility, based on an evaluation function. The evaluation function reflects information about the problem domain. In a conventional, or pure, genetic algorithm, a population of bit strings, or instances, would be maintained. At each generation, new strings are created using selection, crossover, and mutation on pairs

of strings. The new strings are called the offspring and the paired strings are called the parents of the offspring. This is repeated at each generation until the population converges on some strong individual string, or until the population can make no further changes. Each string is thought of as being a chromosome, with each bit being a loci in a chromosome. The chromosomes and the reproductive operators supposedly mimic Darwinian natural selection. But genetic algorithms, as we use them, operate on finite populations to solve a finite task [17, Goldberg and Richardson, 1987]. Some would argue that the focus, in open systems, should lie less on a fitness gradient and more on self-organizing mechanisms and an increasing structural complexity, though the principle of natural selection is a powerful one [3, Austin, 1990].

1.3 Description of inductive inference

Inductive inference is “the process of hypothesizing a general rule from examples.” [2, Angluin and Smith, 1983] Classifier systems are an inductive inference process. They use examples of seen cases of the problem domain, to generate a rule data base that will generalize to unseen cases. Inductive inference is “a fundamental component of intelligent behavior”. One example of an inference process is a black box that outputs a sequence of numbers representing some unknown function.

For example, for $x = (1, \dots, n)$ the sequence:

$$1, 3, 5, 7, 9, \dots$$

is an example of the function,

$$f(x) = (2 \times x) - 1.$$

A user would interact with the black box by making guesses as to the next number in the sequence. The black box contains a component called an oracle. An oracle answers yes or no as to whether the guess was right or wrong [24, Hopcroft and Ullman, 1979]. Using trial and error, the user would interact with the black box and the oracle to form a rule about the unknown function. The user will have discovered a good, or plausible, rule when she/he is able to correctly guess the next number in the sequence with a certain degree of accuracy. For instance, if 95% of the time the user guesses the correct next number in the sequence, then she/he has formed a plausible rule about the function.

Angluin and Smith cite two basic inductive methods from [14, Gold, 1967] for discovering a good rule:

1. identification in the limit and,
2. identification by enumeration [2, Angluin and Smith, 1983].

Identification in the limit is a method that views inductive inference as an infinite process. The process converges when the rule no longer requires modification. The limit is reached when all succeeding rules are the same. Identification makes no assumptions on the type of rule that it forms, nor on the

form of the training examples. The goal in the multiplexer problem is to find a population of rules of any string length, and any cardinality alphabet, trained with any length string exemplars, that encodes the multiplexer concept. We break up the problem into the six, eleven, and twenty cases, but the overall goal is to find some population of rules that can give correct answers for any size problem. Classifier systems are using a form of identification in the limit. (Judd also refers to Gold's identification in the limit when analyzing the learning times of neural networks [28, Judd, 1990].)

Identification by enumeration is a method whereby rules are systematically generated and tried until a plausible rule is found. In the case of polynomial functions, each degree polynomial would be systematically tried until one was found that worked. Identification by enumeration would require that strings be a specific length, from a certain cardinality alphabet, trained with a specific set of examples. Genetic algorithms enumerate strings in the rule space using crossover and mutation.

Classifier systems are an inductive inference process. They use both identification in the limit and identification by enumeration for generating a rule base. They strive to generate populations of rules that yield maximum reinforcement to the system. The genetic algorithm, the primary search mechanism for a classifier system, uses a form of enumeration with hill-climbing to generate new rules for the classifier system to test. Genetic algorithms enumerate the possible rules in the population and use hill-climbing techniques to determine an optima.

Sometimes they converge to the global point; sometimes they prematurely converge on a local minima or maxima. But, in any case, they enumerate and test populations of rules.

1.4 Classifier systems and concept learning

Classifier systems are concept learning programs. They learn concepts about the problem domain. They offer an alternative to having the problem solution hard-wired, or pre-programmed, into them as in a procedural algorithm.

Classifier systems can be thought of as stimulus-response systems. They view an input, or stimulus, the environmental message, and they give an output, the competing, winning rule's conclusion. In stimulus-response problems the classifier population is rewarded or punished based on the accuracy of the winning rule's conclusion.

Classifier systems do not always depend on an oracle that decides whether a particular rule was right or wrong. For chaining classifiers, the bucket brigade is supposed to yield a reasonable estimate of each rule's worth.

In principle, classifier systems can learn either in a stimulus-response mode or a delayed reinforcement mode. Not all classifier systems are necessarily stimulus-response based; some use reinforcement learning. The application studied here is formulated as a stimulus-response problem.

Stimulus-response as a learning process has been well studied, and forms the

basis in psychology for classical conditioning. However, there is controversy concerning the validity of stimulus-response systems as true measures of learning [25, Hunt, 1962]. Another question concerns what types of problems are learnable using stimulus-response methods. There is similar controversy surrounding the performance and capabilities of computational learning models, such as classifier systems, that use stimulus-response as their learning mechanism.

One of the early learning programs was Hunt's *concept learning system*, (CLS),

An intelligent device, including man, can be thought of as something capable of adjusting to its environment. To make such adjustments the device must continually be classifying slightly different states of the environment as equivalent or not equivalent. The particular classifications may vary from time to time: sometimes we group whales with bears; sometimes we group whales with sharks. Still, the intelligent device must classify. Now suppose that we wish to design such a device. Unless we, the designers, know all the states of the environment which the device may encounter, we have no way of prewiring it with the appropriate classification rules. Instead we must provide it with a capacity to develop classification rules from experience. [26, (pp. 3) Hunt, Marin, and Stone, 1966.]

Hunt developed the program called CLS for Concept Learning System in the 1960's on an IBM 7090 computer using Algol. CLS learned problems from symbolic logic. For instance:

- Conjunction - $A \bullet B$
- Inclusive disjunction - $A \vee B$
- Implication - $A \implies B$
- Exclusive disjunction - $(A \vee B) \bullet \neg(A \bullet B)$
- Biconditional - $A \iff B$

Hunt studied three problem sets each formed from the set above and increasing in complexity. There were 9 variants of CLS(CLS-1,...,CLS-9) in which he tried various technical modifications to the basic CLS algorithm to improve performance.

CLS used a decision tree approach in which a vector of attributes was used to describe an instance of the problem. One field, or variable, in the vector was used as the solution, or the class, to which the instance belonged. In this case, there were only positive and negative instances of the concept, so the solution was binary. CLS learned the problem, $(A \vee B) \vee C$ from problem set II, in 12,000 *program cycles* with an 86% accuracy rate[26, Hunt, Marin, and Stone, 1966].

Decision tree approaches, generally, use a small sample of test cases, and separate the test cases into sub-samples of test cases using a binary tree and a tree-splitting criterion. CLS decided splits using a relative frequency count on the value of an attribute in a node along with its class. The tree splits are repeated until the leaf nodes of the tree contain only test cases that are all of the same category.

Later, Quinlan developed ID3, a decision tree learning program, that borrowed on Hunt's early CLS algorithms [32, Quinlan, 1983]. Quinlan's ID3 has successfully learned a variety of concepts quickly and accurately. Decision tree methods, including ID3, have been used in commercial applications such as **Expert-Ease**, and **TIMM**, two expert system building tools [19, Harmon and King, 1985]. Most machine learning methods have typically only been used in

prototypes [9, Buchanan, 1989]. The aim of machine learning has been to build general purpose problem solving systems. Systems that can "learn" to solve any arbitrary problem within a reasonable amount of time. This ambition has not been fully realized using inductive models.

CHAPTER 2

MOTIVATION

2.1 Can technical modifications to the classifier system algorithm improve performance?

What are the capabilities of inductive learning models such as classifier systems? What class of problems are classifier systems good at learning? What class of problems are easily solved using an inductive inference process? In this thesis we study only one problem, the multiplexer function. The multiplexer function is a member of a larger class of problems, all Boolean functions. There have been other studies concerning the multiplexer problem and inductive learning models. A description of some of these studies follows.

Along with the question of which problem domains are solvable using classifier systems is the question of whether or not modifications to the classifier system algorithm can improve its performance. Learning speed, or performance, is measured as the number of examples, or environmental messages, that need to be shown to the classifier system before it can form a plausible set of rules. The fewer examples that need to be shown to the system the faster its learning speed. Performance is measured as both speed and accuracy. For purposes of

this study, accuracy is measured as the number of correct answers given by the system in the last fifty running trials.

2.2 Decision tree methods

There have been some studies on the multiplexer problem using decision trees. Decision trees are another inductive learning method. In this method the finished model of the environment consists of a tree structure instead of a rule base as in classifier systems. After the tree is built a problem would be run down the tree to be classified accordingly. A decision tree can be thought of as a message generator, that extracts information from a source of data[32, Quinlan, 1983].

Decision trees, generally, use a portion of the universe, or instance space, called a learning sample. The learning sample is usually split into a 2/3, 1/3 split. The larger portion, consisting of several test cases, is used to build the tree, and the smaller portion is used to test the model [8, Breiman, Friedman, Olshen, and Stone, 1987]. Some decision tree methods, such as ID5R [36, Utgoff, 1988], use incremental methods to build the tree structure instead of using a number of cases all at once.

Quinlan created a method called, ID3[31, Quinlan, 1979], in which he borrowed from Hunt's earlier CLS algorithms[32, Quinlan, 1983]. A group of exemplars, instead of one at a time, are used to generate a tree structure, which

is then tested using test cases saved out of the learning set. In ID3 the root node of a binary tree begins with a set of all of the cases in the learning sample. The next two nodes in the tree are obtained by splitting the set of cases in the root node using an attribute, or feature, of the exemplar vectors. Quinlan used information theoretic heuristics, Shannon's law, to decide which attribute to use to decide the tree splits. Shannon's law finds the attribute that when split on will give the greatest information content, or the greatest decrease in test case node diversity. The ID3 method builds several trees using a learning *window*. The window consists of the training set and any exceptions, or test cases that the tree failed to correctly classify. Several trees are iteratively built using the learning window until one is found that is the smallest tree that can accurately classify all test cases. The goal is to find the smallest tree that can correctly classify all of the samples in the universe of cases with a certain degree of accuracy.

Quinlan also studied the multiplexer problem using a decision tree method called C4[33, Quinlan, 1988]. In this study Quinlan gives insight into the multiplexer problem itself. First he identifies a straight-forward decision tree for the six multiplexer problem that a human problem solver would generate. In this decision tree the address bits are tested first, then the data bit addressed by the address bits to determine class membership. But Quinlan says that when C4 attempts to build a decision tree for the multiplexer problem, it tests a data bit first. This is because the address bits give a 50% probability of class membership.

Given an address, the probability for the function to be on or off is one-half. But given a data bit, the probabilities change. This can be seen by enumerating all 64 instances and counting class membership, holding one data bit constant. Thus C4 will initially use the bit that gives the greatest information gain, and that would be a data bit.

But Quinlan goes on to successfully build some accurate decision trees for the six multiplexer. He used training set sizes of 50, 100, 150, and 200 with accuracy rates on 1,000 unseen cases of 77% with a 10 node tree, 92% with a 22 node tree, 98% with a 22 node tree, and 100% with a 25 node tree, respectively. These trees could classify unseen cases well even though they were larger than the straight-forward hand built(optimal solution) example using 15 nodes.

Quinlan also examined the eleven and the twenty multiplexer problems using C4. For the eleven multiplexer he used training set sizes of 100, 200, 300, and 400 with accuracy rates of 63% with 23 nodes, 78% with 42 nodes, 87% with 63 nodes, and 93% with 69 nodes. For the twenty multiplexer he used training set sizes of 200, 400, 600, and 800 with accuracy rates of 69% with 49 nodes, 82% with 96 nodes, 87% with 121 nodes and 92% with 171 nodes. He found that as he scaled-up the bushiness of the trees increased over that of the optimal tree. It should also be noted that for the six multiplexer problem all the instance space could be seen in the training set for the 100, or greater, instance sets, but the eleven and the twenty saw only a small portion of their instance space with training set sizes of less than 400 for the eleven and 800 for the the twenty.

Quinlan is able to build accurate classifiers using C4's non-incremental methods with small training sets.

Utgoff used a variation of Quinlan's ID3 method. In this system, called ID5R, Utgoff builds the tree using incremental exemplar sampling (as opposed to grouping the exemplars), which allows the algorithm to selectively choose training cases and, thus, build more optimal trees. Utgoff has demonstrated that incremental ID5R can build the same trees as nonincremental ID3. ID5R builds the tree by beginning the process with the first instance as the root node. If the next example is the same class as the root node then it would be added into the set at the root node otherwise it would become a branch off the root node. On the next example, if there are more nodes than just the root node then the tree *pulls up* to the root node the node with highest frequency attribute that gives the greatest information gain, and so on for each example. Instead of a static structure, as in ID3, the decision trees of ID5R dynamically rearrange themselves.

Utgoff studied the multiplexer problem. ID5R built a 38 node decision tree to learn the six multiplexer function using on average only 126 training instances with a classification rate of 100% accuracy.

Another method for building decision trees is called CART which stands for Classification And Regression Trees [8, Breiman, Friedman, Olsen, and Stone, 1987]. This is another nonincremental method for building a decision tree using a learning sample. CART also uses a goodness of split criteria, as in ID3's

entropy measure, but other criteria such as the Gini-index, and the two-ing criteria, can also be used. CART uses a rule for assigning every terminal node to a class, and a stop-splitting criteria. The misclassification rate measures the goodness of the structure and this is used as a heuristic to guide the search for an optimal tree.

2.3 Neural network methods

There have been some studies on the multiplexer problem using the neural network method for problem solution. Neural networks are also an inductive learning method. Neural networks are structures consisting of nodes, links, and weights. The weights on the links between nodes are used to form an internal model of the environment. The network uses a learning algorithm to adjust, or *learn*, the optimal weights. The algorithms are generally optimization techniques, in which the goal of the algorithm is to find the minima of the error surface for the multi-dimensional weight space. Similar to a classifier system, a neural network uses a stimulus-response mechanism to find the error of the weight settings. An exemplar is presented to the input units of the network, which then computes the output using a function of the inputs. It does this for each layer of nodes until the final layer output is obtained. A supervisor in the network's environment determines the accuracy of the answer and uses the difference between the actual and desired response as feedback to the learning

algorithm. One run through the entire training set is called an epoch. Instead of a rule base as in classifier systems, the neural network rebuilds itself into a network that models its environment.

Jacobs studied four functions, including the six multiplexer function, using a neural network and several different learning algorithms [27, Jacobs, 1988]. In his version the six multiplexer was solved using six input nodes, six hidden nodes, and one output node. Jacobs used back propagation, which is a gradient descent procedure that uses the derivative of the output error with respect to the weights to move the weights in the correct direction. He also tried a momentum procedure which is a version of back propagation that saves the previous derivatives and uses this as a heuristic to determine if the weights are oscillating. He also tried something called delta-bar-delta, and a hybrid method. Delta-bar-delta is another version of back propagation that saves the learning parameter (proportion of change in weights) so that it can be dynamically changed. The hybrid method uses a combination of momentum and delta-bar-delta. For the multiplexer task, Jacobs' network learned the multiplexer problem on average 344 epochs using back propagation, 124 epochs using momentum, 137 epochs using delta-bar-delta, and 105 epochs using the hybrid method. An epoch for the six multiplexer should consist of 64 instances, so that the fastest learning time occurred when using the hybrid method at around 7,000 training examples.

Anderson also studied the multiplexer function using a neural network model [1, Anderson, 1986]. Anderson used two test sets. In the first set he

tested different representations of the input vector. In the second set he tested different weight learning algorithms.

In his first test set Anderson tried three representations of the input vector. In the first representation he used a seven input feature vector to learn the six multiplexer. Two inputs were used as the address bits, four were used as the data bits, and one input was always set to 0.5. In this model the network used a single-layer network. Anderson states that the multiplexer problem cannot be learned using a single-layer network because the multiplexer problem can be reduced to the exclusive-or problem, which is a non-separable task for a single linear function.

In the second representation Anderson used sixty-four components in the input vector and trained the network using only basis vectors. The network that resulted could learn the problem but couldn't generalize which, essentially, amounts to a look up table.

In the third representation Anderson used the original seven component input vector plus four hidden nodes. The hidden nodes represented *missing features* that weren't directly present in the input vector. The missing features were the four disjuncts in the disjunctive normal formula for the multiplexer.

Anderson's results for the first test set showed that the six multiplexer could be learned in 200 time steps using the hidden layer network.

In the second set of tests on the six multiplexer, Anderson analyzed eleven different learning algorithms. Previously he had used Rosenblatt's perceptron

learning algorithm. Anderson found that Rumelhart's version of back propagation outperformed other learning algorithms he investigated.

2.4 Classifier system methods

Classifier systems have also been applied to learn the multiplexer function.

Stuart Wilson wrote a paper called, "Classifier System Learning of a Boolean Function" [39, Wilson, 1986], about a system called *Boole* that learns the Boolean multiplexer function. Wilson tested a classifier system on the six, eleven, and twenty multiplexer problems. Wilson used a pared down version of what is known as the "standard" Holland system. In this simplified model Wilson used a reinforcement regime with a special sharing function. This method is the one we adopt in this thesis. In this simplified reinforcement regime, Wilson splits the match set into two parts based on the actions of the matching classifiers. The sharing function pays off correct classifiers based on their generality and the reward is shared among the winning set classifiers. The greater a classifier's generality the larger is its portion of payoff.

Wilson also tried using dynamic operator control. In his method Wilson increased or decreased the crossover rate according to a population entropy, or diversity, measure. Wilson, in his paper, emphasized the solution set classifiers, the set of most general correct rules. These are the set of rules in the rule space that always correctly identify the function. Wilson attempted to move

the population towards a lower entropy measure to allow the the population to converge on the solution set classifiers and to prevent unproductive search.

Wilson also tried slower crossover and mutation rates, and calls to the genetic algorithm for the eleven and the twenty. In his method the genetic algorithm for the eleven and the twenty was called only 1/4th the time that it was called for the six.

Wilson found improvements in learning curves for the eleven multiplexer, but he does not show graphs on results for the twenty problem. And, in another paper on his Animat system, Wilson says this about Boole and the bigger problems,

Learning times for the 11-multiplexer were somewhat longer than for the 6-multiplexer..., though not dramatically so. To probe farther, we did a preliminary experiment with the 20-multiplexer,... Under payoff-penalty, the system reached an average score above 90% within 70,000 trials; by 120,000 trials, the solution count ... reached 1200 out of a population size of 1600. This seems quite remarkable given that by 120,000 trials Boole had seen, on average, less than 1/8 of the possible input strings. Thus, it appears Boole can solve the multiplexer for longer and longer strings. But the learning times clearly increase, and in this experiment we had to increase the population size to get learning started. [40, (pp. 224) Wilson, 1987.]

Sandip Sen, a student at the University of Alabama, tried a variation on Wilson's reinforcement regime for a class project [35, Sen, Class Project]. Sen used a gift tax along with Wilson's payoff-penalty regime. Sen also tried other modes of payoff-penalty. Sen showed improved performance curves over Wilson's method alone for the six multiplexer.

CHAPTER 3

OBJECTIVE

3.1 An empirical study of classifier system performance

What are the capabilities of classifier systems? There has been much debate concerning inductive inference processes [38, Weiss and Kapouleas, 1989]. There are many computational theories about learning algorithms [29, Valiant, 1984], [5, Blumer, Ehrenfeucht, Haussler, and Warmuth, 1987]. Little has been done to practically interpret these theoretical results. On the other hand, genetic algorithm researchers have tried to identify optimal crossover rates, population sizes, mutation rates and other internal parameters [12, DeJong, 1975], [18, Grefenstette, 1986], [16, Goldberg, 1989].

An in depth analysis of an inductive learning process could lead to some clues about the phenomena that drive an inductive model. An empirical study of classifier systems could even generate some rules-of-thumb for optimal parameters for genetic algorithms. Perhaps while empirically studying classifier systems their performance could be improved.

3.2 The multiplexer function

In this thesis we study only one problem, the multiplexer function. This function belongs to the class of Boolean functions. Boolean functions are defined as the class of $\{0,1\}$ valued functions over the set of all finite strings of 0's and 1's. And from this problem class we study one function, the multiplexer function. The multiplexer function is a function that separates the input bit string, x , into two parts, an address part and a data part. The address part is decoded into an integer index. This index is used as an address into the data part to find the value of the function.

The bit string is decomposed and evaluated as follows:

$$\overbrace{a_m, a_{m-1}, \dots, a_0}^{\text{address}} \overbrace{d_0, \dots, d_{n-1}, d_n}^{\text{data}}$$

$$index = \sum a_i \times 2^i$$

$$f(x) = d_{index}$$

Table A-8 in the appendix shows an example of a six multiplexer function decoding. All figures, graphs, and tables will be listed in the Appendices in the back. Appendix A contains data for the six multiplexer along with the experimental designs and some summary information pertaining to all systems. Appendix B contains data for the eleven, and Appendix C contains data for the twenty.

The multiplexer is one representative function from a subclass of Boolean functions called k-DNF. These functions are disjunctive normal formulas of k literals defined as follows:

Let U be a set of Boolean variables, or literals,

$$U = (u_1, \dots, u_n).$$

A truth assignment for U is a function,

$$t : U \rightarrow \{T, F\}.$$

If $t(u)=T$ we say that the literal u is true and if $t(u) = F$ we say that u is false.

A clause over U is a set of literals, such as $\{u_1, \overline{u_3}, u_8\}$ which represents the conjunction of those literals and is satisfied by a truth assignment if and only if all members are true under that assignment. A disjunctive normal formula consists of one or more conjunctive clauses disjoined together to form a function.

For example:

A 3-DNF function - the six multiplexer

$$f(x) = \overline{a_0}\overline{a_1}d_0 + \overline{a_0}a_1d_1 + a_0\overline{a_1}d_2 + a_0a_1d_3$$

The function is true if at least one of the conjunctive clauses is true. In this case they are disjoint: only one can be true. For the k-DNF problem, k represents the number of literals per conjunctive clause [13, Garey and Johnson, 1979].

In this study the number of literals in the $k+1$ -DNF function the multiplexer

function, is tested for $k=2,3$, and 4 (the address lengths for the six, eleven, and twenty multiplexer problems).

3.3 Rule spaces

The rule space consists of a list of all possible rules for that size problem. The rules used are *IF-THEN* rules that contain a premise and a conclusion. In classifier systems we call the premise part of the rule the condition, and the conclusion part we call the action. The condition part of the rule is made up of characters taken from a three letter, or ternary, alphabet, $\{0,1,\#\}$. The hash mark is a wild card that will match with a 0 or 1 when matched against the environmental message. The action part of the rule is made up of characters taken from a two letter, or binary, alphabet, $\{0,1\}$.

The six multiplexer has a six letter condition part taken from $\{0,1,\#\}$, and a one letter action part taken from $\{0,1\}$.

Table A-9 shows a small sample of a rule space for the six multiplexer. The six multiplexer has a total rule space of $3^6 \times 2$, or 1458 total rules, or classifiers as they will be called. For the class of rules that (properly) predict the outcome to be 0, those strings properly leading to a 0 output are positive examples and those leading to a 1 are negative examples. Conversely, for the class of rules predicting the outcome to be 1. For example:

- $\# 0 0100 : 1$ - negative, or counter, example of the multiplexer function.

- 0 # 1100 : 1 - positive example of the multiplexer function.
- # 0 0110 : 1 - equivocal.

The eleven multiplexer has a total rule space of $3^{11} \times 2$, or 354,294 total rules. The twenty multiplexer has a total rule space of $3^{20} \times 2$, or 6.9736×10^9 total rules.

The rule length increases exponentially as a function of the address length. Adding one new input variable causes an exponential increase in the size of the problem space. This is known as the *scaling up* problem.

3.4 Instance spaces

The instance space is the space of all possible instances of the function. Instances are drawn from the alphabet containing two characters, $\{0,1\}$. The environmental message draws from this space of all instances to form its message. An instance is picked using a uniform distribution function with replacement. This means that each instance has an equal chance of becoming the environmental message at each and every trial.

The six multiplexer has a 2^6 or a 64 element instance space. Table A-10 shows a small sample of the six multiplexer's instance space. The eleven multiplexer has a 2^{11} or a 2048 element instance space. The twenty multiplexer has a 2^{20} or a 1,048,576 element instance space.

3.5 Population space

The population space is a small sample of the rule space. It contains a population of rules taken from the universe of all possible rules. The population is a space where the classifier system can test rules to find a good set. This is also called the rule, or knowledge, base. The population is often set at a constant 400 classifiers, but the population can be made to dynamically change its size. Wilson set the population sizes for the eleven multiplexer at (probably) 800 and for the twenty at 1600 [39, Wilson, 1987]. These will be the population sizes that we will use.

3.6 Examples of genetic algorithm operators

The classifier population is changed or manipulated using genetic algorithm operators. These operators are selection, crossover, and mutation.

Selection picks two parent classifiers to cross. Selection can only select rules that are already in the population rule base. This is usually done using a roulette wheel, where every classifier in the population gets a portion of the wheel based on its figure of merit. The stronger classifiers get a bigger slice of the wheel, thus more chances at being selected.

Crossover uses two-point crossover[12, DeJong, 1975] on two parent classifiers to create one offspring.

For example:

- parent1 = ##0000:0
- parent2 = 001###:1

Here the indices run $x = (x_0, \dots, x_5)$. Pick two crossover points, say point 2 and 4.

Then one of the new offspring would be:

offspring = ##1##0:0

The new offspring is completely different from either parent, although this is not always the case, and it will be added to the roulette wheel the next time selection is invoked.

The mutation operator allows a $\{0,1,\#\}$ bit to flip.

For example:

- offspring = ## 1## 0 : 0 could be mutated to:
- offspring = ## 0## 1 : 1

The mutation operator guarantees that no allele will be permanently lost from the population. It is a background operator having a very small chance, say one in a thousand bits, of being invoked. Usually only one bit per classifier would be flipped in a mutation invocation. Crossover allows for bigger steps in the rule search space.

3.7 Vose's stable, unstable schemata theorems

Genetic algorithm operators are discussed in a paper by Vose [37, Vose, 1988]. Vose analyzes a minimal deceptive problem in which he describes schemata as

stable, semi-stable, and unstable. A minimal deceptive problem occurs when the evaluation function assigns a fitness to strings in such a way that building blocks are ineffective in producing highly fit individuals [15, Goldberg, 1989].

Vose says that stable schemata, or predicates as he calls them, are unaffected by crossover. Semi-stable schemata are slowly degraded by crossover and unstable schemata are destroyed by crossover. This, of course, is based on the assigned values of the fitness functions.

For example:

For strings - {00, 11, 10, 01}

with fitness- {4, 5, 1, 3} respectively

schemata {11} is stable

schemata {00, 11} is unstable

schemata {01, 00, 11} is semi-stable

schemata {10, 01, 00, 11} is stable.

[37, (pp. 8) Vose, 1988]

The stability of the schemata affect how well the search can progress. Schemata, in turn, are affected by the string representation and the evaluation function which uses that representation.

3.8 Internal models

Internal models, such as default hierachies, are important because they can generate more parsimonious correct rule bases than would otherwise be possible. They play an important part in the development of a concept. Default rules do not necessarily have to be hiarchical. They can be any set of rules that approximates the concept to be learned.

If a creature can answer a question about a hypothetical experiment, without actually performing that experiment, then the answer must have been obtained from some submachine inside the creature. The output of that submachine(representing a correct answer) as well as the input(representing the question) must be coded descriptions of the corresponding external events or event classes. Seen through this pair of encoding and decoding channels, the internal submachine acts like the environment, and so it has the character of a "model". The inductive inference problem may then be regarded as the problem of constructing a model. [30, (pp. 449)Minsky, 1963.]

The internal model, or default set of rules, that we are the most interested in when learning the concept of the multiplexer function is the set of classifiers known as the *solution set*. The solution set classifiers represent a minimal cardinality *solution* to the k-DNF function. Finding a minimal representation of a k-DNF function is a known NP-hard problem. But in this thesis we study the solution set classifiers because they always give the right action; they correctly model the environment.

The solution set classifiers for the six multiplexer consist of $2^2 \times 2$, or 2 raised to the length of the address times 2, which would be 8 classifiers, or 2% of the population size of 400 classifiers.

Table A-11 shows the set of solution set classifiers for the six multiplexer. This is the minimal representation for the k-DNF function for the six length problem. The solution set classifiers for the eleven multiplexer consist of $2^3 \times 2$, or 16 classifiers. For a population size of 800 this would make up 2% of the population. The solution set classifiers for the twenty multiplexer consist of $2^4 \times 2$, or 32 classifiers. For a population size of 1600 this would make up 2% of the population.

CHAPTER 4

A SIMPLIFIED CLASSIFIER SYSTEM

4.1 Introduction

The following reports on three test systems. The first test system consists of an exploratory system in which a base classifier system is developed for use in further experimentation. The second test system consists of the technical modifications system in which options are tested to measure their influence on learning performance. The third test system is a special set of variations in which more complicated options are tested. Tables A-1, A-2, and A-3 show the experimental designs for the three test systems. All tables, figures, and graphs are given in the appendixes. Learning performance is measured using a running average score in which the accuracy of the system answer is recorded for the last fifty trials. The environmental message is chosen at random and scoring is done by using the average success over a suite of 50 exemplars, where a wrong answer contributes 0 to the total, a failure to match contributes 0.5, and a match with a correct answer contributes 1.0.

4.2 Methods

In the first group of experiments a simplified classifier system was used to learn the Boolean multiplexer function. This algorithm is the same as that developed by Wilson in his experiments with the Boolean multiplexer function [39, Wilson, 1986]. Wilson in [39, Wilson, 1986] used a set of 400 classifiers as the population rule base for each of the systems, the six, the eleven and the twenty. We used a population size of 800 for the eleven, and 1600 for the twenty. These sizes were chosen based on sizes chosen by Wilson in [40, Wilson, 1987] and on later evidence found in experimentations on optimal population sizes.

In general, the classifier system algorithm goes by the following format. An environmental message is presented to the system. The environmental message is a sample string from the domain of the multiplexer function, the function to be learned. Within the system is an oracle that knows what the true answer should be. The environmental message can be thought of as the input x , and the oracle can be thought of as the value of the function at x , $f(x)$. At each presentation of the environmental message classifiers within the population have their conditions matched against the bits in the message. Matching classifiers compete for the right to have their action bit become the *system answer*. The match set is then separated into two sets; those classifiers that agree in action bit with the winning classifier, and those classifiers that have the opposite action. The oracle's answer, or the true solution to the function $f(x)$ where x is the environmental message, is

checked against the system answer. If the oracle finds the same answer as that given as the system answer then the action set [A] is rewarded, otherwise the action set is penalized for a wrong answer. A classifier can accrue strength and, thus, the *utility* (or strength) of a classifier can be learned through interaction, and rewards and punishments, with the environment.

After the matching and rewarding phase, a search(or discovery) component, is invoked to search the rule space. The primary search method in classifier systems is the genetic algorithm. Wilson used something similar to what is known as a *steady state* genetic algorithm[39, Wilson, 1986]. It is given the name steady state because only one offspring is created per trial, and this offspring goes into competition with its parents. It does not replace its parents, but rather all three classifiers remain in the population together to compete at the next trial. Strength is divided up among the three of them. This is so that three classifiers remain in the population with similar genetic material, none are as strong as before, but collectively they have the same strength. The total strength of the population remains fairly constant, with only a weaker, possibly zero strength, population member being purged to make room for the new offspring, thus it gets its name steady state.

These components of a classifier system, matching, rewarding, and discovering, are collectively known as a trial. Trials are repeated until the maximum number of trials has been reached.

Through the rise and decay of classifier strengths, the population eventually

converges on a rule base. If the classifier system finds a plausible rule base then it will have been successful at *modeling* the critical aspects of its environment.

4.3 Operational description

1. Wilson used an initial population, [P], consisting of 400 randomly chosen, uniformly distributed classifiers. Solution set classifiers are purged from the initial population. We use a population size of 800 classifiers for the eleven system, and a size of 1600 classifiers for the twenty system.
2. An environmental message is presented to the classifier system. This represents an instance of the problem domain. The condition part of the classifiers in [P] are matched with the environmental message to form the match set [M].
3. A winner from [M] is chosen using a probability distribution based on strengths. This is known as noisy bidding. Classifiers compete, or bid, for the right to have their action become the system answer.
4. The set [M] is partitioned into two subsets. The set [A] is the set of classifiers from [M] that match the action of the winner. The set [NOTA] is the set of classifiers from [M] that do not match the action of the winner.
5. Reinforcement is made to the sets [A] and [NOTA] depending on whether or not the [A] set action was correct. Wilson used different schemes for reinforcement: payoff-penalty, payoff-nil, and payoff-payoff. In this study payoff-penalty is used; the correct set is rewarded and the incorrect set is penalized. Also instituted, as per Wilson, is a sharing function in the distribution of payoff to the [A] set. Classifiers in the correct winning set share in the reward with distribution of payoff being biased towards the more general classifiers in the action set, [A].
6. A genetic algorithm is called to bring in more of the rule space for testing. Two parents are chosen using a probability distribution based on strengths. The parents can be crossed, or cloned, then mutated to introduce one new offspring into the population. One classifier from [P] is purged using a probability distribution based on inverse strength. The new offspring replaces the purged classifier. New classifiers can also be introduced if the match set [M] is empty. In the case of an empty match set, [M], a new classifier is created using the genetic material of the environmental message. This is called CREATE and is described below.
7. Performance measures can be used so that internal fine tuning can occur from within the system itself during execution. Parameters, such as the

crossover or mutation rate, can be dynamically controlled during execution.

4.3.1 Genetic algorithm

1. **Selection:** The strings that will participate in crossover are chosen by spinning a roulette wheel based on classifier strengths. Each classifier in the population gets a slice of the roulette wheel that is proportional to its strength; the greater the strength the more chances at being selected. For *ringed crossover* two parents, C_1 and C_2 , are selected from the roulette wheel for reproduction. Any classifiers not in the population have no chances at being selected for reproduction.
2. **Purging:** In order for the new offspring to be added into the classifier population, a weak population member must be purged. This is done by finding a minimum strength classifier but using noise in the sorting process. This way a pool of weak classifiers is created from which to pick the member to purge.
3. **Ringed crossover:** Two-point crossover is performed probabilistically using the crossover rate. When the (crossover selection) random variable exceeds the (crossover) threshold the new offspring first copies all of the (co-parent) string C_2 . Two points are picked at random. The first point can be less or greater than the second point. Starting at the first point and ending at the second point C_1 is copied over the contents in the offspring. The copying may wrap around the end of the string and restart at the beginning.
4. **Cloning:** If the random variable is not great enough then a copy of C_1 is written into the contents of the offspring.
5. **Mutation:** The new offspring, after being crossed or cloned, is sent to the mutation operator where its alleles may be mutated to a 0, 1, or # if the probability for mutation is great enough.
6. **Action bit:** The action bit of the offspring is set to the action of C_2 if it is a result of crossover, or C_1 if it is a result of cloning.
7. **Inheritance:** If the offspring is a result of crossover then it inherits one-third of each of its parents' strengths and they are each then depleted by this amount. If the offspring is a result of cloning then it receives one-half of C_1 's strength and C_1 is depleted by one-half. Both the parents and the offspring generally remain in the population to compete against each other.

4.3.2 Create

(This routine is called when the match set, [M], is empty.)

1. The environmental message that matched no classifiers, has its condition code allele values mutated with hash marks with a probability equal to the total hash percentage of the classifier population.
2. This newly created classifier's action bit is set randomly with an equal chance of becoming a 0 or 1.
3. Its strength is set to the average strength of the population.
4. One weak population classifier is chosen to be purged to make room for the newly created classifier.
5. This newly created classifier then makes up the match set [M] and, thus, the action set [A]. Classifier system execution resumes at the reward and penalty phase.

An experimental design table is given for the three size systems; the exploratory systems, the technical modifications systems, and the variant systems (see Tables A-1 through A-3). The parameter sets (i.e. crossover, mutation rates, etc.) used for all experiments are given in Tables A-5 through A-7. If a different parameter value is used in an experiment this is stated in the description of the experiment.

4.4 Exploratory systems

To answer certain questions concerning the basic classifier system algorithm an exploratory system was run. There are questions, for instance, concerning the optimal population size for the different classifier lengths, the optimal generality of the initial classifier population, and the type of reinforcement method.

The following gives an account of the exploratory systems that were used to determine the basic classifier system algorithm.

4.4.1 Dynamic population sizes

Classifier systems borrow from natural algorithms. What determines the optimal population size for a particular problem environment? Goldberg has tried to answer this question for genetic algorithms, as well as classifier systems[16, Goldberg, 1989]. He says that the size of the population should be determined by the number of schemata that can be represented in the population and, thus, processed by the genetic algorithm. Goldberg gives a formula for determining the initial number of population schemata, assuming that the population is generated randomly, along with a formula for determining the rate of schemata processed. He says that the higher the rate of schemata processed, or the more diverse schemata in the population, the better the performance. Goldberg says that larger population sizes slow down the genetic algorithm and that a better alternative is to use a small population and restart it when it prematurely converges.

To determine the optimal population size for the multiplexer problem we tried letting the classifier system itself adaptively find an optimal population size. In this test system the classifier population was allowed to grow with the rate of no matches(CREATE rate) with the environmental message. When

the environmental message could not match with any classifiers in the classifier population, a new classifier was created from the genetic material of the environmental message. The new classifier was added to the population without purging any classifiers already in the population. The population size was increased by one for each newly created classifier. The genetic algorithm continued to purge weak population members to add new offspring to the population. This is necessary because bad classifiers still needed to be weeded out.

Figures B-1a, and C-1a show the dramatic growth rate results for the two larger systems(the six multiplexer showed little or no growth).

Cover detector(or CREATE) is a stochastic variable that depends on the number and generality of the classifiers in the population. Figures B-1a, and C-1a show that the population sizes for the eleven and the twenty remain constant for a while before beginning to grow. Hash percentage begins at 33% for a population size of 400. As the population size increases the hash percentage drops (see Figures B-1b and C-1b). Figure A-1a shows that this is not the case for the six multiplexer. This is caused by the CREATE routine which appends(in these experiments) classifiers to the end of the population. When the match set, [M], is null CREATE forms a classifier from the genetic material of the environmental message. It does this by randomly inserting hash marks into the loci of the message. This is done with a probability equal to the hash percentage of the classifier population. CREATE has a tendency to inject highly specific classifiers into the population because the exemplar's genetic material (which

is 0% hash) has a chance (and only an average chance) at becoming averagely general. Therefore, newly CREATED classifiers have a tendency for a greater than average specificity. These classifiers may or may not be good at matching future environmental messages. Cover detector injects a random rule into the population that could contain erroneous information that would have to be unlearned. The population sizes for the eleven and the twenty do not start growing until after they begin to call CREATE. The CREATE rate continues to climb and the population sizes then grow without bound. Cover detector may serve to increase the problem of not matching with the environmental message because the newly created classifier may not be general enough, it may contain erroneous information (it's randomly generated), and there is no guarantee that the population will match any better after adding this classifier (the environmental message is generated from a uniform distribution with replacement).

Cover detector is a controversial solution to the problem of a null match set. Booker[7, Booker, 1985] says that injecting a random rule into the population that has not evolved its alleles via adaptation is a questionable practice. He suggests that when the match set is null to try to rematch with the environmental message on fewer alleles, or *partial cover detector*. We chose to use Wilson's method of CREATE due to its greater simplicity. It may, however, not be the best solution and it may, in fact, cause learning to slow down.

4.4.2 Controlled initial hash percentage

Classifier systems are search methods; they search for the good rules and try to generate a plausible rule base. Because they are a search method, they have the same problems as other, more traditional, search methods. One such problem is how to determine the initial starting point in the search. The initial starting point can be crucial to the success or failure of the search method.

A test system using controlled initial population hash percentages was run. The hash marks, recall, denote the generality of a classifier. Classifiers with more hash marks are more general, or match with more environmental messages, than classifiers with fewer hash marks, or more highly specified condition parts. A completely specified classifier can be thought of as a point in $n+1$ -space, where n is the length of the condition part. Hash marks allow the classifier to represent many points in space, a plane. Changing the number of hash marks per classifier changes the the shape and distribution of the plane. The population as a whole represents a set of such planes. The initial population is the starting set. Appropriate generality is probably dependent on the recombination operators used so that it is difficult to study one parameter, such as hash percent, in isolation when other factors may interact with the parameter in question.

We ran comparative tests for the six multiplexer using initial hash percentages of 33%, 50%, and 64%. The eleven multiplexer was compared using initial hash percentages of 50%, 64%, and 75%. And the twenty multiplexer was com-

pared using initial hash percentages of 50%, 64%, and 75%.

Figures A-2, B-2, and C-2 show the results of these tests for the multiplexer problem. The graphs show that a good initial hash percent for the six is 33%, for the eleven it is 50%, and for the twenty it is 64%. Note that the initial hash percents are low compared to the *goal*, the solution set classifiers. For the six, the solution set classifiers are made up of 33% hash. The eleven's solution set has 64% hash. And the twenty's solution set has 75% hash.

A higher initial generality can cause "overgeneralization collapse" [40, Wilson, 1987]. Too many overly general classifiers in the population will cause an oscillation effect that will tend to keep average scores random. The mutation operator, which is the only other operator besides crossover used in these experiments, stays in the background at 1 mutation per 1,000 bits. Crossover, the primary operator, will then tend to select more, and more, overly general classifiers which will have been evaluated more often than highly specified classifiers.

4.4.3 Reinforcement regimes

The reinforcement regime is the method chosen for rewarding and punishing the classifiers in the match set, $[M]$, according to the accuracy of their answers. Recall that the match set, $[M]$, the set of classifiers whose condition parts match with the bits in the environmental message, is separated into the action set, $[A]$, those classifiers that agree in action with the winning classifier's action (the

system answer), and the complement of [A] the set [NOTA], those classifiers that have the opposite action of the system answer. These two sets receive rewards and penalties based on the accuracy of the system answer.

Three different reinforcement regimes are analyzed. The first method studied is the regime used by Wilson [39, Wilson,1986]. In this method, the classifiers in the match set,[M], are always taxed for matching the exemplar. For correct answers, the classifiers in the action set, [A], receive a payoff distributed among the classifiers in [A]. For incorrect answers, each classifier in [A] receives a penalty equal to a fraction of its strength. The second method studied is the regime used by Sandip Sen [35, Sen, Class Project]. In this method, the match set is not always taxed. All classifiers in the action set,[A], have a tax removed for matching. For correct answers, the [A] set is rewarded using a payoff distribution, and the [NOTA] set is then taxed for matching. For incorrect answers, the [A] set receives a penalty, and the [NOTA] set receives a gift. The third method studied is called the symmetric method. In this method, the match set is always taxed. For correct answers, the [A] set receives a payoff distribution and the [NOTA] set receives a penalty. For incorrect answers, the [NOTA] set receives a payoff and the [A] set receives a penalty.

Wilson's reinforcement regime

In the reinforcement method used by Wilson all classifiers in the set [M] have their strengths depleted for bidding. The classifier strengths in the [A] set are

either increased or decreased according to the accuracy of the answer. The [A] set alone gets a truer estimate of worth.

[M] - depleted by a small amount for bidding

[A] - increased/decreased by larger amount for action

Sandip Sen's reinforcement regime

In the reinforcement method used by Sen only the classifiers in the set [A] have their strengths depleted for bidding. The set [NOTA] is only taxed when [A] answers correctly. Both sets have their strengths adjusted according to the accuracy of the answer, but with varying degrees. So both sets, [A] and [NOTA], receive a somewhat truer estimate of worth.

[A] - depleted by a small amount for bidding

[A] - increased/decreased by larger amount for action

[NOTA] - increased/decreased by small amount for action

A symmetric reinforcement regime

In the symmetric reinforcement method all classifiers in the match set are depleted for bidding. Both sets, [A] and [NOTA], have their strengths adjusted according to the accuracy of the answer. So the set [M] gets a truer estimate of its worth.

[M] - depleted by a small amount for bidding

[M] - increased/decreased by larger amount for action

Of the three reinforcement regimes, the one used by Sen and the symmetric method show a significant improvement over the regime used by Wilson. The symmetric method was chosen over Sen's method as the optimal reinforcement method due to its greater simplicity.

Figure A-3 shows the results of these tests for the six multiplexer. Figure B-3 shows the results for the eleven. And, Figure C-3 shows the results for the twenty.

It is important to have a good reinforcement method because reinforcement guides the search of the genetic algorithm by allocating greater payoff to correct classifiers. In order for the genetic algorithm to be successful at discovering the optima of the function to be learned, it must have a pool of classifiers with good estimates of utility.

Figure A-4a shows the symmetric regime using a CREATE routine that forms a classifier from the environmental message and population hash percentage but instead of assigning a random action the newly created classifier gets the correct action. When compared to Figure A-4b, a randomly generated action bit, it is obvious that a correct action assignment unduly influences average score. Sandip Sen used the correct action assignment method, and got good scores for the six[35, Sen, Class Project], but Sen does not state explicitly how scores were computed. Wilson used the random action assignment because he said that the system should try not to use pre-programmed knowledge to avoid the problem of brittleness[40, Wilson, 1987]. We chose to use the random action

assignment because in computing average score the random action score from a newly CREATED classifier is used in the running total.

4.5 Technical modifications

The exploratory system runs were successful in finding some reasonably good settings for population size, initial population hash percentage, and reinforcement regime. The results showed that a good population size for the six multiplexer is 400, for the eleven it is 800, and for the twenty it is 1600. A good initial hash percentage for the six was found to be 33%, for the eleven it was 50%, and for the twenty it was 64%. Symmetric reinforcement was found to be the better reinforcement method.

The next set of test systems concerns the technical modification options. These were analyzed using the basic classifier system algorithm found above from running the exploratory systems. In these test systems each technical option is tested by itself to measure its affect on the running average score. Then some options are tested together to measure the influence of the combinations of these options.

4.6 Genetic algorithms

The primary discovery component in a classifier system is the genetic algorithm. The following test systems examine the impact of the choice of genetic

algorithm on classifier system learning of the Boolean multiplexer function.

4.6.1 Mate with like

The mate with like genetic algorithm speciates the classifier population into two subpopulations. The classifiers with actions that are on make up one species and the classifiers with actions that are off make up another. Only like species are allowed to mate. Each subpopulation is processed separately.

By separating the population into two groups, the genetic algorithm can search two parts of the total space at one time. Speciation also prevents competition between unrelated classifiers, or classifiers that represent different modes of the function. The multiplexer function is a multi-modal function. It has several optima not just one. The solution set classifiers are the modes, or optima, of the multiplexer function(see Table A-11).

Figures A-5b, B-4b, and C-4b show the results using mate with like. The six shows a definite improvement using this genetic algorithm compared to the base system (Figures A-5a, B-4a, and C-4a). The eleven also shows improved scores. The twenty shows some change using this method. The hash percent is decreasing and in both the six and the eleven the hash percent initially decreases before average scores improve. Because the twenty is a bigger system it may be at a different point in its evolution than the smaller systems.

4.6.2 Reduced surrogates

This method of genetic algorithm is a method used by Booker to better explore the rule space [6, Booker, 1987]. In this method, only unique offspring are allowed into the population. After two parents have been selected, crossed and mutated, the offspring is checked against its own parents, then with the other classifiers in the population to see if it is different from the other population members. If it is not unique then the parents are crossed and mutated again. This is repeated until a unique offspring is created or until a timeout has been reached. If no unique offspring have been created by the timeout then two new parents are selected for reproduction. The process is repeated until a unique offspring has been added to the population or until the entire population has been selected for reproduction.

The reduced surrogates genetic algorithm forces the population to become more diverse. This is advantageous for bigger systems because it forces the population to represent more of the larger rule space.

Figures A-5c, B-4c, and C-4c show the results using reduced surrogates. The six is degraded using this method. This is because the rule space for the six is very small. If the population is forced to diverge then it will lose the problem solution, it will be unable to converge on the eight optima. The eleven and the twenty systems are positively influenced by this method due to the fact that their rule spaces are large. Premature convergence will not allow the bigger

systems to find the sixteen and thirty-two optima that represent the function. Diversity, or premature convergence prevention, allows the system to explore a greater range of the rule space which is productive for systems with larger rule spaces.

4.6.3 Reduced surrogates and mate with like

The reduced surrogates method of genetic algorithm was tested using speciation on the action bit, or mate with like.

In this method the population was separated into two subpopulations of actions on and off. The reduced surrogates genetic algorithm was performed on these two subpopulations. The most diverse classifiers from each action subpopulation were added to the classifier population.

Figures A-5d, B-4d, and C-4d show the results. Again the six is degraded by this method. The eleven shows definite improvement and the twenty could be improving.

4.7 Operators

Some of the technical modifications studied were the use of new recombination operators. Operators are functional operations that modify classifier alleles. Crossover and mutation are two examples of genetic algorithm operators. CREATE is an example of an environmental operator.

4.7.1 Bit flipping

The bit flip operator is used during the reinforcement phase of the classifier system. This operator is used only if the action set [A] gives the incorrect answer but can be used symmetrically on [NOTA] when [A] is correct. For an incorrect system answer the same payoff and penalty is instituted according to the reinforcement regime chosen, but the classifiers in the [A] set have their action bit modified(flipped) to give the “correct” answer(i.e. the answer that the environmental message gave). In these experiments the bit flip operator is used with a probability for flipping the action bit of 50%. The bit flip operator is used probabilistically, and is not implemented every time the system answer is wrong.

Bit flipping causes incorrect classifiers that match with the exemplar to flip their actions to the opposite action, thereby matching the action of the exemplar. And, should that classifier match with that exemplar again, and become a member of the [A] set, it will then have acquired the correct action. Bit flipping depends on matching with the exemplar in the future. Therefore, systems with a small instance space are more apt to be positively influenced by the bit flip operator than systems, such as the twenty multiplexer, that have enormous instance space sizes and rarely view the same exemplar twice.

And, bit flipping, because it is only concerned with the action part of the classifier, does not necessarily create “good” classifiers. For example:

- ##### : 0 is an incorrect classifier in [A].
- 001000 is the exemplar.
- ##### : 1 is the newly modified classifier.

Bit flipping can sometimes be futile. It is a more severe form of recombination operator because, with only one operation, it changes the niche of the classifier. The classifier then goes on to compete with other classifiers in a new niche without any prior indication that it will do well. Bit flipping changes the consequence of the premise in the production rule. It does not alter the “knowledge” in the rule but rather forces the rule to blindly form a different conclusion.

Figures A-6, B-5, and C-5 show the results when using the bit flip operator. The six multiplexer, with the smallest instance space size, shows the greatest effect when compared to the base system, Figure A-5a. The eleven multiplexer is also positively influenced by bit flipping. It, too, has a small instance space size. But the twenty, with an instance space size in the millions, is degraded by this operator. This is not surprising considering that bit flip depends on re-matching and that the twenty has a larger number of niches to discover. Bit flipping may interfere with niche formation.

Mate with like and bit flipping

Mate with like genetic algorithm was tested using the bit flip operator.

Speciation should always act as an improver because genetically non-related

classifiers should not compete with each other for payoff. Each mode of the multiplexer function should establish its own niche, and allow species to grow and compete only with members of the same species. Bit flipping, because it operates on the action bit (and, in this case, speciation bit) should act to complement the mate with like genetic algorithm. If a classifier is incorrect and its action bit gets flipped this would cause the classifier to move to, or to create, a different niche. This should serve to improve the whole population, not just the incorrect classifiers, because it should allow for the discovery and formation of niches.

Figures A-6b, B-5b, and C-5b show the results. The six always learns the function so it is difficult to detect a real effect. The eleven seems to learn earlier. The twenty seems degraded using this operator combination. Bit flipping may confuse the genetic algorithm if it is causing oscillating effects.

Reduced surrogates and bit flipping

Reduced surrogates genetic algorithm was tested using the bit flip operator. Reduced surrogates creates a diverse classifier population. Bit flipping should be enhanced by population diversity. This is because the chance for oscillating patterns of bit flipping the action of too general classifiers is reduced. The population is divergent, therefore, flipping the bit will have a greater probability of creating a correct classifier.

The combination of creating diverse classifiers using reduced surrogates and

bit flipping the action bit of incorrect classifiers should improve performance over that of using reduced surrogates alone.

Figures A-6c, B-5c, and C-5c show the results. The six shows degradation in average scores. The eleven shows learning but not the best scores seen so far. The twenty seems unaffected.

Reduced surrogates, mate with like, and bit flipping

Reduced surrogates using mate with like was tested using the bit flip operator. In this case the genetic algorithm operates on a diverse population that is speciated on the action bit. This should cause the population to become even more diverse because speciation should force the number of classifiers in a niche to remain small (they compete with each other). The multiplexer function is a multi-modal function. The number of optima, or niches, is equal to the cardinality of the solution set. Bit flip does not automatically create correct classifiers.

Figures A-6d, B-5d, and C-5d show the results. The six is degraded, the eleven is improved, and the twenty is unaffected.

4.7.2 Relative complement

Another operator that is analyzed for its effect on performance is the relative complement operator. This is an operator that is used only on incorrect classifiers. The action set classifiers in $[A]$, that do not match the exemplar's action, will each have one of their hash marks mutated to the opposite of the

allele value in the exemplar at that same loci. For example:

- 0#1#01:0 is an incorrect classifier in [A].
- 011101 is the exemplar.
- 001#01:1 is the newly modified classifier.

This operator causes classifiers to become more specific.

Matching classifiers that do not have the same action as the exemplar can no longer match with that exemplar. Relative complement differs from bit flipping in that it does not depend on re-matching with the exemplar a second time in order to be effective. Relative complement changes the content of the premise in an attempt to make the rule more relevant to the concept to be learned. In the case that the classifier is completely specified(i.e. contains no hash marks) the action of the incorrect classifier would have its action bit flipped.

By forcing a hash mark to become specific and opposite to the bit at the same position of the exemplar, the classifier will be unable to match that exemplar again and give the incorrect action a second time. After several such trials using the relative complement operator incorrect classifiers are forced to properly represent the function to be learned. Relative complement does create “good” classifiers, because it changes the premise part of the rule to inject better knowledge into the classifier. And, because relative complement does not depend on matching with the exemplar a second time, it is not affected by the size of the instance space of the problem. Relative complement works productively

for any size system.

Figures A-7a, B-6a, and C-6a show the results using the relative complement operator. Both the six and the eleven are very positively influenced by this operator. The twenty is not. The twenty's hash percentage drops. But by looking at the CREATE rates (these statistics were collected but not graphed) it is evident that there are other forces at work besides this operator in isolation. The CREATE rate increases dramatically for the twenty, which means that the match set is small; a singleton set consisting of the one newly CREATED classifier. The eleven multiplexer's CREATE rate first rises then levels off—this is what would be expected.

Relative complement, which operates on all of an incorrect [A] set, cannot be very effective if it operates on one classifier at a time because it only changes one allele per classifier. Relative complement requires a bigger action set than just the singleton case to have any noticeable effect. The CREATE rate can also explain the drop in hash percent in the twenty system due to the fact that CREATE tends to inject highly specific classifiers into the population. The eleven, as well as the six, see an initial decline in hash which could be due to relative complement exchanging hash for specific bits. And as the hash percent rises the average scores also rise. This means that relative complement would not be used because relative complement only works on incorrect classifiers. Crossover, the only other primary operator, tends to create more general classifiers(i.e. higher hash percents) due to the fact that selection is biased towards more general clas-

sifiers that match more often and receive more of the reward due to the payoff bias. In both the six and the eleven as the hash percent rises so does the average score. As average scores rise (more correct [A] sets) the relative complement operator decreases in invocations (it only operates on incorrect [A] sets).

Mate with like and relative complement

Mate with like genetic algorithm was tested using the relative complement operator. Relative complement can improve classifier system performance no matter what the size of the problem space. The combination of relative complement with mate with like should not be as dynamic as bit flipping, though, because relative complement does not concern itself with speciation or niche formation. Recursive application of relative complement tends to make correct classifiers. A good operator, such as relative complement, used along with a good genetic algorithm, such as mate with like, should improve performance over that when the two methods are used alone.

Figures A-7b, B-6b, and C-6b show the results using this combination. All systems are positively influenced using this combination. The eleven shows a dramatic improvement.

Reduced surrogates and relative complement

Reduced surrogates was tested using the relative complement operator. Reduced surrogates creates a diverse population. Relative complement is a reinforcement

operator which only acts on classifiers that have made it into the match set. It is highly likely that the match set sizes for a diverse population will be very much smaller than the match set size for a converged population. Relative complement may not have as big an impact on systems using this method of genetic algorithm, because it only changes one hash bit per classifier for every classifier in an incorrect action set.

Figures A-7c, B-6c, and C-6c show the results. The eleven is very positively influenced using this combination. This probably has something to do with the size of the match set. The eleven has a smaller instance space than the twenty therefore it should have a larger matching rate as well as size. Both the twenty and the eleven are positively influenced by the reduced surrogates genetic algorithm due to their larger population sizes and the tendency for large populations to prematurely converge.

Reduced surrogates, mate with like, and relative complement

Reduced surrogates using speciation was tested using the relative complement operator. Once again the relative complement operator should have a smaller impact when used with diversity creating genetic algorithms. This is due to a smaller match set.

Figures A-7d, B-6d, and C-6d show the results. The eleven seems to be the best influenced by using this method. The twenty system shows a slight upward trend at the end of 15,000 trials. The six shows mediocre performance. Again,

too much diversity in the population causes unproductive search and crossover in the six.

4.8 Dynamic parameter control

Wilson experimented with dynamic crossover control in his classifier system, Boole [39, Wilson, 1986]. He used an internal measure of the diversity of the population as an entropy measure. When the population was highly diverse (high entropy) he turned up the crossover rate, and when the population had converged (low entropy) he turned down the crossover rate to alleviate the problem of unproductive crossover. His method for computing the entropy measure was highly computer intensive because he used something called the macrostate to analyze the number of unique classifiers in the population. He found in tests using the eleven multiplexer that dynamically controlling the crossover rate was effective in increasing the learning speed, but at a cost in CPU dollars.

Davis also used a form of internal parameter control. In Davis' method the probability (or rate) for the operator (in this case only crossover and mutation) was determined by using the operator's performance level [11, Davis, 1989]. The operator's ability to create good classifiers was used as its probability for being used again. This method is much less CPU intensive than Wilson's macrostate method, however, it suffers from something known as the credit assignment problem [30, Minsky, 1963]. The credit assignment problem concerns the problem

of rewarding the cause of the good effect as opposed to rewarding the immediate source. The currently successful classifier may have become a good rule through many transformations, not just the previous one.

4.8.1 Dynamic operator control

To analyze the effect of dynamic operator control using a method similar to Davis', dynamic control was instituted on only the following operators: crossover, bit flip, and relative complement. The mutation operator was not dynamically controlled for in these experiments because it used as a background operator.

A waiting period of the first 200 trials was used in which dynamic control was turned off. This was done to allow the operators to accrue some invocations. Each operator, crossover, bit flip, and relative complement, began with an initial probability of being invoked. Crossover had a probability of 12% in the six multiplexer, and 25% in the eleven and the twenty systems. The probability for calling the genetic algorithm was turned up to 100% in the bigger systems. Ordinarily they have a slower genetic algorithm rate than the six at 25%. The flags to use the relative complement and bit flip operators were turned on and the probability for flip the bit was set to 50%, as before. Only one out of the three operators could be invoked per trial.

After the first 200 trials the choice was made. The performance measure was

updated every 50th trial. The population was rematched using evaluation on demand. As the classifiers matched with the exemplar their opcodes, which were previously recorded as: r - relative complement, x - crossover, b - bit flip, and c - initial population member or create; were noted and also whether the classifier was correct or not. A flag was also set in each matching classifier structure so that it would only be counted once. When all members of the population had been matched, operator scores were then computed as the number of correct classifiers with a particular opcode divided by the total number of classifiers with that opcode. This ratio is then used as the probability for invoking that operator in the future - the operator's performance measure.

Figures A-8a, B-7a, and C-7a show the results using dynamic operator control. Figures A-9a, B-8a, and C-8a show the operator performance.

The six shows faster learning but at the expense of solution set classifiers. The eleven shows the greatest improvement. And the twenty shows that there is some perturbation although it does not look like a trend. The curves for the operators show that crossover dominates while bit flip is the lowest performance operator.

Mate with like and dynamic operator control

Mate with like genetic algorithm was tested using dynamic operator control for crossover, bit flip, and relative complement operators. Speciation is always beneficial to the genetic algorithm. Combining speciation along with opera-

tor performance control should cause learning speeds to increase, because the genetic algorithm is getting feedback on how well crossover is performing on speciated(action bit) populations. However, in this case it is an either-or situation. One of the three operators is chosen as the next operator. Crossover and relative complement cannot occur on the same trial. If relative complement takes over, which is quite likely(it is a successful operator), then crossover will slow down. The same is true for bit flipping and crossover. It is very possible that these two reinforcement operators could drive the crossover operator out.

Figures A-8b, B-7b, and C-7b show the results for mate with like with dynamic operator control. Both the six and the eleven show good average scores, the twenty seems unaffected. Operator scores are shown in Figures A-9b, B-8b, and C-8b. Crossover becomes the dominant operator as average scores increase and bit flip and relative complement are no longer necessary(i.e. the [A] set is correct).

Reduced surrogates and dynamic operator control

Reduced surrogates was used along with dynamic operator control for crossover, bit flip and relative complement. Reduced surrogates creates a diverse population. The operator performance measures using dynamic control give the system feedback via crossover performance on how well the genetic algorithm is performing. The diversity and the feedback should work complementarily to improve learning speeds.

Figures A-8c, B-7c, and C-7c show the results. All three systems seem to be positively influenced using this combination. Figures A-9c, B-8c, and C-8c show the operator performance curves.

Reduced surrogates, mate with like, and dynamic operator control

Reduced surrogates using speciation is tested using dynamic operator control for crossover, bit flip, and relative complement.

Figures A-8d, B-7d, and C-7d show the results. Both the eleven and the twenty show improvements in average scores.

4.9 Training set generation

Ordinarily, the environmental message is presented to the system using a uniform distribution function with replacement. At each and every trial any string in the instance space has an equal chance to become the next exemplar. The bigger systems have instance space sizes that grow exponentially larger for each new input variable. Therefore, the larger systems have a bad matching rate. The bigger systems tend to have high CREATE rates. Questions then arise concerning what is driving the system; the CREATE routine, or the adaptive mechanisms of the classifier system algorithm. The following test systems analyzed a method of generating training examples.

4.9.1 Evaluation on demand

The environmental message is taken from the genetic material of the classifier population. The purpose of this is to try to get a more accurate evaluation of the classifiers. When exemplars are generated this way, there is always a non-empty match set. The force that is driving the system is that of the adaptive mechanisms and not the CREATE routine because CREATE need not be invoked. The training set that is being used to allow learning then is a controlled set(i.e. the training set is not random). Each classifier in the population is used, **alternately with a randomly picked exemplar**, as a template for an exemplar. This exemplar is shown to the system during the match phase and then matched against the population. Rewards and punishments are given for correct or incorrect answers as before. This is repeated for all classifiers in each population at trial t . Only after all members of the population at trial t have been used as the exemplar template, matched, and their behavior reinforced is the next phase in the system taken which is the genetic algorithm. In this way, all classifiers in any population at trial t will have been exercised at least one time before the genetic algorithm is invoked.

When using exemplars on demand average scores are computed using a testing phase, which is separate from the classifier system algorithm. In the test phase the population is not changed. It is shown an exemplar from a uniform random distribution with replacement. Average scores are computed as an aver-

age over fifty test trials on all random exemplars. If CREATE needs to be called a new classifier is not added to the population but a coin is flipped to determine its contribution to the average score. During the training phase first an exemplar is taken from the genetic material of a population member, then the next exemplar is randomly generated, so that during training CREATE may have to be invoked and a new classifier generated. But during testing we need to get a true measure of utility by testing the population on all random exemplars. So there is no need to change the population during testing, we are simply looking at the ability of the population to correctly classify exemplars that it was not trained on.

The reason why this method should serve to improve performance is because classifiers that are exercised(matched and reinforced) more often have a truer estimate of their worth than classifiers that are exercised little or not at all. This guarantees a minimal evaluation rate for all the classifiers in the population. This helps the genetic algorithm, which works off the premise that classifiers selected for reproduction have a good estimate of their utility. The genetic algorithm assumes it is selecting from a pool of evaluated individuals. The genetic algorithm should work more efficiently because it has better information.

Figures A-10a, B-9a, and C-9a show the results using this method of exemplar generation. This method shows some degradation in performance for the six and the eleven when compared to other methods that we have tried. It is difficult to judge the twenty system. It is possible that evaluation on demand

when used alone is not a dramatic improver of average score, but when used with another operator may have a synergistic effect. For instance, when evaluation on demand is used with a particular genetic algorithm the two could complement one another.

Mate with like and evaluation on demand

Mate with like genetic algorithm was tested using evaluation on demand. Evaluation on demand is a non-random method of generating the training examples. Using a speciated population's genetic material to generate training examples should improve performance over that using evaluation on demand alone because it gives the genetic algorithm even more information; the selection pool is well evaluated, and the selection pool is speciated on the action bit.

Figures A-10b, B-9b, and C-9b show the results. The six and the eleven show good scores, although not the best. The twenty seems to possibly have a trend at the end of 15,000 trials so this system was re-run out to 70,000 trials to see what would happen to the average scores. In this experiment(not graphed) the trend seen in the earlier tests did not continue. The system stays at around 70% accuracy with the hash percentage declining.

Reduced surrogates and evaluation on demand

Reduced surrogates was tested using evaluation on demand. Reduced surrogates creates a diverse classifier population. Evaluation on demand picks from this

diverse population classifiers to use as the exemplar template. Diverse classifiers in the population are then able to become well evaluated. The genetic algorithm selects classifiers for reproduction from a diverse, well-evaluated population. This combination should give the genetic algorithm more information about the concept to be learned because the population spans a wider range of points and the points have better estimates of their fitness.

Figures A-10c, B-9c, and C-9c show the results. This combination does not seem to do well. This could be due to the fact that the average scores are taken from a less diverse space than that which is formed by the reduced surrogates genetic algorithm. A diverse population may not test well on a narrow test set.

Reduced surrogates, mate with like, and evaluation on demand

Reduced surrogates using speciation(on the action bit) was tested using evaluation on demand. Reduced surrogates creates a diverse population. Speciation allows for niche formation. Evaluation on demand gives the genetic algorithm a better estimate of a classifier's worth. This should work together to improve learning speed. Figures A-10d, B-9d, and C-9d show the results. The six is degraded due to the reduced surrogates genetic algorithm. But both the eleven and the twenty systems show increased learning later in the run.

The twenty system was run out to 70,000 trials(Figure C-12c). Its learning curve is slow and erratic over the first 30,000 trials. It smooths out to an average score of around 100% accuracy after 40,000 trials. This system also has a gradual

initial decline in hash percentage followed by an increase until it peaks at a little over 60%. The solution set classifiers begin to grow at around the 34,000th trial and continue to slowly grow until the end of maximum trials.

4.10 Dynamic operator control combined with evaluation on demand

Mate with like, dynamic operator control, and evaluation on demand

Mate with like genetic algorithm was tested using dynamic operator control for crossover, bit flip, and relative complement operators along with evaluation on demand. This combination of three options should be very complementary. Evaluation on demand gives the genetic algorithm better evaluated classifiers to select from for reproduction. Dynamic operator control gives the genetic algorithm feedback on how well crossover is performing. Relative complement creates correct classifiers.

As before when using evaluation on demand, the average score performance results are obtained using a test phase in which the system is tested in a 50 trial phase using randomly generated exemplars.

Figures A-11a, B-10a, and C-10a show the results. The six shows faster learning rates at the price of solution set classifiers, while the eleven shows faster learning rates with the addition of more solution set classifiers. The twenty seems to be in a small upward trend throughout the 15,000 trials. So

this system was run out to a maximum number of trials of 70,000.

The trend seen in the earlier graphs continues on in its upward pattern(see Figure C-12b). The system reaches 80% to 90% accuracy at around 40,000 trials. The initial hash percentage in this system, as in all other twenty multiplexer systems not using hash percent as its option, is set at 64%. This number drops until the 15,000th trial when it begins to increase until at 70,000 trials it is a little over 60%. The solution set classifiers begin to grow in number at the 30,000th trial. They continue to grow until half of the 1600 member population is made up of the minimal correct set.

Reduced surrogates, dynamic operator control, and evaluation on demand

Reduced surrogates was used along with dynamic operator control and evaluation on demand. A diverse population with genetic algorithm feedback and well-evaluated classifiers for the genetic algorithm to select from should be a good combination. The population spans a greater space, and the points in that space have a better estimate of merit, and if the genetic algorithm is not making progress, then other operators can take over until the crossover operator can gain another foothold.

Figures A-11c, B-10c, and C-10c show the results. The six learns the function although the curve is more erratic. The eleven learns the function at around 5,000 trials. And the twenty shows a possible upward trend at 15,000 trials.

Reduced surrogates, mate with like, dynamic operator control, and evaluation on demand

Reduced surrogates using speciation was tested using dynamic operator control and evaluation on demand.

Figures A-11d, B-10d, and C-10d show the results.

The twenty system showed a learning trend using this method in 15,000 trials. This system was run out to 70,000 trials(Figure C-12a). In this system the learning curve is slow and erratic. It sees solution set classifier growth at around 20,000 trials. This growth continues although at a smaller rate due to the fact that reduced surrogates genetic algorithm would prohibit duplicate classifiers in the population. The hash percentage also sees an initial decline followed by a gradual increase to a little over 60%. This twenty system achieves 90-100% accuracy at the 30,000th trial.

4.11 Variant systems

In the next group of experiments variations on the methods for operator, dynamic control, and exemplar generation are made and their effects analyzed. In all of these systems the reduced surrogates genetic algorithm is used.

4.11.1 Variant I

In this variation the initial population is generated not by using a hash percentage probability but rather by using a generality threshold. Each classifier in the initial population can only have a certain number of specified bits, the rest must be hash marks. In Variant I for the six multiplexer the threshold was set at two; only two bits could be a 0 or a 1 in any classifier in the initial population. For the eleven the generality threshold was set at seven, and for the twenty it was set at thirteen.

Figures A-12a, B-11a, and C-11a show the results. The six learns the function, although the average score shows greater variation. The eleven's learning speed slows down dramatically using Variant I. The twenty shows the usual random score.

4.11.2 Variant II

In this method the same generality threshold factors are used to generate the initial populations. And, in addition, the exemplars are generated using the classifier population as templates, as in evaluation on demand.

Figures A-12b, B-11b, and C-11b show the results. The six shows better learning than in Variant I. The eleven shows no obvious improvement. And, the twenty shows no change.

4.11.3 Variant III

In this method the same routine for initial population generation was used, a generality threshold. And a mate with like algorithm was instituted.

Figures A-12c, B-11c, and C-11c show the results. The six shows learning but not as accurate when using a genetic algorithm that does not create diversity. The eleven shows that it is improved by using speciation on the action bit. The twenty shows no change.

4.11.4 Variant IV

Variant IV used the same method for generating the initial population as used in Variant I. Mate with like is used as in Variant III. And exemplar generation as in Variant II is used only this time the templates are queued. This means that as new classifiers are created using reproduction or CREATE they are assigned to a FIFO queue for evaluation on demand.

Figures A-12d, B-11d, and C-11d show the results. The six is improved using a FIFO queue for the training set. The eleven shows that, it too, is improved using an ordered training set. And, the twenty shows improvement.

The twenty system in 15,000 trials actually showed a learning trend using this method. This system was also run out to 70,000 trials(Figure C-12d). In this test the initial hash percentage was determined by the generality threshold. This began at around 65% initial hash. This is seen to quickly decline to only

about 45% at the 10,000th trial. It then increases to about 64% and stays there. The solution set classifiers begin to grow at about the 15,000th trial. The system achieves 90-100% accuracy at around the 25,000th trial. This system can be seen to be markedly different from the other above methods run out to 70,000(i.e. smoother, faster learning).

4.12 Discussion

Table A-4 summarizes the results found using the above test systems. One obvious result found is that reduced surrogates does not work well for the six multiplexer. This is because the six has a smaller rule space and so needs to converge on the solution set classifiers early on. The eleven and the twenty systems have a larger rule space and also larger population sizes so that premature convergence must be actively prevented in these systems if they are to be able to learn the concept.

Mate with like, or speciation, is a requirement for learning the multiplexer concept. This is due to the fact that the multiplexer function is a multi-modal function which requires the use of niche formation so that unrelated rules won't compete for same resources. Sharing during payoff to the [A] set also allows for niche formation by dividing up the reward among similar classifiers.

The relative complement operator is the best operator at turning negative examples of the rule into positive examples. However, in order for relative

complement to be effective it must work on large match sets. The twenty, which generally has smaller sized match sets, is not as dramatically affected by this operator.

Exemplars on demand works best when used with a queueing method. Variant IV uses a FIFO queue on exemplar templates. This showed a significant improvement on systems performance. The queueing forces evaluation on demand to be even more selective in creating a spanning training set. This means that the choice of training set when building minimal, accurate rule bases is very important and that the number of times a particular exemplar is presented (repetitions) also has an effect.

CHAPTER 5

A PURE GENETIC ALGORITHM

5.1 Introduction

The next set of tests uses a pure genetic algorithm to learn the multiplexer function. The experimental design for these systems is given in Table A-12. The method of determining average score performance is the same as that used in the classifier system. Learning performance is measured using a running average score in which the accuracy of the system answer is recorded for the last fifty trials. The environmental message is chosen at random and scoring is done by using the average success over a suite of 50 exemplars, where a wrong answer contributes 0 to the total, a failure to match contributes 0.5, and a match with a correct answer contributes 1.0.

5.2 Methods

In the second group of experiments a pure genetic algorithm was used to learn the multiplexer function. This method used a genetic algorithm alone, without the stimulus-response mechanism of a classifier system, to find the optima of the function. A Goldberg, "generational" genetic algorithm[15, Goldberg, 1989] is

implemented in which many new offspring can be created at each new generation.

This method begins by first testing the population to see how well it performs. The population is then trained using cover detector (CREATE) to create new classifiers for null match sets.

The genetic algorithm assigns classifiers an estimate of utility using an evaluation function. Each classifier is assigned a functional value that is predetermined beforehand. This is called a static evaluation function. This method of learning is much different than that of a classifier system. A classifier system uses reinforcement learning to evolve an estimate of classifier utility. It learns the evaluation function as it builds the rule base. A genetic algorithm does not have to learn the function, this is pre-programmed.

5.3 Operational description

1. Initialization

- An initial population, [P], of 400 classifiers is selected using a specific initial hash percentage rate for the population. Solution set classifiers are purged from the initial population. The eleven uses an initial population size of 800 and the twenty uses 1600 classifiers.

2. Evaluation

- The classifiers have their strengths computed based on a static evaluation function.

3. Testing

- An environmental message is presented to the pure genetic algorithm. The condition part of the classifiers in [P] are matched with the environmental message to form the match set [M].
- If the match set is empty then ordinarily CREATE would be invoked to form a matching classifier. But during the testing phase the population is not changed so instead a coin is flipped to determine what the probable action of a newly CREATEd classifier would be.

- If the match set is non-empty then a winner from [M] is chosen using a probability distribution based on classifier strength.

4. Training

- An environmental message is presented to the pure genetic algorithm. The condition part of the classifiers in [P] are matched with the environmental message to form the match set [M].
- If the match set is empty then CREATE is called to form a matching classifier. The newly CREATEed classifier replaces a weak population member.

5. Genetic Algorithm

- The genetic algorithm is called after evaluation, training and testing have been completed.
- In a Goldberg-like genetic algorithm many new offspring can be created at each new generation.

5.3.1 Genetic algorithm

1. **Selection:** The strings that will participate in crossover are chosen by spinning a roulette wheel based on classifier strengths. Each classifier in the population gets a slice of the roulette wheel that is proportional to its strength; the greater the strength the more chances at being selected. For *ringed crossover* two parents, C_1 and C_2 , are selected from the roulette wheel for reproduction. Any classifiers not in the population have no chances at being selected for reproduction.
2. **Purging:** In order for the new offspring to be added into the classifier population, a weak population member must be purged. This is done by finding a minimum strength classifier using noise in the sorting process. This way a pool of weak classifiers is created from which to pick the member to purge.
3. **Ringed crossover:** Two-point crossover is performed probabilistically using the crossover rate. When the probability for crossover is great enough C_2 is copied into the new offspring. Two points on the offspring are picked at random. The first point can be less or greater than the second point. Starting at the first point and ending at the second point C_1 is copied over the contents in the offspring. The copying may wrap around the end of the string and restart at the beginning.
4. **Cloning:** If the probability for crossover is not great enough then the entire contents of C_1 are copied into the offspring.

5. **Mutation:** The new offspring, after being crossed or cloned, is sent to the mutation operator where its alleles may be mutated to a 0, 1, or # if the probability for mutation is great enough.
6. **Action bit:** The action bit of the offspring is set to the action of C_2 if it is a result of crossover, or C_1 if it is a result of cloning.
7. **Strength:** The offspring's strength is computed using the static evaluation function.

5.3.2 Create

(This routine is called when the match set, $[M]$, is empty.)

1. The environmental message that matched no classifiers, has its condition code allele values mutated with hash marks with a probability equal to the total hash percentage of the classifier population.
2. This newly created classifier's action bit is set randomly with an equal chance of becoming a 0 or 1.
3. Its strength is computed using the static evaluation function.
4. One weak population classifier is chosen to be purged to make room for the newly created classifier.
5. This newly created classifier then makes up the match set $[M]$ and, thus, the action set $[A]$ and the bid winner.

5.4 Static evaluation functions

Three static evaluation functions are analyzed. The three different functions are used to show how difficult it can be to choose a good evaluation function.

The first function defined will be called F1. F1 consists of two parts:

1. %correct
 - Each data bit specified by the address(if the address is incompletely specified then fill in the hashes with all possible combinations) that matches with the action bit(i.e. has the correct action) counts as one and falls under the heading of a fixed correct bit.

- Each data bit pointed to by the address that consists of a hash mark counts as one-half and falls under the heading of hashes pointed to.
- Each hash mark in the address portion counts as one and is added to a counter called A.

2. generality

- Each hash mark in the data portion counts as one and is added to the counter called N.

$$f_1(x) = (\%correct)^\alpha \times (generality)^\beta, 1 \leq \alpha \leq 2 \text{ and } 0 \leq \beta \leq \infty$$

$$\begin{aligned} \%correct &= (0.5)^A \times \left(\sum_{fixedcorrect} 1.0 + \sum_{hashptd.to} 0.5 \right) \\ generality &= N + 1 \end{aligned}$$

Strength is based on the interaction of classifier correctness with classifier generality. In this function, the solution set classifiers (Wilson's perfect set) receive the highest estimation of worth while incorrect, specific classifiers receive the lowest. Hash marks in the address portion weaken a classifier's strength, so classifiers with fully specified addresses have higher strengths. Hashes in the data portion pointed to by the address receive half a point due to the fact that half the time that allele will be wrong and the other half right. Correct bits receive a full point, and incorrect bits receive no points.

The hierarchical rule-ordering for F1 is shown in Figure A-14a. This figure shows the three multiplexer evaluations using the F1 function. The percentages indicate the percent correctness of the classifier when all bit positions are filled in. The floating point number in the last column to the right indicates the F1 evaluation of the classifier. The figure shows that hash marks in the address bits, as well as highly specific classifiers, are penalized.

The parameter set for the pure genetic algorithm tests is given in Table A-13. In all of the pure genetic algorithm runs the optimal parameters found in the classifier system, such as initial population sizes, and initial hash percents, were also used for the pure genetic algorithm. And the maximum trials were set to only 10,000 due to the fact that the pure genetic algorithm did not have to learn the utility function.

Figures A-13a, B-12a, and C-13a show the results using F1 and a plain(non-speciated) Goldberg reproduction scheme. In these tests the average score initially rises then falls off. This is due to the fact that the genetic algorithm is an optimization technique. It finds the peak of the static evaluation function. In the case of a multi-peaked function, such as the multiplexer function, the genetic algorithm must use some form of speciation, or sharing, on all the peaks of the function [17, Goldberg and Richardson, 1987]. When nonrelated classifiers are forced to compete for the same resources one species will take over.

In this case the number of distinct species would be equal to the number of solution set classifiers. For the six, there are 8 optima, or peaks, for the eleven there are 16 optima, and for the twenty there are 32 optima. Figure A-16a shows the *macrostate* for the six multiplexer. The macrostate is a term used by Wilson to list unique classifiers in the population [39, Wilson, 1987]. In this test system the genetic algorithm finds only one optima. This is caused by genetic drift. Even though there are many high strength classifiers, one will be continually allotted more and more of the selection trials. This then causes too many copies

of one classifier to emerge in the population, and the population drifts towards only one peak [17, Goldberg and Richardson, 1987].

The second static evaluation function used was F2 and is defined as follows:

$$f_2(x) = \left(\sum_{\text{correctptd.to}} 1.0 - \sum_{\text{incorrectptd.to}} 0.1 \right) \times 2^{\text{datahash}}$$

This was a simpler function than F1 and its hierarchical rule-ordering for the three multiplexer is shown in Figure A-14b. This shows that F2 mistakenly gives some weight to incorrect classifiers and places highly specific, but correct, classifiers above those that are general and correct half the time.

Figures A-13b, B-12b, and C-13b show the results for F2. This function does not work as well as F1.

The third static evaluation function was even simpler than the first two functions. This function, known as F3, is defined as follows:

If the classifier contains hash marks then :

$$f_3(x) = \frac{\sum_{\text{correctptd.to}} 1.0}{(\sum_{\text{correctptd.to}} 1.0 + \sum_{\text{incorrectptd.to}} 0.1)}$$

If the classifier contains no hash marks then :

$$f_3(x) = \frac{\sum_{\text{correctptd.to}} 1.0}{(\sum_{\text{correctptd.to}} 1.0 + \sum_{\text{incorrectptd.to}} 0.1)} \div 2.5$$

The hierarchical rule-ordering for F3 using the three multiplexer problem is shown in Figure A-14c. This shows that F3 gives a truer estimate of merit than F2, but may not allow for epistasis, or dependency among alleles, as in F1. Figures A-13c, B-12c, and C-13c show the results for F3. This function works very well for the six multiplexer.

5.5 Genetic algorithms

5.5.1 Mate with like

Mate with like is implemented the same as for the classifier system with the exception that due to generational crossover many more offspring are reproduced at each trial. As usual, only classifiers proscribing the same action are allowed to produce offspring.

Figures A-15a, B-13a, and C-14a show the results. These results show that initially the average scores are very good, but they again tend to drop off. Figure A-16a shows the macrostate for the six multiplexer using F1 without speciation. This shows that one classifier takes over the population and so that only one peak of the function is found. As trials continue this one classifier takes up all the room in the population preventing it from spanning the instance space. Gradually the system's average scores decline due to an increased CREATE rate caused by non-matching. Figure A16-b shows the macrostate for F1 with speciation. This shows that two classifiers have taken over the population. Mate with like speciates on two species, actions on and actions off, so that the genetic algorithm now finds two peaks of the function. With more speciation, such as using the address bits, more optima of the function could be found.

5.6 Discussion

It is evident that speciation, or sharing, should be done for all the optima of the function. Goldberg and Richardson were right when they proposed that multi-modal evaluation functions require some form of sharing [17, Goldberg and Richardson, 1987]. Goldberg and Richardson used a sharing function based on computed Hamming distances between binary strings. We found this to be too computationally complex for systems using a ternary alphabet. We tried incorporating small steps in the evaluation function to compensate for sharing. We found the trivial problem, the six, was learnable using a genetic algorithm and a simple evaluation function. The larger systems require a more complicated approach. What should be noted is that the simplified classifier system in chapter 4 was able to evolve an evaluation function for the eleven and the twenty systems, something that we were unable to do.

CHAPTER 6

SUMMARY AND CONCLUSIONS

To summarize, we first used an exploratory classifier system to find initial settings for learning the six, eleven, and twenty multiplexer problem. Using the exploratory systems we found that initial population hash percentage does influence later learning. We found initial settings to be 33% for the six, 50% for the eleven, and 64% for the twenty. We also found that the choice of reinforcement method impacts learning. We chose the simpler symmetric reinforcement method over Sen's method, although both were equally good. We found that initial population size has an effect on learning. We chose settings of 400 for the six, 800 for the eleven, and 1600 for the twenty. This gave an equal amount of space for the differing solution set sizes(2%), and these were the sizes that Wilson used in [40, Wilson, 1987].

Next, technical modification options were tried to see their effect. We tried various genetic algorithms, recombination operators, dynamic internal parameter control, and training set generation. We found that a genetic algorithm that used speciation on the action bit did better than one without speciation. The multiplexer is a multi-modal function. We found that the large system, the twenty, had better performance using a diversity-creating genetic algorithm,

such as reduced surrogates, because this kept the large population(1600 classifiers) from prematurely converging. We found that relative complement, that does not depend on matching with the training example a second time, is a better operator than bit flipping, which does. We found that when trying to dynamically control internal parameters that it is difficult to define the performance of an operator due to the credit assignment problem. We found that training set generation impacted genetic algorithm performance. Better evaluated classifiers give the genetic algorithm better information with which to search the rule space. And, we found that dynamic operator control and evaluation on-demand were necessary for the twenty multiplexer to learn the function.

Later, we used a plain genetic algorithm to learn the multiplexer problem. This is a ternary genetic algorithm. We found that it is difficult to choose a good static evaluation function—it is easier to let a classifier system find it. And, sharing or speciation, or some strategy, in the evaluation function must account for all of the optima of the function in a multi-modal problem.

In conclusion, we found that the scaling up problem is more than just an exponential growth in the search space. We found that the twenty multiplexer simply could not learn the function using the solution as was used in the six. As the problem size was increased the problem dynamics were different and required different solutions. Bit flipping was not an effective operator for the twenty with its massive instance space, but bit flipping was an effective operator for the six. Reduced surrogates, a diversity creating genetic algorithm, was not an effective

choice for the trivial problem, the six, but was an effective search method for the twenty. The same solution, the simple algorithm, could not be applied to the same problem, scaled-up, because as the problem grew larger it became a different problem. In other words, we had to start from scratch with each new added input variable.

Because this was an empirical study, we did not focus on the underlying mechanisms of this learning algorithm. But before classifier systems can be useful on real world problems some theory needs to be generated on why the problem as it scales up becomes different, and how these differences can be anticipated.

BIBLIOGRAPHY

BIBLIOGRAPHY

- [1] Charles W. Anderson.
Learning and Problem Solving with Multilayer Connectionist Systems.
PhD thesis, University of Massachusetts, 1986.
- [2] Dana Angluin and Carl H. Smith.
Inductive inference: theory and methods.
Computer Surveys, 15(3):237-269, September 1983.
- [3] Scott Austin.
An introduction to genetic algorithms.
AI Expert, 5:49-53, March 1990.
- [4] Avron Barr and Edward A. Feigenbaum.
The Handbook of Artificial Intelligence: Volume I.
Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1981.
- [5] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth.
Occam's razor.
Information Processing Letters, 24:377-380, April 1987.
- [6] Lashon Booker.
Genetic Algorithms and Simulated Annealing, chapter 5, pages 61-73.
Pitman Publishing, London, 1987.
- [7] Lashon B. Booker.
Improving the performance of genetic algorithms in classifier systems.
In *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, pages 80-92, 1985.
- [8] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone.
Classification and Regression Trees.
Wadsworth International Group, Belmont, California, 1984.
- [9] Bruce G. Buchanan.
Can machine learning offer anything to expert systems?
Machine Learning, 4:251-254, December 1989.
- [10] Maureen Caudill and Charles Butler.
Naturally Intelligent Systems.
MIT Press, Cambridge, Massachusetts, 1990.

- [11] Lawrence Davis.
Adapting operator probabilities in genetic algorithms.
In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 61–67, 1989.
- [12] K.A. DeJong.
Analysis of the Behavior of a Class of Genetic Adaptive Systems.
PhD thesis, University of Michigan, 1975.
- [13] Michael R. Garey and David S. Johnson.
Computers and Intractability: A Guide to the Theory of NP-Completeness.
W.H. Freeman and Company, San Francisco, 1979.
- [14] E. M. Gold.
Language identification in the limit.
Inf. Control, 10:447–474, 1967.
- [15] David E. Goldberg.
Genetic Algorithms in Search, Optimization, and Machine Learning.
Addison-Wesley Pub. Co., Reading, Massachusetts, 1989.
- [16] David E. Goldberg.
Sizing populations for serial and parallel genetic algorithms.
In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 70–85, 1989.
- [17] David E. Goldberg and Jon Richardson.
Genetic algorithms with sharing for multimodal function optimization.
In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 41–49, 1987.
- [18] John J. Grefenstette.
Optimization of control parameters for genetic algorithms.
Transactions on Systems, Man, and Cybernetics, SMC-16:122–128, January 1986.
- [19] Paul Harmon and David King.
Expert Systems: Artificial Intelligence in Business.
John Wiley and Sons, Inc., New York, 1985.
- [20] John H. Holland.
Adaptation in Natural and Artificial Systems.
University of Michigan Press, Ann Arbor, Michigan, 1975.
- [21] John H. Holland.
Escaping brittleness.
In R. S. Mickalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning, Volume 2*, Morgan Kaufmann, Los Altos, California, 1986.

- [22] John H. Holland.
Properties of the bucket brigade algorithm.
In *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, pages 1-7, 1985.
- [23] John H. Holland, Keith J. Holyoak, Richard E. Nisbett, and Paul R. Thagard.
Induction: Processes of Inference, Learning, and Discovery.
The MIT Press, Cambridge, Massachusetts, 1986.
- [24] John E. Hopcroft and Jeffrey D. Ullman.
Introduction to Automata Theory, Languages, and Computation.
Addison-Wesley Publishing Company, Reading, Massachusetts, 1979.
- [25] Earl B. Hunt.
Concept Learning: An Information Processing Problem.
John Wiley and Sons, Inc., New York, 1962.
- [26] Earl B. Hunt, Janet Marin, and Philip J. Stone.
Experiments in Induction.
Academic Press, New York, 1966.
- [27] Robert A. Jacobs.
Increased rates of convergence through learning rate adaptation.
Neural Networks, 1:295-307, January 1988.
- [28] Stephen Judd.
Neural Network Design and the Complexity of Learning.
MIT Press, Cambridge, 1990.
- [29] Michael Kearns, Ming Li, Leonard Pitt, and Leslie Valiant.
On the learnability of boolean formulae.
19th STOC, 19:285-295, May 1987.
- [30] Marvin Minsky.
Steps toward artificial intelligence.
In Edward A. Feigenbaum and Julian Feldman, editors, *Computers and Thought*, McGraw-Hill Book Company, Inc., New York, 1963.
- [31] J. Ross Quinlan.
Induction over large data bases.
Technical Report HPP-79-14, Stanford University, Palo Alto, California, 1979.
Heuristic Programming Project.

- [32] J. Ross Quinlan.
Learning efficient classification procedures and their application to chess end games.
In *Machine Learning: An Artificial Intelligence Approach*, chapter 15, pages 463–482, Morgan Kauffman Publishers, Inc., California, 1983.
- [33] J.R. Quinlan.
An empirical comparison of genetic and decision-tree classifiers.
In *Proceedings of the Fifth International Machine Learning Conference*, pages 135–141, 1988.
- [34] Elaine Rich.
Artificial Intelligence.
McGraw-Hill Book Company, New York, 1983.
- [35] Sandip Sen.
Classifier system learning of multiplexer function.
Class Project, The University of Alabama.
- [36] Paul E. Utgoff.
Incremental induction of decision trees.
Machine Learning, 4:161–186, November 1989.
- [37] Michael D. Vose.
Generalizing the Notion of Schema in Genetic Algorithms.
Technical Report, University of Tennessee, Knoxville, Tennessee, 1988.
Computer Science Department.
- [38] Sholom M. Weiss and Ioannis Kapouleas.
An empirical comparison of pattern recognition, neural nets, and machine learning classification methods.
In *IJCAI*, pages 781–787, 1989.
- [39] Stewart W. Wilson.
Classifier System Learning of a Boolean Function.
Research Memo RIS No. 27r, The Rowland Institute for Science, 100 Cambridge Parkway, Cambridge, Massachusetts 02142 USA, February 1986.
Submitted for Publication.
- [40] Stewart W. Wilson.
Classifier systems and the animat problem.
Machine Learning, 2:199–228, May 1987.

APPENDIXES

APPENDIX A

SIX MULTIPLEXER DATA SETS

Table A-1. Experimental design for the exploratory systems

Test System	Dyn. Pop Sizes	Initial Pop. Size	Initial HashPct	CREATE Action bit	Reinf. Regime
1	*Yes	400	33%	random	Wilson
2	No	400,800,1600	*33% — 75%	random	Wilson
3	No	400,800,1600	33%,50%,64%	random	*Wilson, Sen, Symmetric
4	No	400,800,1600	33%,50%,64%	*correct	Wilson

* - variable tested

Table A-2. Experimental design for the technical modifications systems

Test System	Bit Flipping	Rel. Complem.	Mate w/ Like	Reduced Surrogate	Dyn. Op. Control	Evaluation on Demand
1	X	-	-	-	-	-
2	-	X	-	-	-	-
3	X	X	-	-	X	-
4	-	-	-	-	-	X
5	-	-	X	-	-	-
6	-	-	-	X	-	-
7	-	-	X	X	-	-
8	X	-	X	-	-	-
9	-	X	X	-	-	-
10	-	-	X	-	-	X
11	X	X	X	-	X	-
12	X	X	X	-	X	X
13	X	-	-	X	-	-
14	-	X	-	X	-	-
15	-	-	-	X	-	X
16	X	X	-	X	X	-
17	X	X	-	X	X	X
18	X	-	X	X	-	-
19	-	X	X	X	-	-
20	-	-	X	X	-	X
21	X	X	X	X	X	-
22	X	X	X	X	X	X

X - variable tested
(-) - variable not tested

Table A-3. Experimental design for the variant systems

Test System	Bit Flip	Relative Complement	Reduced Surrogate	Mate w/ Like	Dyn. Op. Control	Evaluation on Demand
1	-	-	X	-	-	-
2	-	-	X	-	-	X
3	-	-	X	X	-	-
4	X	X	X	X	X	X FIFO

X - variable tested
(-) - variable not tested

Table A-4. Summary of performance results for the six, eleven, and twenty multiplexers

Table Entry	Test Option	Six	Eleven	Twenty
A-2.1	Bit flipping	high avg. score; clean top end	slow learning; wide interval	no trend
A-2.2	Rel. complement	cleaner top end;	faster learning; cleaner top end	no trend
A-2.3	Dyn. op. ctrl.	clean top end; no sol. set cfs	fast learning; smoother curve	no trend
A-2.4	Eval. on demand	delayed learning after 1,500 trials; clean top end	delayed learning after 10,000 trials;	no trend
A-2.5	Mate with like	cleaner top end; all sol.set cfs at 7,500 trials	clean top end delayed learning after 6,000 trials;	no trend
A-2.6	Red. surrogates	lower avg. scores; wide interval	clean top end faster learning at 3,000 trials;	slight upward trend
A-2.7	Red. surg. and mate w/like	degraded learning	clean top end faster learning; clean top end; some sol.set cfs	slight upward trend
A-2.8	Mate w/like and bit flipping	fast learning under 1,500 trials; all sol.set cfs at 5,000 trials	fast learning; wide interval	no trend
A-2.9	Mate w/like and rel.complement	fast learning; clean top end	fast learning at 1,500 trials; clean top end	no trend
A-2.10	Mate w/like and eval.on demand	clean top end; delayed learning after 1,000 trials; sol.set after 12,000	delayed learning; wide interval	slight upward trend at 15,000 trials
A-2.11	Mate w/like and dyn.operator	clean top end; no sol.set cfs	faster learning at 3,000 trials some sol.set cfs	no trend
A-2.12	Mate w/like, dyn.op., and eval.on demand	clean top end; delayed sol.set cfs	delayed learning after 3,000 trials sol.set cfs;	slight upward trend
A-2.13	Red.surrogates, and bit flipping	wide interval;	cleaner top end delayed learning after 12,000 trials wide interval;	no trend
A-2.14	Red.surrogates, and rel.complement	wide interval;	faster learning; cleaner top end	no trend
A-2.15	Red.surrogates, and eval. on demand	wide interval; delayed learning	slight upward trend after 12,000 trials	no trend
A-2.16	Red.surrogates, and dyn.op.	clean top end;	faster learning at 4,000 trials; cleaner top end	no trend
A-2.17	Red.surrogates, dyn.op., and eval. on demand	clean top end;	faster learning; clean top end	slight upward trend
A-2.18	Red.surrogates, mate w/like and bit flipping	wide interval; lower avg. scores	faster learning at 3,500 trials; clean top end	no trend
A-2.19	Red.surrogates, mate w/like and rel.complement	wide interval; lower scores	faster learning at 3,000 trials; clean top end some sol.set cfs	no trend
A-2.20	Red.surrogates, mate w/like and eval.on demand	slower learning; wider interval	delayed learning after 6,000 trials; clean top end	slight upward trend at 15,000 trials
A-2.21	Red.surrogates, mate w/like and dyn.operator	faster learning; clean top end	faster learning at 3,000 trials; clean top end	slight upward trend at 15,000 trials
A-2.22	Red.surrogates, mate w/like, dyn.operator, and eval. on demand	faster learning; clean top end; some sol.set cfs	faster learning at 3,000 trials; clean top end some sol.set cfs	definite upward trend at 10,000 trials
A-3.1	Variant I	delayed learning after 3,000 trials; wide interval	slight upward trend at 13,000 trials	no trend
A-3.2	Variant II	delayed learning; wide interval	slight upward trend at 15,000 trials	no trend
A-3.3	Variant III	lower avg. scores; wide interval	delayed learning after 5,000 trials; clean top end	slight upward trend at 15,000 trials
A-3.4	Variant IV	faster learning at 4,000 trials; clean top end	faster learning at 2,500 trials; cleaner top end	definite upward trend at 5,000 trials; wide interval

Table A-5. Parameter set for the exploratory systems

Parameter	Setting
Payoff to [A]	1,000.00
Estimated fraction from [M]	0.10
Tax fraction from [NOTA]	0.10
Penalty fraction for wrong answers	0.89
P' - penalty prime	0.0
G - generality payoff pressure	4.0
Gift tax to [NOTA]	0.25
Pnoise - payoff noise	0.0
Uniform random seed	987654321
CREATE - set cover used	yes
Correct action on set cover	no
Initial classifier distribution	by allele
Initial population size	*400
Dynamic population size	*yes
Initial classifier strength	1,000.0
Initial hash percent	*
Initially purge solution set classifiers	yes
Maximum trials	15,000
Test period(random exemplars)	0
GA probability	1.0(6),0.25(11,20)
Mutation probability	0.001
Crossover probability	0.12
Roulette wheel selection	yes
Purge noise for classifier replacement	$\pm 5.0(6), \pm 10.0(11), \pm 20.0(20)$
Number of offspring from GA	1
Noisy bidding in [M]	yes
Reinforcement regime	*
Genetic algorithm	Wilson's steady state ga

* - parameter is tested

Table A-6. Parameter set for the technical modifications systems

Parameter	Setting
Payoff to [A]	1,000.00
Estimated fraction from [M]	0.10
Tax fraction from [NOTA]	0.10
Penalty fraction for wrong answers	0.89
G - generality payoff pressure	4.0
P ¹ - penalty prime	0.0
Gift tax to [NOTA]	0.25
Pnoise - payoff noise	0.0
Uniform random seed	987654321
CREATE - set cover used	yes
Correct action on set cover	no
Initial classifier distribution	by allele
Initial population size	400(6),800(11),1600(20)
Dynamic population size	no
Initial classifier strength	1,000.0
Initial hash percent	33%(6),50%(11),64%(20)
Initially purge solution set classifiers	yes
Maximum trials	15,000
Testing period(random exemplars)	0
GA probability	1.0(6),0.25(11,20)
Mutation probability	0.001
Crossover probability	0.12
Roulette wheel selection	yes
Purge noise for classifier replacement	±5.0(6),±10.0(11),±20.0(20)
Number of offspring	1
Noisy bidding in [M]	yes
Reinforcement regime	symmetric
Genetic algorithm	Wilson's steady state ga

Table A-7. Parameter set for the variant systems

Parameter	Setting
Payoff to [A]	1,000.00
Estimated fraction from [M]	0.10
Tax fraction from [NOTA]	0.10
Penalty fraction for wrong answers	0.89
P' - penalty prime	0.0
G - generality payoff pressure	4.0
Gift tax to [NOTA]	0.25
Pnoise - payoff noise	0.0
Uniform random seed	987654321
CREATE - set cover used	yes
Correct action on set cover	no
Initial classifier distribution	g-threshold
Generality threshold	2(6),7(11),13(20)
Initial population size	400(6),800(11),1600(20)
Dynamic population size	no
Initial classifier strength	1,000.0
Initially purge solution set classifiers	yes
Maximum trials	15,000
Testing period(random exemplars)	0
GA probability	1.0(6),0.25(11,20)
Mutation probability	0.001
Crossover probability	0.12
Roulette wheel selection	yes
Purge noise for classifier replacement	$\pm 5.0(6), \pm 10.0(11), \pm 20.0(20)$
Number of offspring	1
Noisy bidding in [M]	yes
Reinforcement regime	symmetric
Genetic algorithm	*reduced surrogates ga

* - parameter is tested

Table A-8. Example of a six multiplexer

int.	classifier	addr. part	data part	integer index	$f(x) \leftrightarrow$ cfs
0	0 0 0 0 0 0 : 0	0 0	0 0 0 0	[0]	0 $\leftrightarrow$ 0
0	0 0 0 0 0 0 : 1	0 0	0 0 0 0	[0]	0 $\leftrightarrow$ 1
63	1 1 1 1 1 1 : 0	1 1	1 1 1 1	[3]	1 $\leftrightarrow$ 0
63	1 1 1 1 1 1 : 1	1 1	1 1 1 1	[3]	1 $\leftrightarrow$ 1

Table A-9. The six multiplexer's rule space

integer	condition	action	identifies function
0	0 0 0 0 0 0 :	0	yes
.	.	.	.
.	.	.	.
728	## ## ## ## :	0	yes
729	0 0 0 0 0 0 :	1	no
.	.	.	.
.	.	.	.
1458	## ## ## ## :	1	yes

Table A-10. The six multiplexer's instance space

integer x	instance x	$f(x)$
0	0 0 0 0 0 0	0
1	0 0 0 0 0 1	0
.	.	.
.	.	.
62	1 1 1 1 1 0	0
63	1 1 1 1 1 1	1

Table A-11. The solution set classifiers for the six multiplexer

address	data	action
00	0###:	0
00	1###:	1
01	#0##:	0
01	#1##:	1
10	##0#:	0
10	##1#:	1
11	###0:	0
11	###1:	1

Table A-12. Experimental design for the pure genetic algorithm

Test System	F1	F2	F3	Mate w/ Like
1	X	-	-	-
2	-	X	-	-
3	-	-	X	-
4	X	-	-	X
5	-	X	-	X
6	-	-	X	X

Table A-13. Parameter set for the pure genetic algorithm

Parameter	Setting
Uniform random seed	987654321
CREATE - set cover used	yes
Correct action on set cover	no
Initial classifier distribution	by allele
Initial population size	400(6),800(11),1600(20)
Dynamic population size	no
Initial classifier strength	stat. eval.fcn.
Initial hash percent	33%(6),50%(11),64%(20)
Initially purge solution set classifiers	yes
Maximum trials	10,000
Testing period(random exemplars)	50
GA probability	1.0
Mutation probability	0.001
Crossover probability	0.12
Roulette wheel selection	yes
Purge noise for classifier replacement	$\pm 5.0(6), \pm 10.0(11), \pm 20.0(20)$
Number of offspring from GA	200(6),400(11),800(20)
Noisy bidding in [M]	yes
Genetic algorithm	Wilson's steady state ga

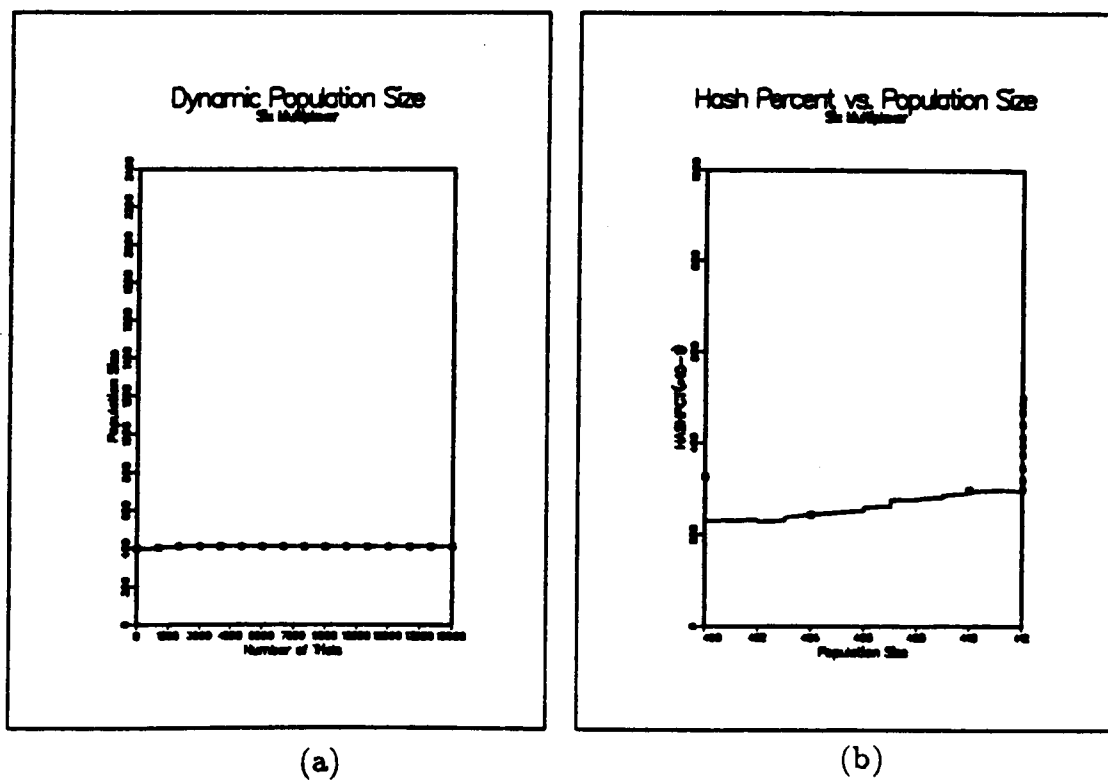


Figure A-1. Dynamic population size and hash percentage for the six multiplexer.

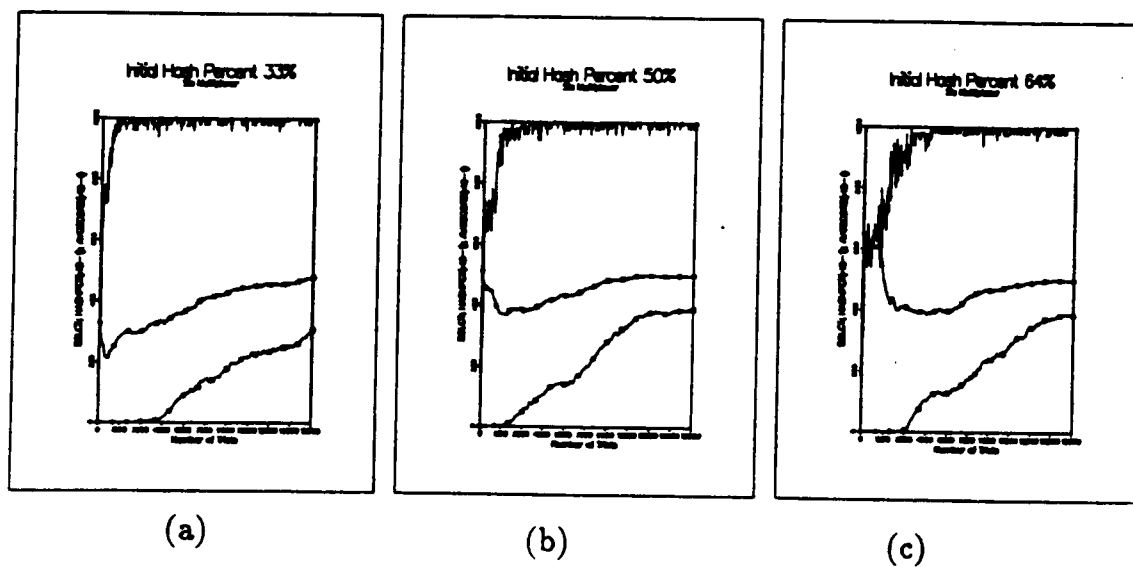
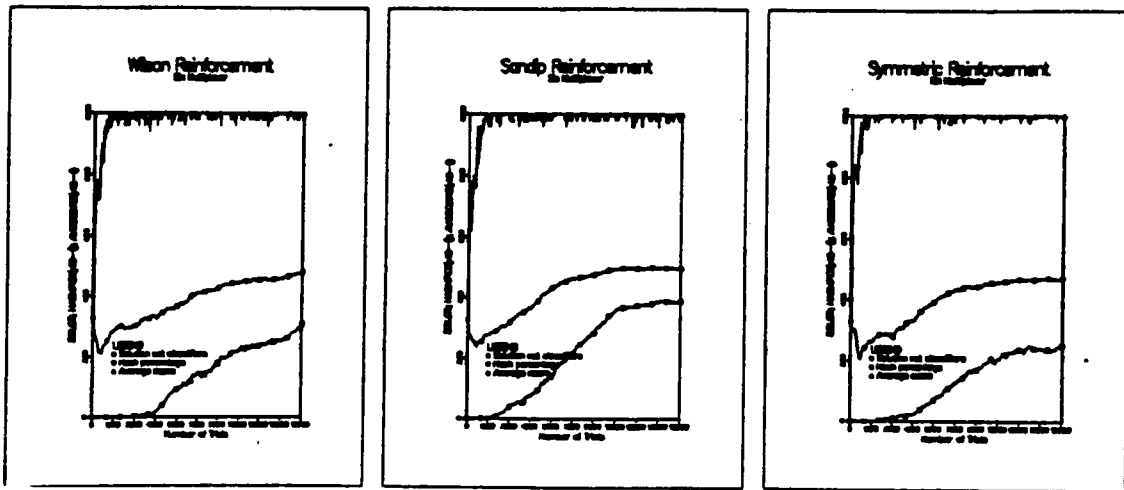


Figure A-2. Controlled initial hash percentage for the six multiplexer.

¹Note: Top curve is the average score, middle curve is the hash percentage, bottom curve is the number of solution set classifiers.



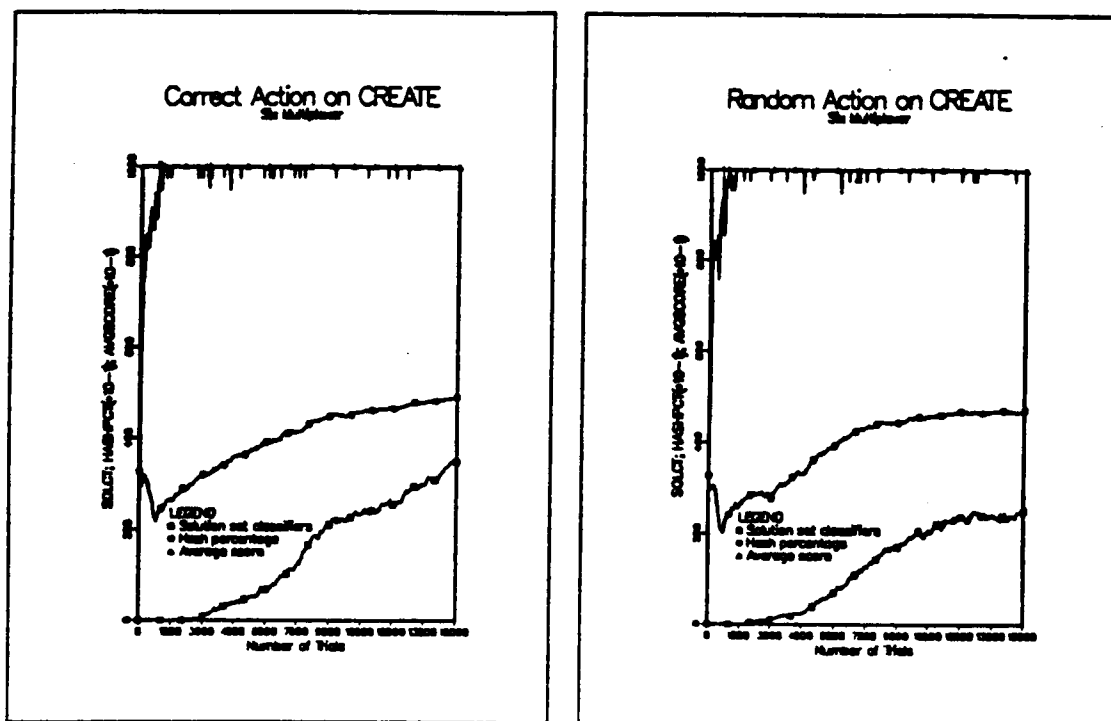
(a)

(b)

(c)

Figure A-3. Wilson, Sen, and symmetric reinforcement for the six multiplexer.

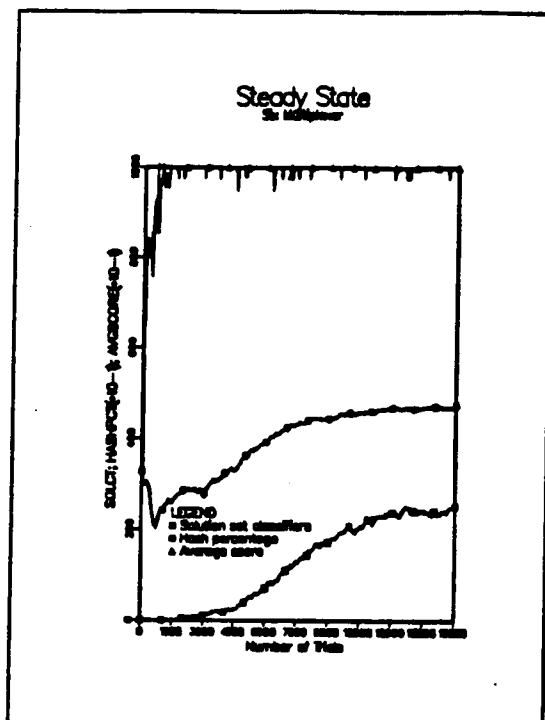
²Note: Top curve is the average score, middle curve is the hash percentage, bottom curve is the number of solution set classifiers.



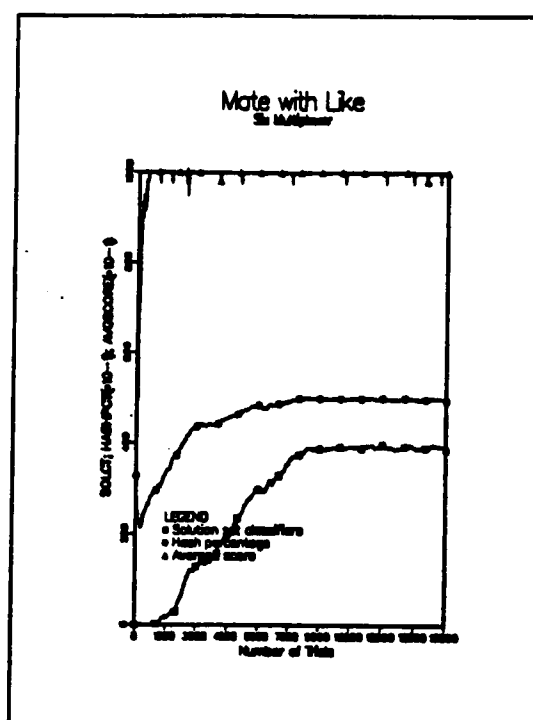
(a)

(b)

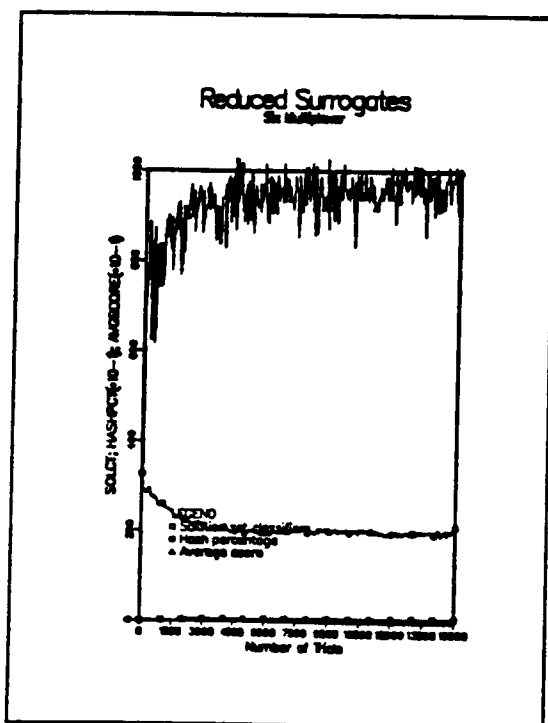
Figure A-4. Correct and random action on CREATE for the six multiplexer.



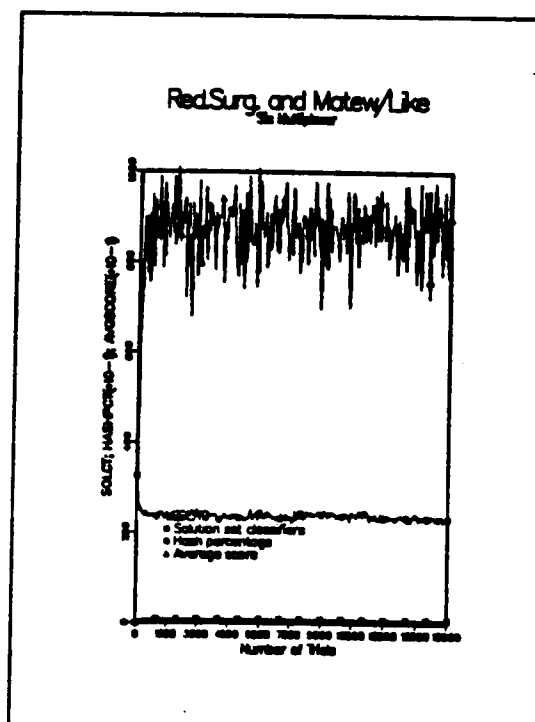
(a)



(b)



(c)



(d)

Figure A-5. Genetic algorithms for the six multiplexer.

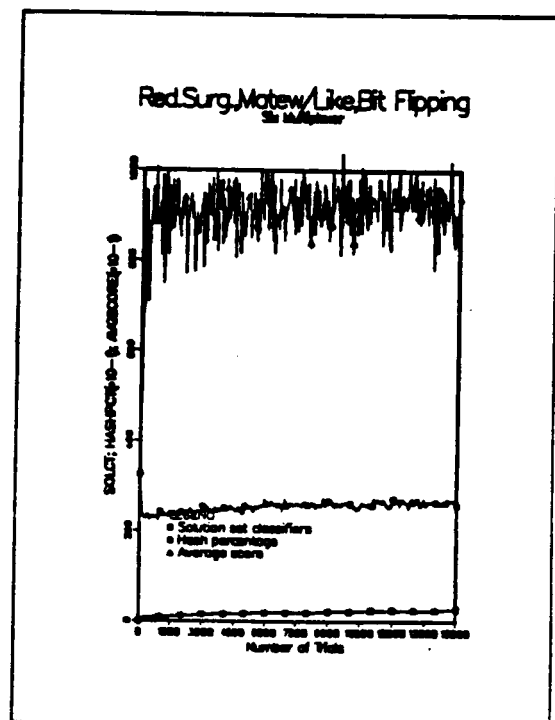
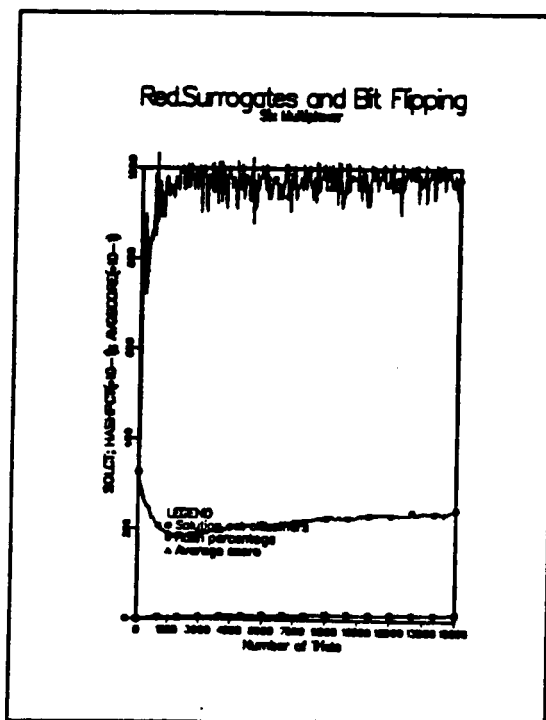
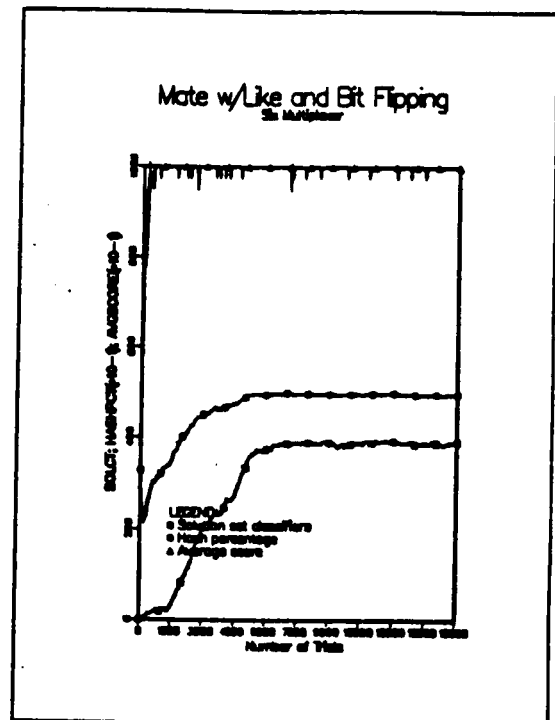
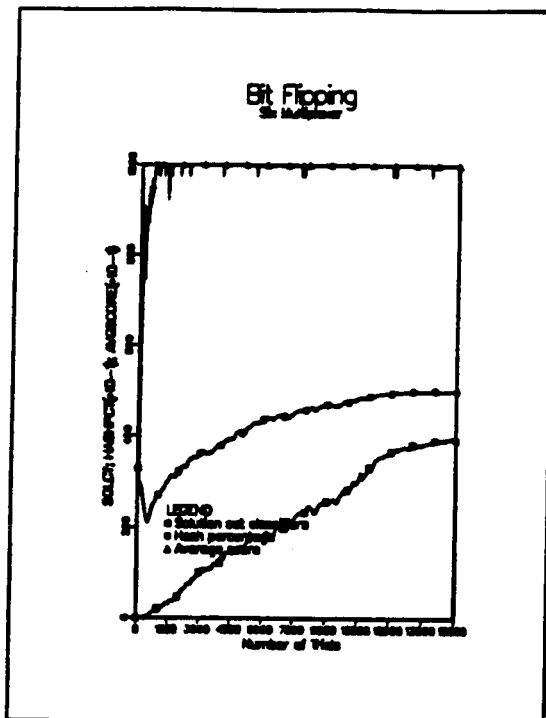


Figure A-6. Bit flipping operator for the six multiplexer.

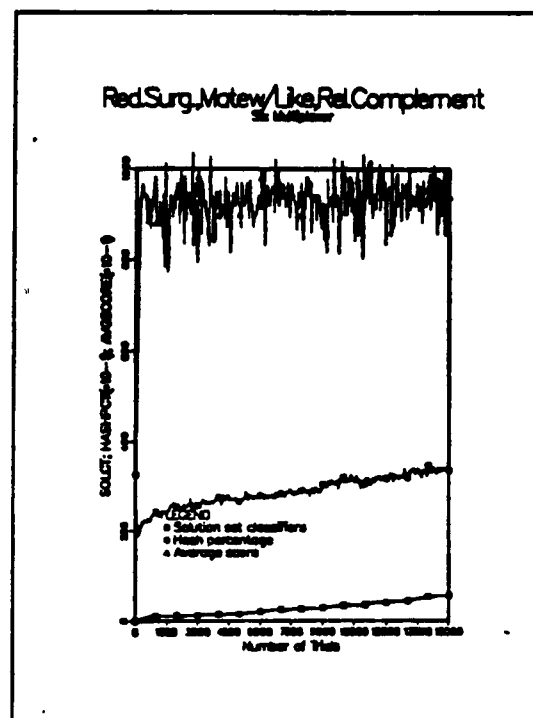
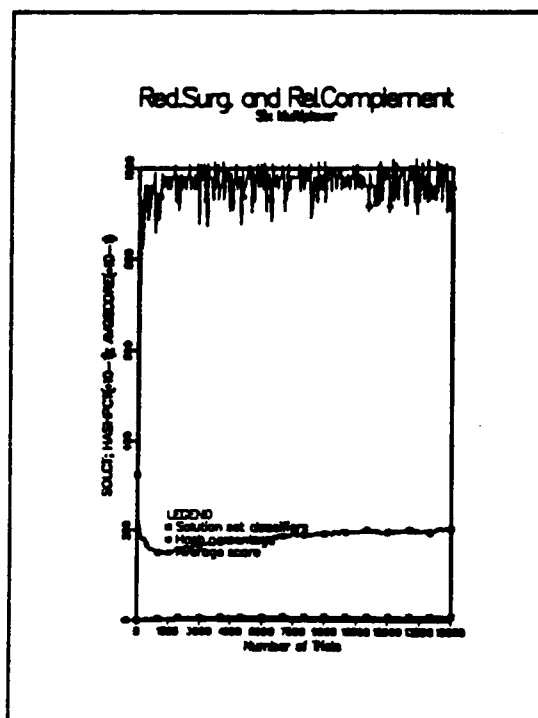
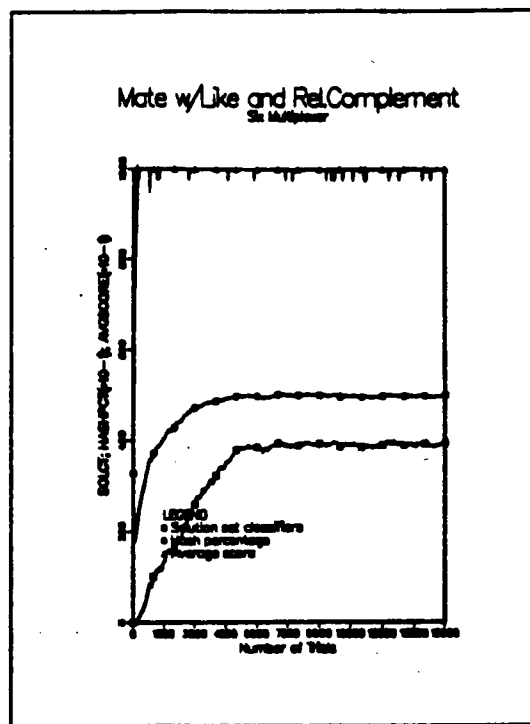
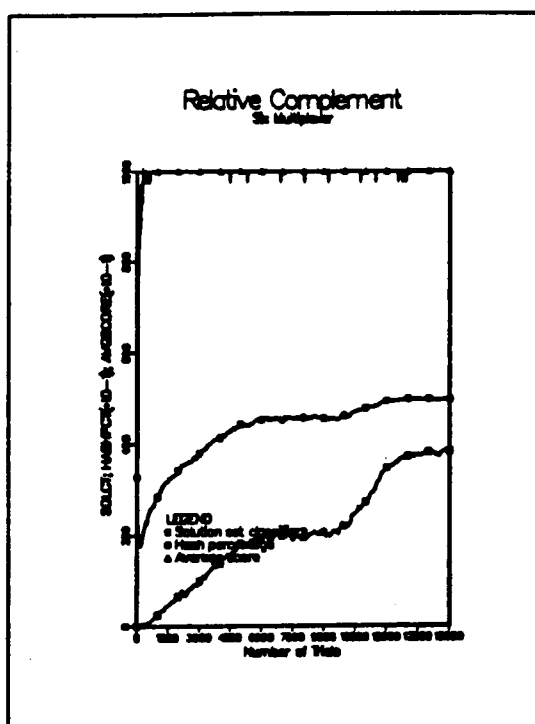
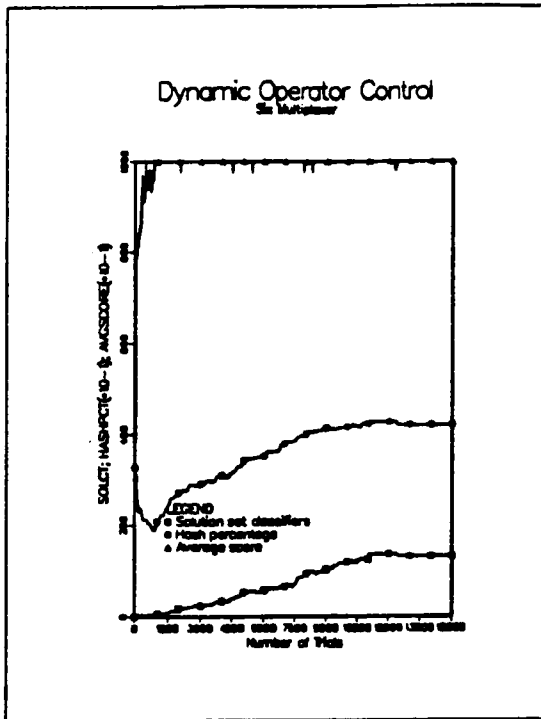
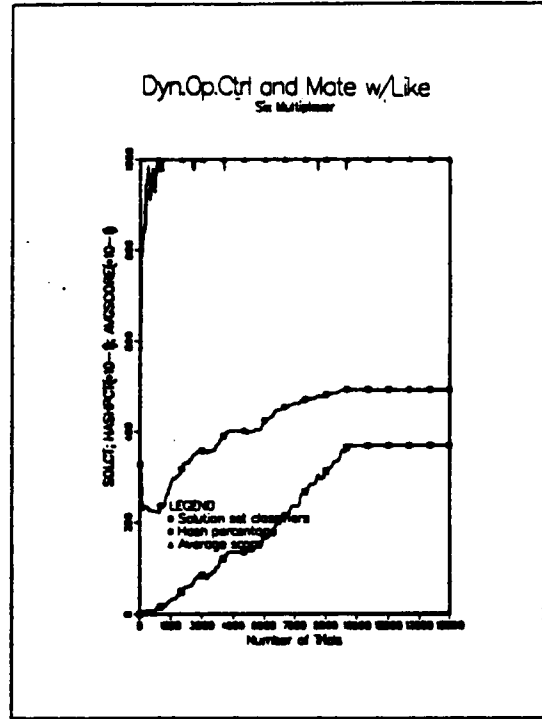


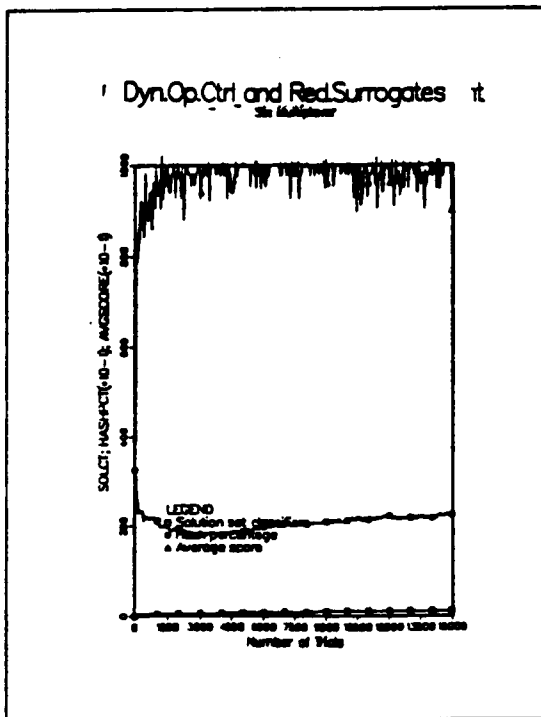
Figure A-7. Relative complement operator for the six multiplexer.



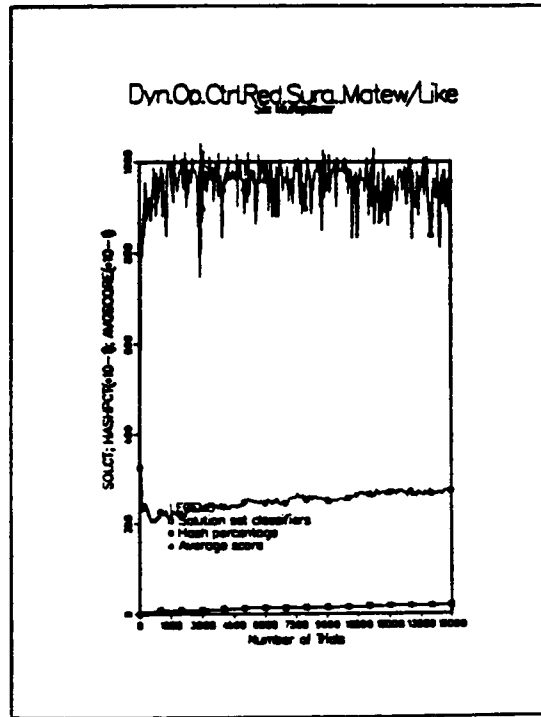
(a)



(b)

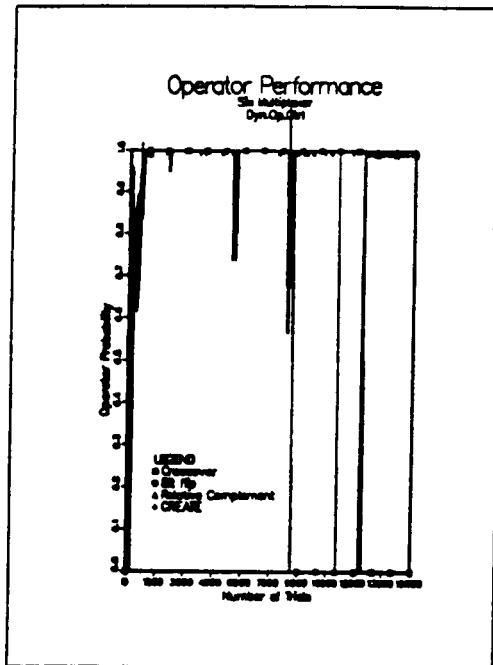


(c)

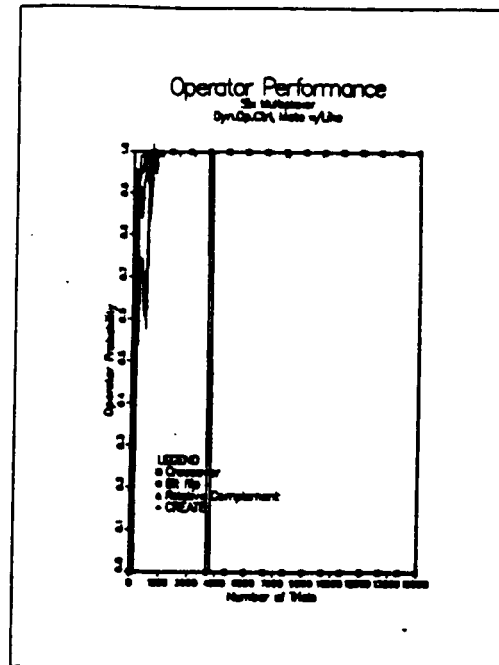


(d)

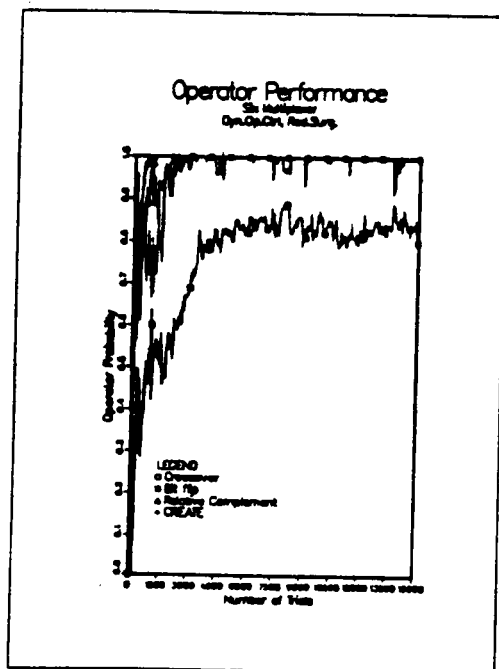
Figure A-8. Dynamic operator control for the six multiplexer.



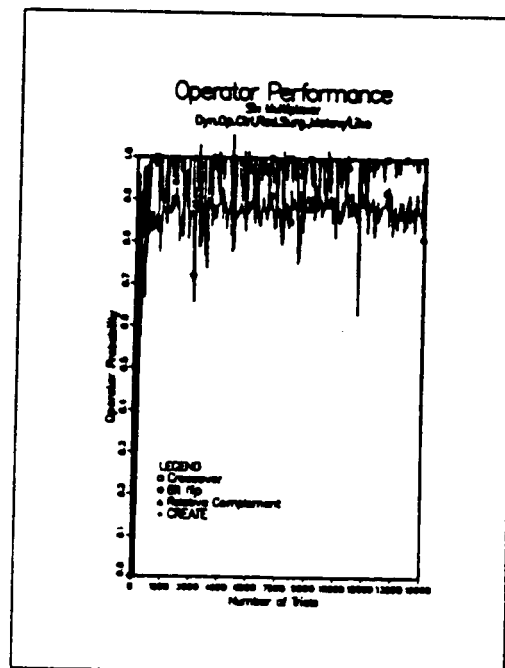
(a)



(b)

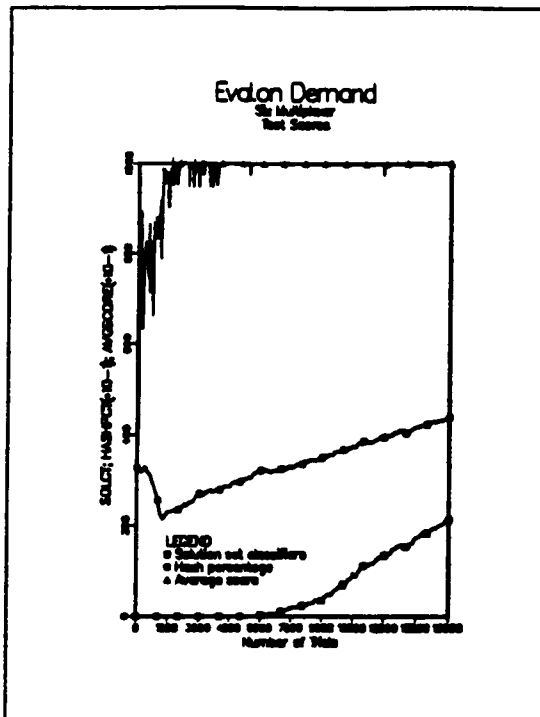


(c)

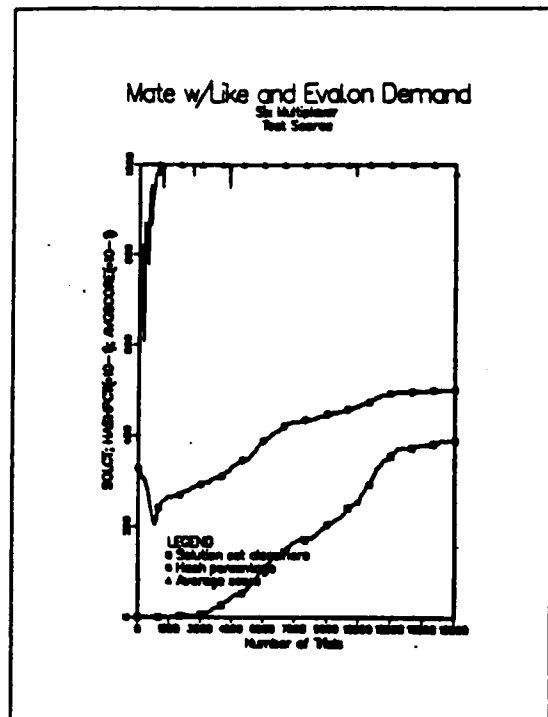


(d)

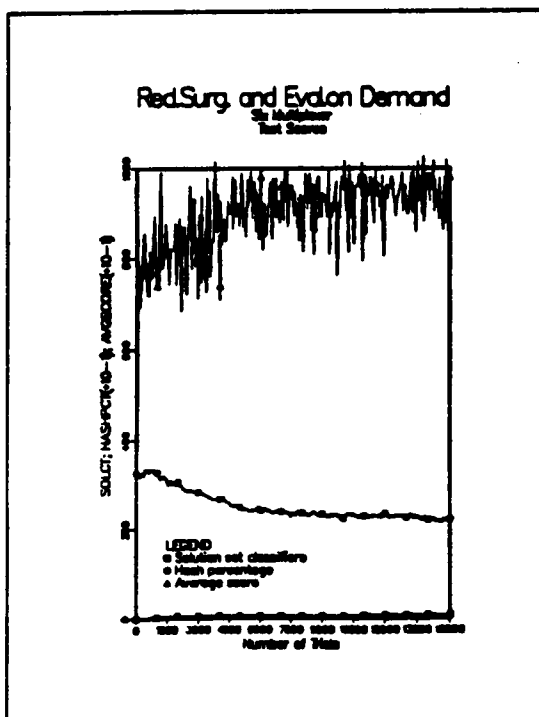
Figure A-9. Dynamic operator control operator performance curves for the six multiplexer.



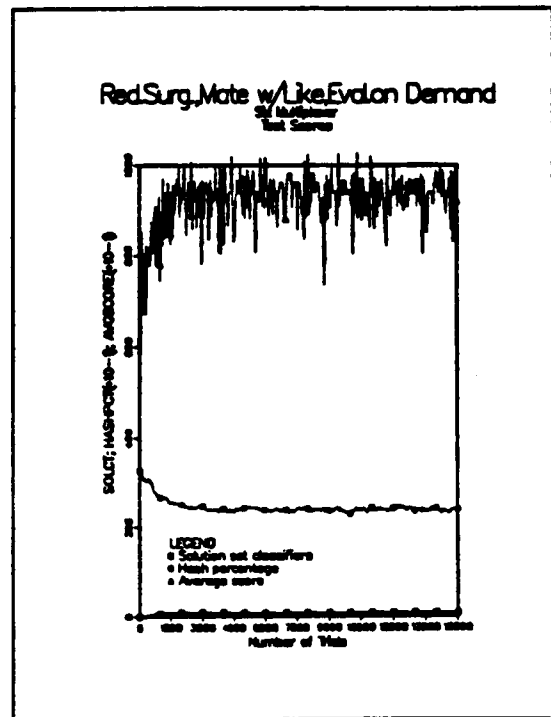
(a)



(b)



(c)



(d)

Figure A-10. Evaluation on demand for the six multiplexer.

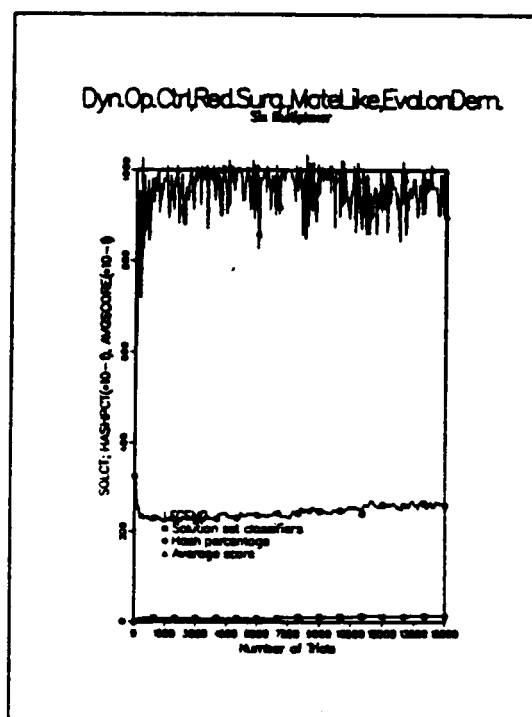
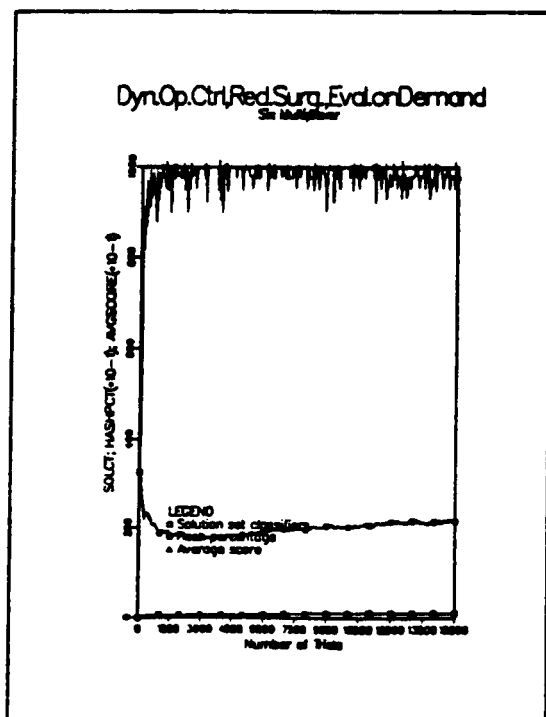
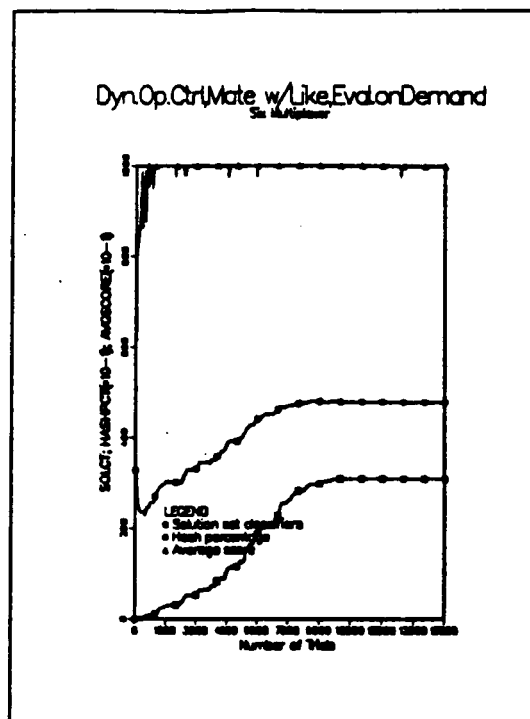
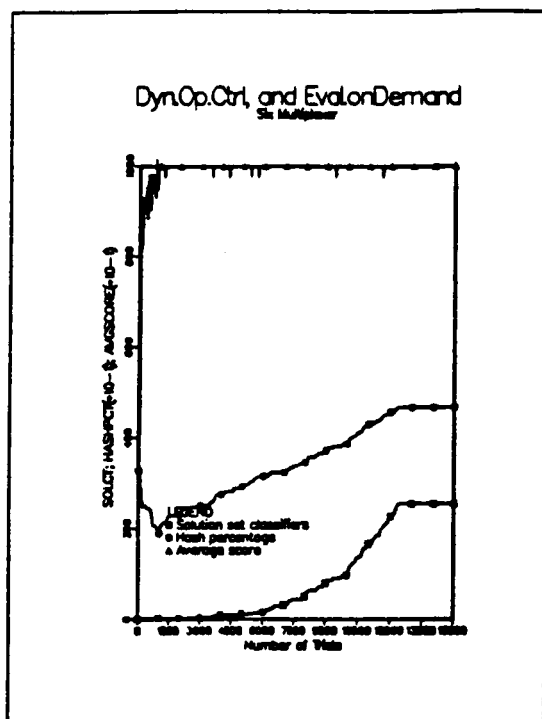
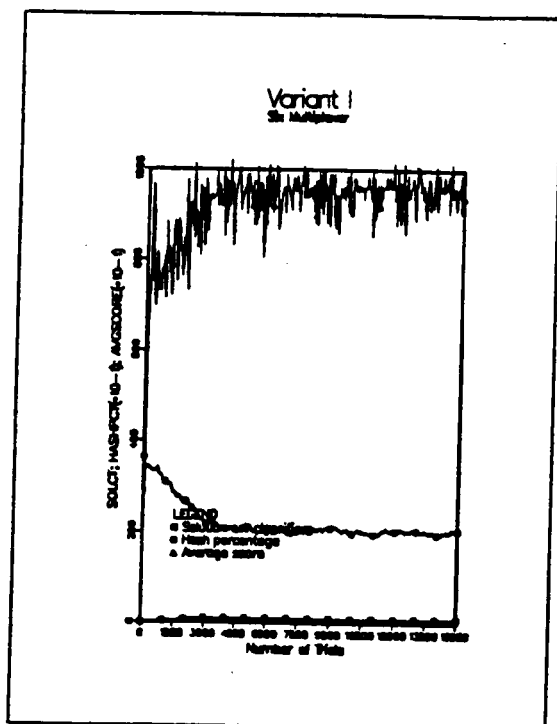
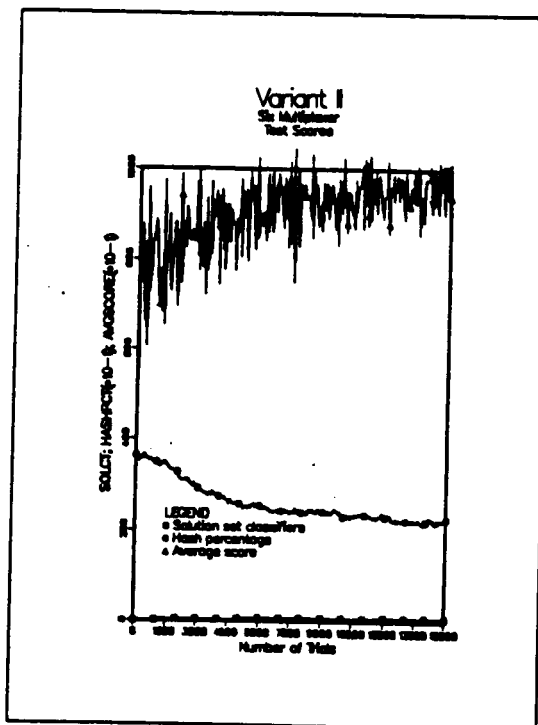


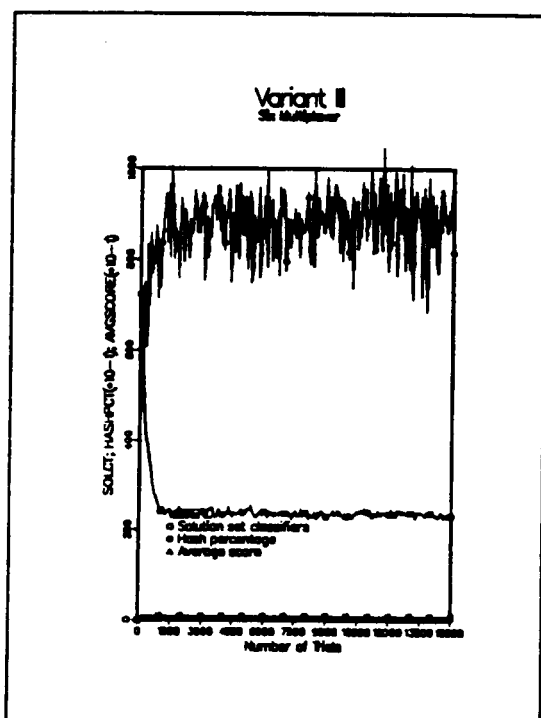
Figure A-11. Dynamic operator control and evaluation on demand for the six multiplexer.



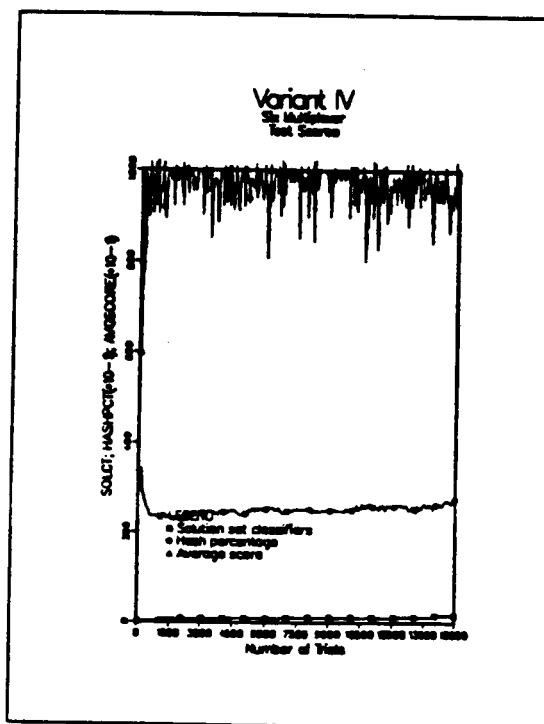
(a)



(b)



(c)



(d)

Figure A-12. Variant I, II, III, and IV for the six multiplexer.

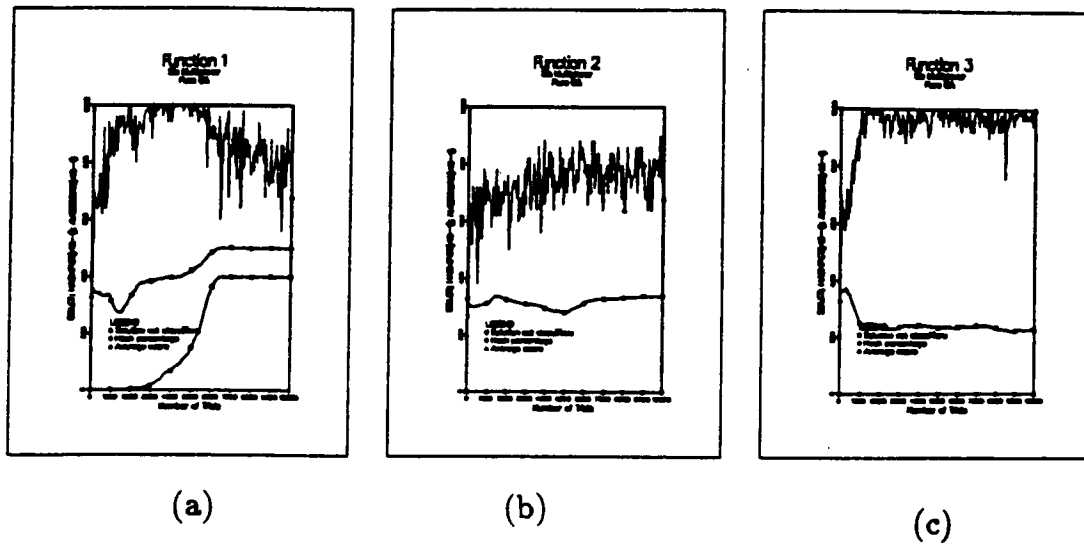


Figure A-13. Pure-GA static evaluation functions for the six multiplexer.

Hierarchical rule-ordering for F1				Hierarchical rule-ordering for F2				Hierarchical rule-ordering for F3			
	Integer representation	Classifier representation	State evaluation		Integer representation	Classifier representation	State evaluation		Integer representation	Classifier representation	State evaluation
100% correct	10	0 00 0	1.00	100% correct	10	0 00 0	4.00	100% correct	10	0 00 0	1.00
	3	0 00 0	2.00		3	0 00 0	4.00		3	0 00 0	1.00
75% correct	10	1 00 0	2.00	75% correct	10	1 00 0	4.00	75% correct	10	1 00 0	1.00
	20	0 00 0	1.75		20	0 00 0	4.00		20	0 00 0	0.50
50% correct	30	0 00 0	1.75	50% correct	30	0 00 0	4.00	50% correct	30	0 00 0	0.50
	0	0 00 0	1.61		0	0 00 0	2.00		0	0 00 0	0.50
	7	0 01 0	1.61		7	0 01 0	2.00		7	0 01 0	0.50
	0	0 00 0	2.13		0	0 00 0	2.00		0	0 00 0	0.50
	11	1 00 0	1.61		11	1 00 0	2.00		11	1 00 0	0.50
	10	1 10 0	1.61		10	1 10 0	2.00		10	1 10 0	0.50
	17	1 00 0	2.13		17	1 00 0	2.00		17	1 00 0	0.50
	20	0 00 0	0.71		20	0 00 0	2.00		20	0 00 0	0.50
	20	0 10 0	0.71		21	0 10 0	2.00		21	0 10 0	0.50
	20	0 00 0	2.13		20	0 00 0	2.00		20	0 00 0	0.50
100% without back	0	0 00 0	1.00	100% without back	0	0 00 0	2.00	100% without back	0	0 00 0	0.50
	1	0 00 0	1.00		1	0 00 0	2.00		1	0 00 0	0.50
	13	1 10 0	1.00		13	1 10 0	2.00		13	1 10 0	0.50
	0	1 00 0	1.00		0	1 00 0	2.00		0	1 00 0	0.50
20% correct	20	0 10 0	1.00	20% correct	20	0 10 0	0.50	20% correct	20	0 10 0	0.50
	20	0 00 0	1.00		20	0 01 0	0.50		20	0 01 0	0.50
0% correct	0	0 10 0	0.50	0% correct	0	0 10 0	1.00	0% correct	0	0 10 0	0.50
	0	0 11 0	0.50		0	0 11 0	1.00		0	0 11 0	0.50
	0	0 10 0	0.50		0	0 10 0	0.50		0	0 10 0	0.50
	10	1 00 0	0.50		10	1 00 0	1.00		10	1 00 0	0.50
	10	1 11 0	0.50		13	1 11 0	1.00		13	1 11 0	0.50
	10	1 00 0	0.50		10	1 01 0	0.50		10	1 01 0	0.50
	20	0 11 0	0.50		20	0 11 0	0.50		20	0 11 0	0.50

(a)

(b)

(c)

Figure A-14. Hierarchical rule-ordering for the three multiplexer.

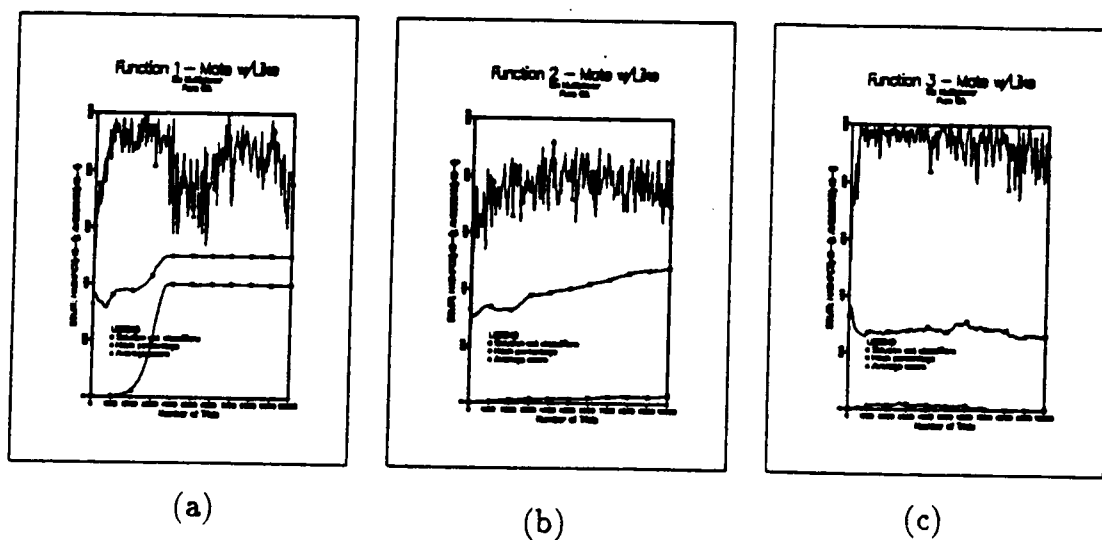


Figure A-15. Pure-GA using speciation for the six multiplexer.

Macrostate for F1 without speciation

Macrostate Number of Instances	"Concepts"		Total Strength
	(taxon)	(action)	
252	01#1#	/	1
100	000#	/	0
45	001#	/	1
1	10#1#	/	1
1	11#0#	/	0
1	10#0#	/	0
400			800.00

(a)

Macrostate for F1 with speciation

Macrostate Number of Instances	"Concepts"		Total Strength
	(taxon)	(action)	
209	001#	/	1
170	10#0#	/	0
15	01#0#	/	0
3	11#1#	/	1
1	1#11	/	1
1	000#	/	0
1	11#0#	/	0
400			799.73

(b)

Figure A-16. Pure-GA macrostates for the six multiplexer.

APPENDIX B

ELEVEN MULTIPLEXER DATA SETS

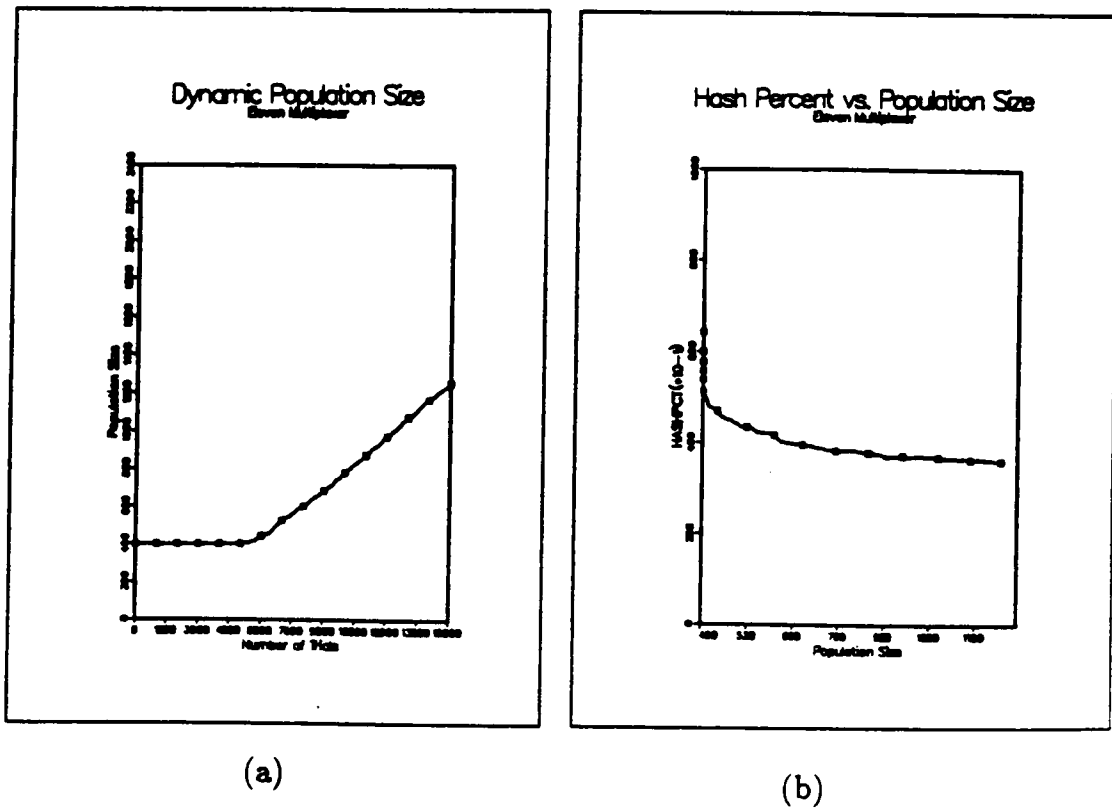
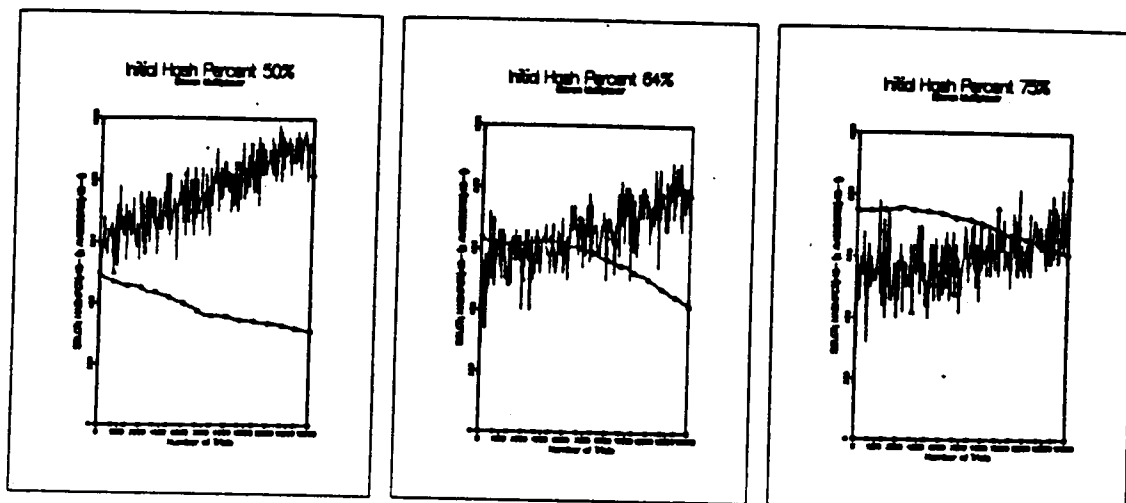


Figure B-1. Dynamic population size and hash percentage for the eleven multiplexer.



(a)

(b)

(c)

Figure B-2. Controlled initial hash percentage for the eleven multiplexer.

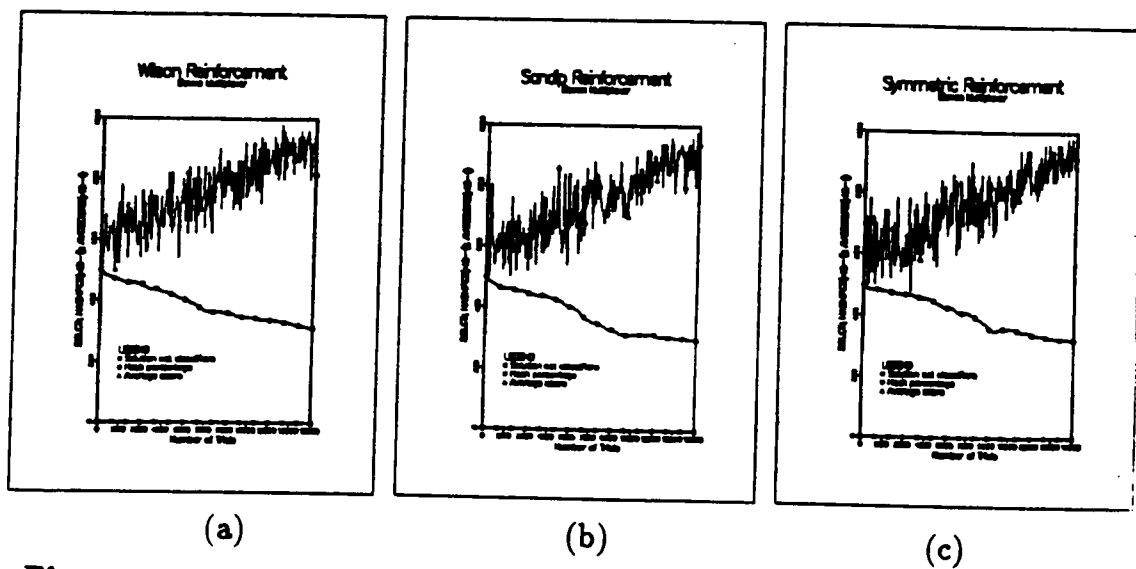


Figure B-3. Wilson, Sen, and symmetric reinforcement for the eleven multiplexer.

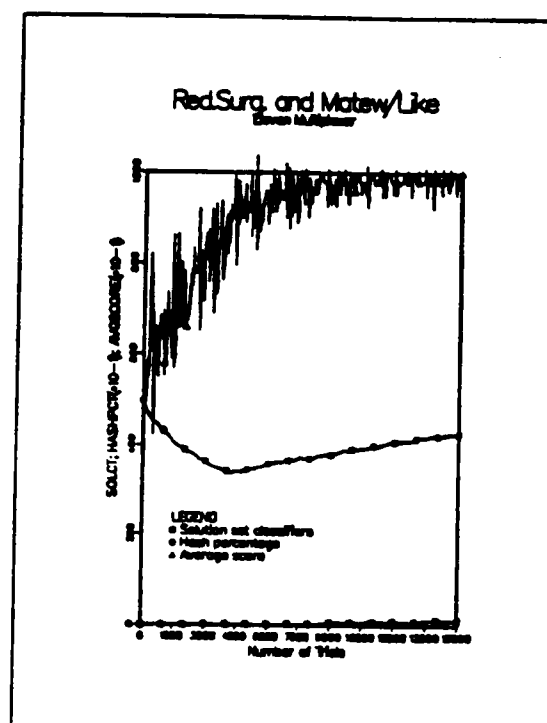
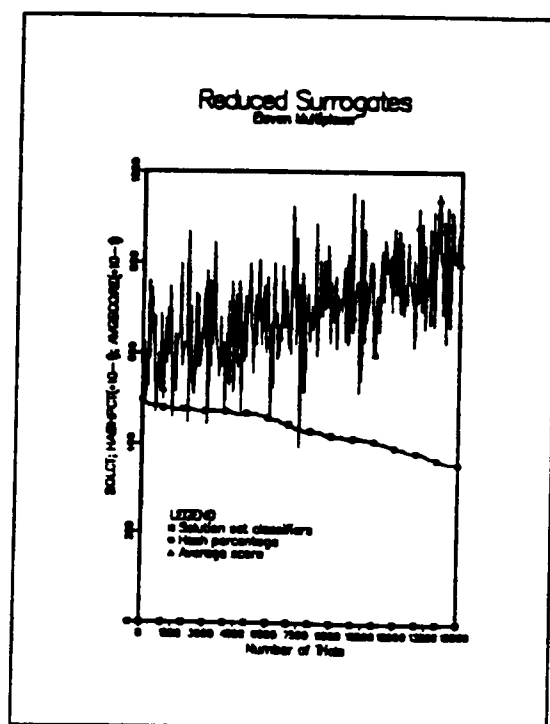
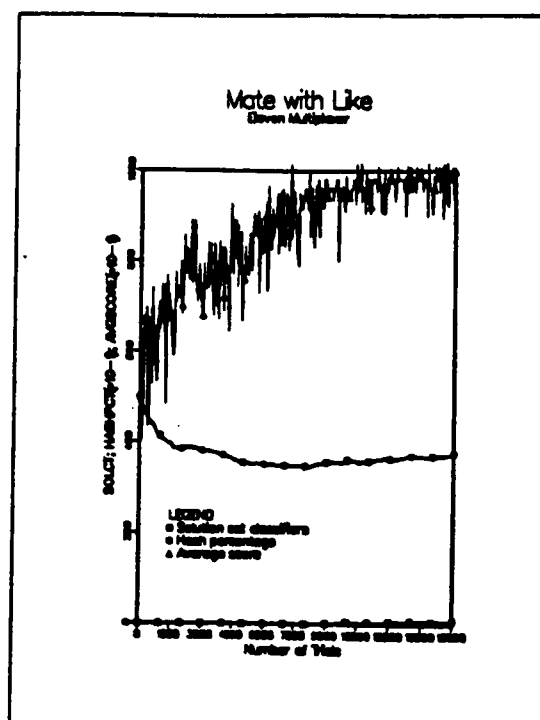
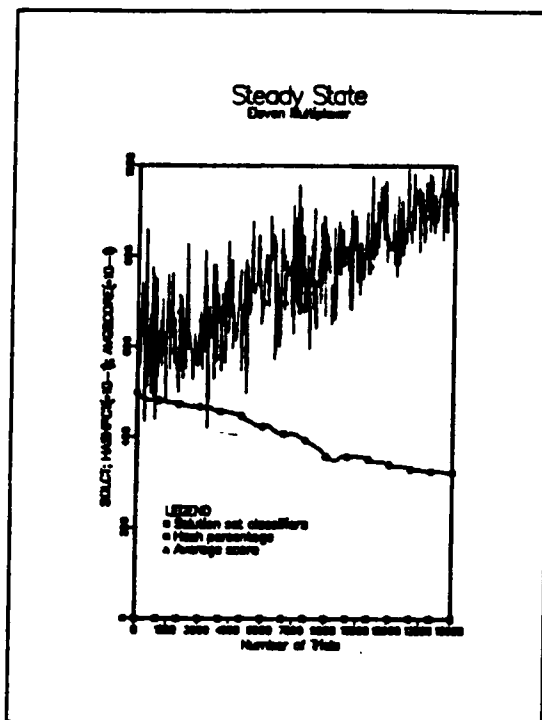


Figure B-4. Genetic algorithms for the eleven multiplexer.

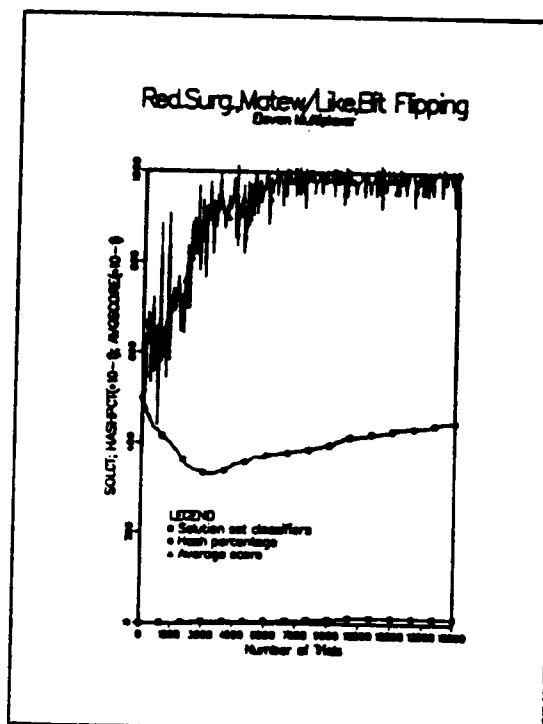
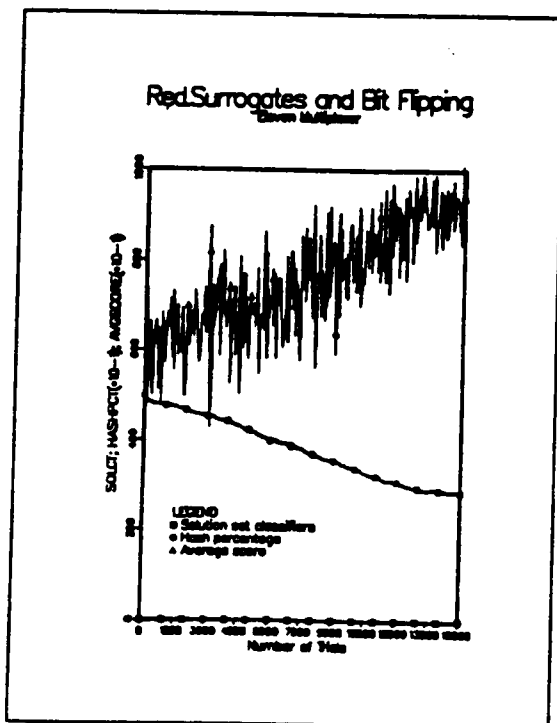
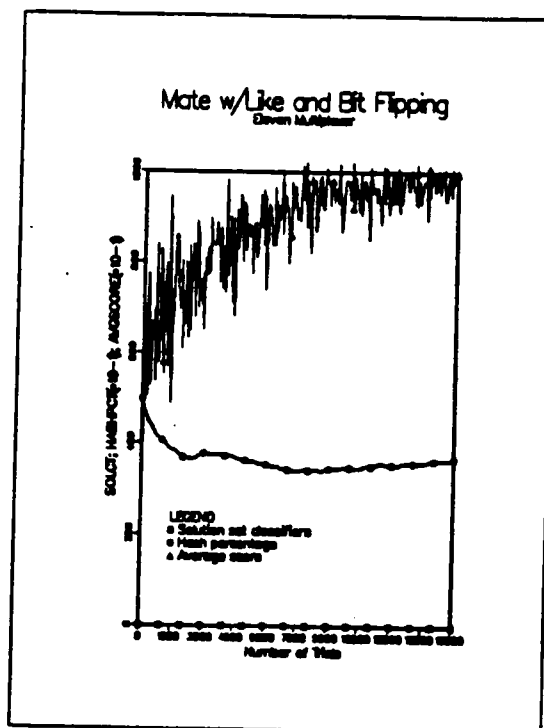
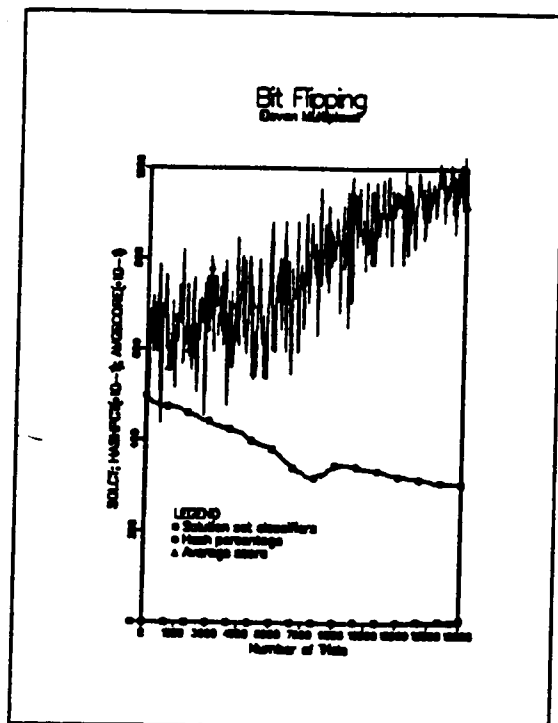


Figure B-5. Bit flipping for the eleven multiplexer.

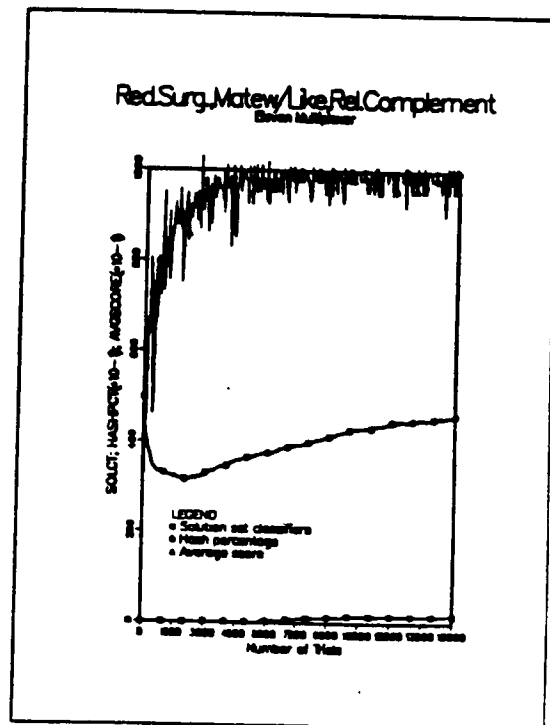
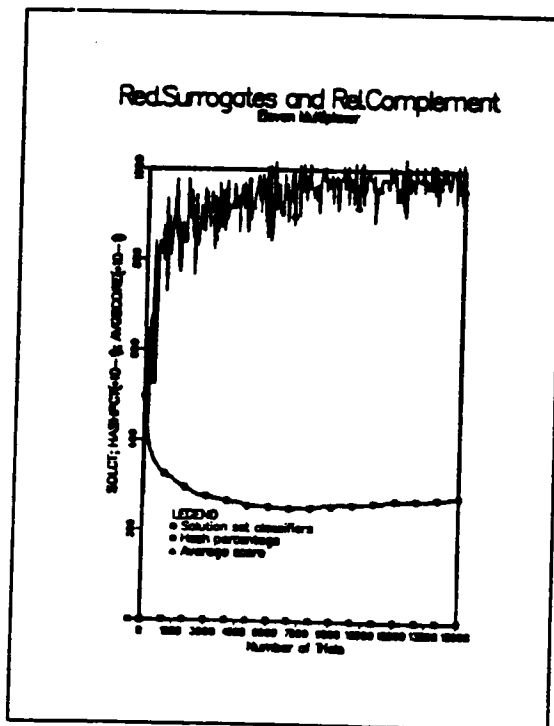
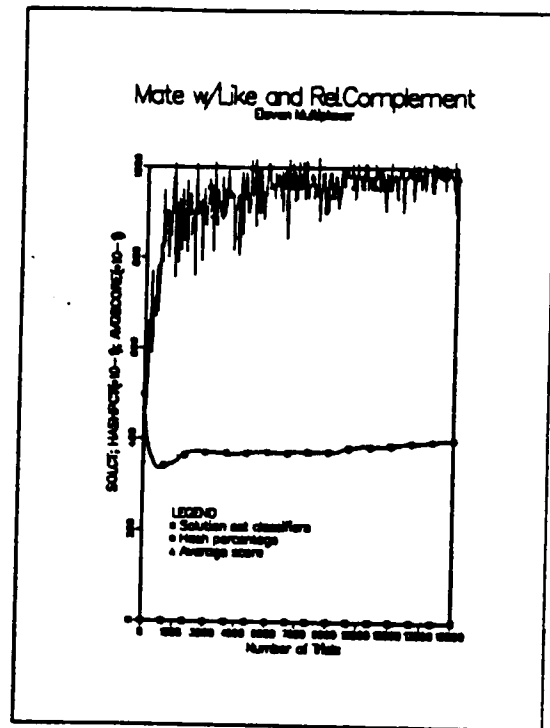
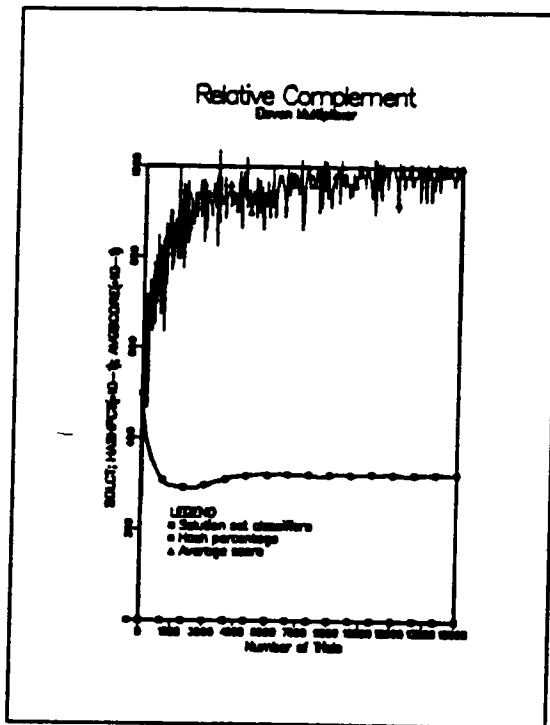
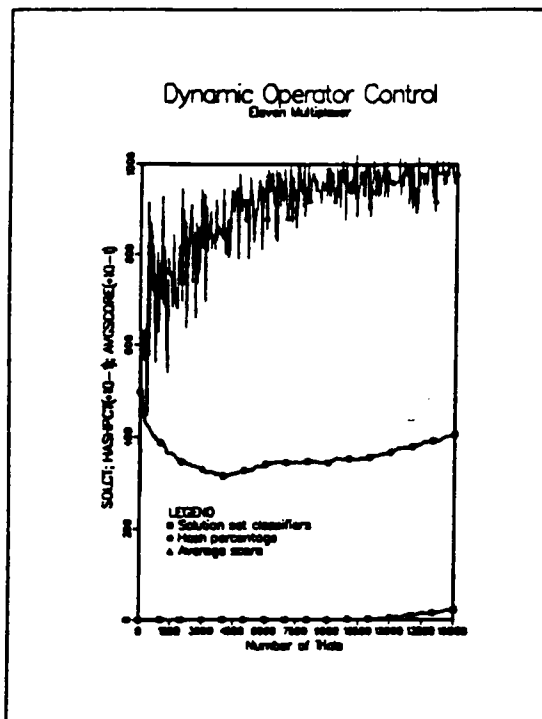
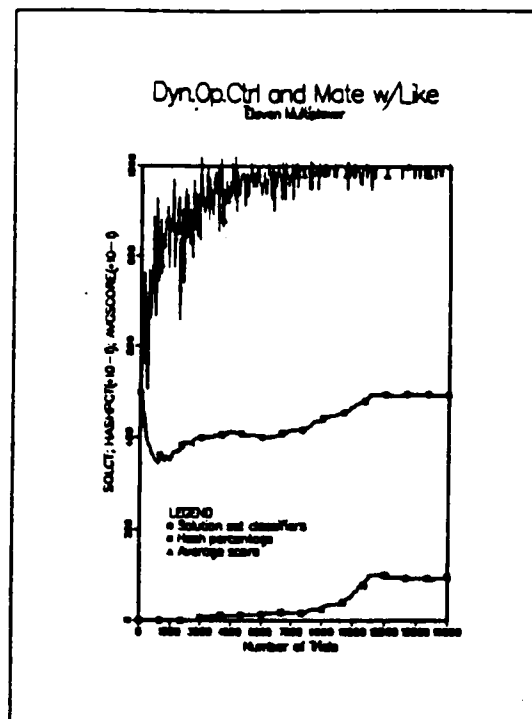


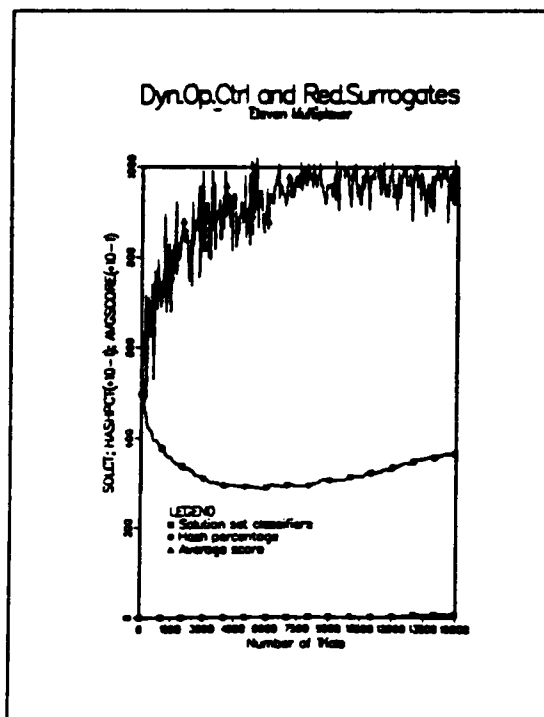
Figure B-6. Relative complement for the eleven multiplexer.



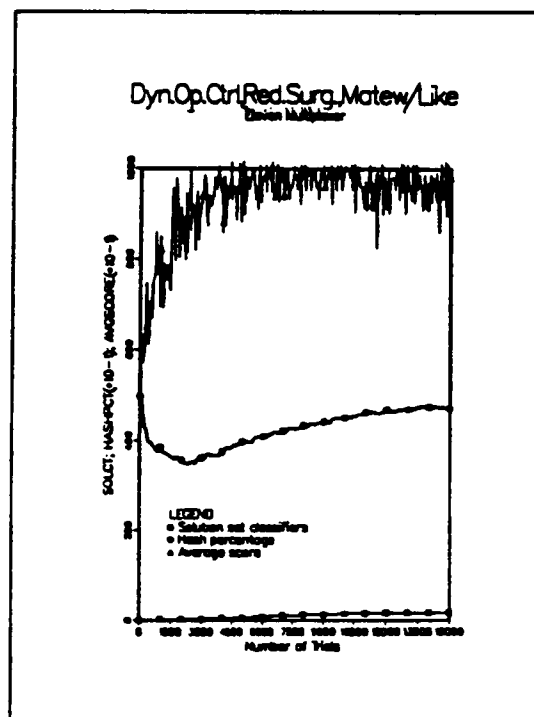
(a)



(b)

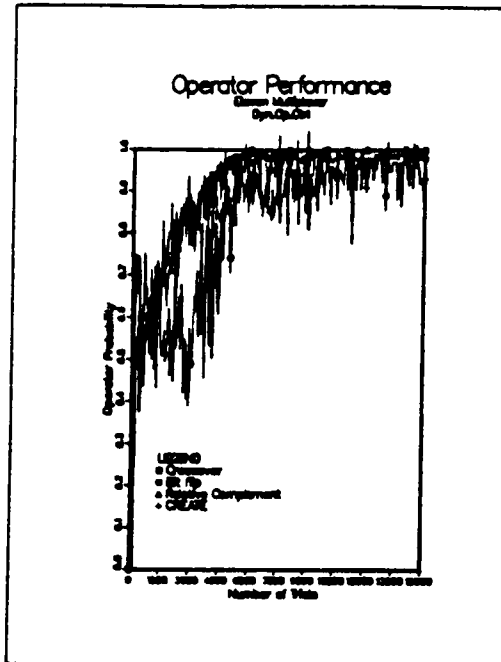


(c)

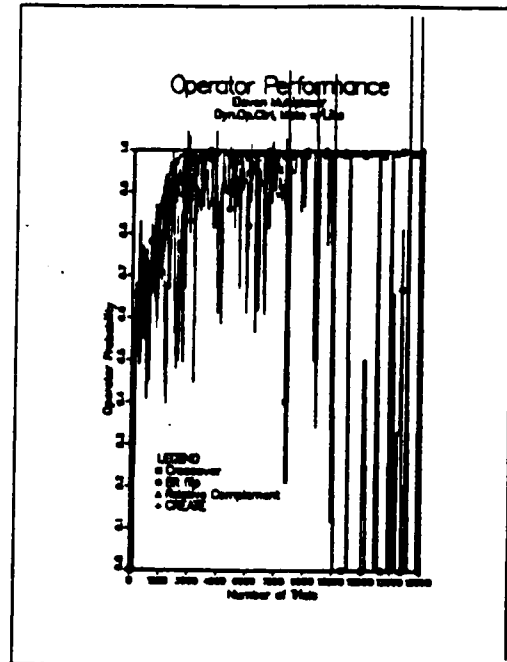


(d)

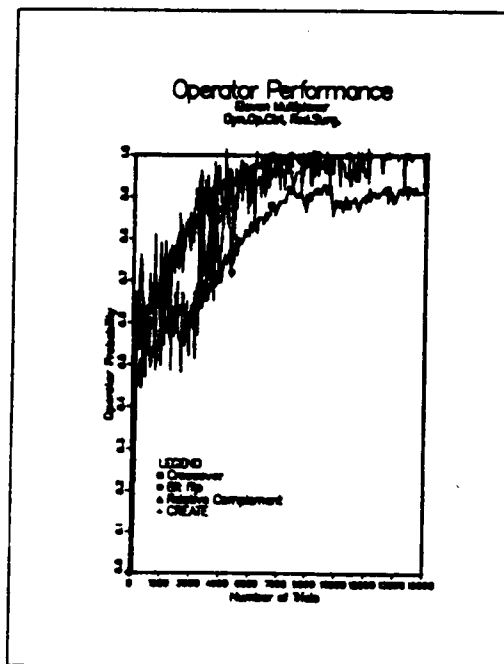
Figure B-7. Dynamic operator control for the eleven multiplexer.



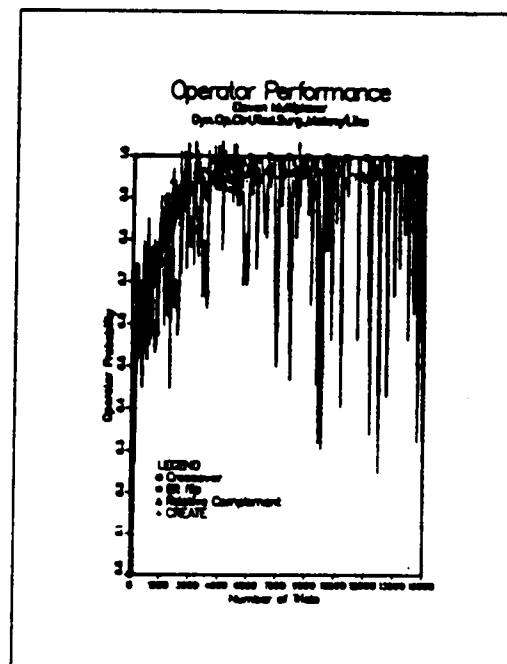
(a)



(b)

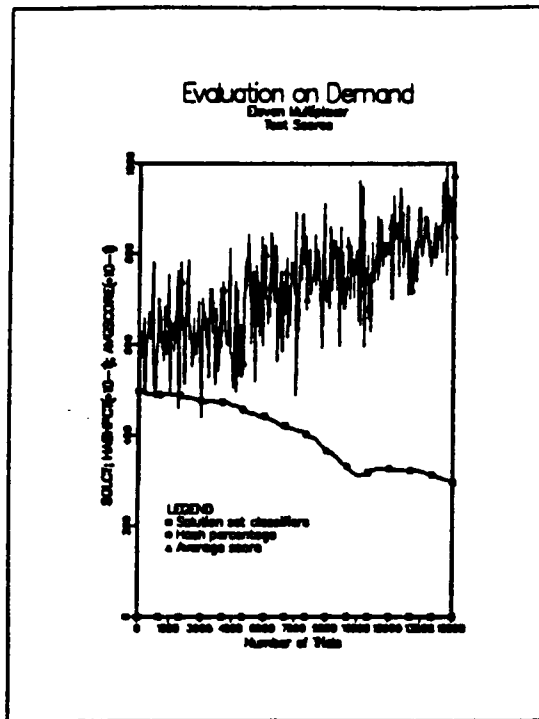


(c)

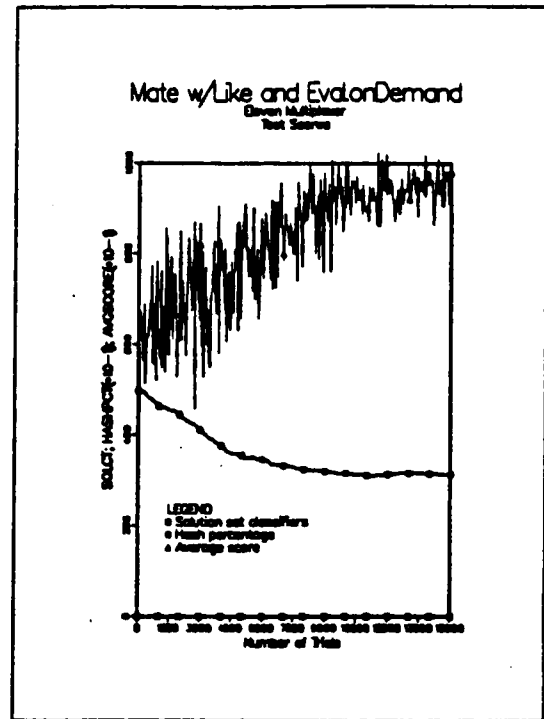


(d)

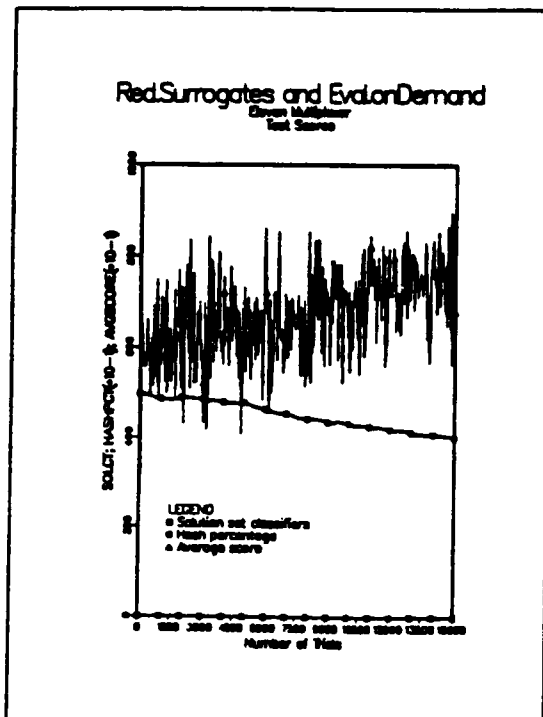
Figure B-8. Dynamic operator control operator performance curves for the eleven multiplexer.



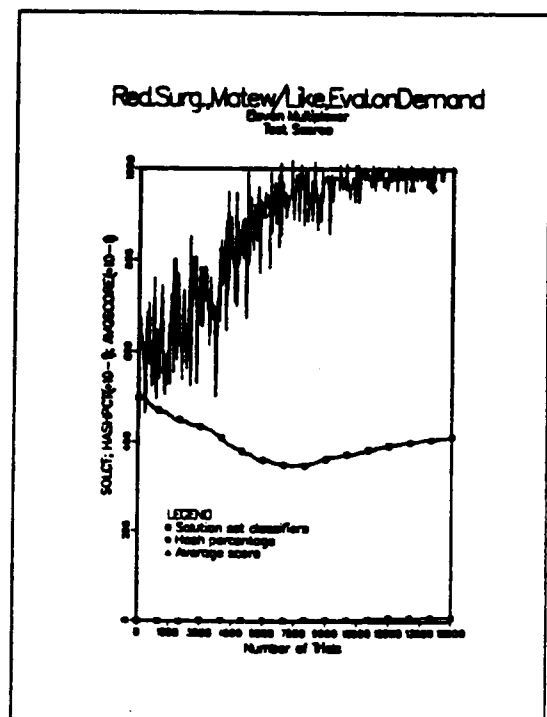
(a)



(b)



(c)



(d)

Figure B-9. Evaluation on demand for the eleven multiplexer.

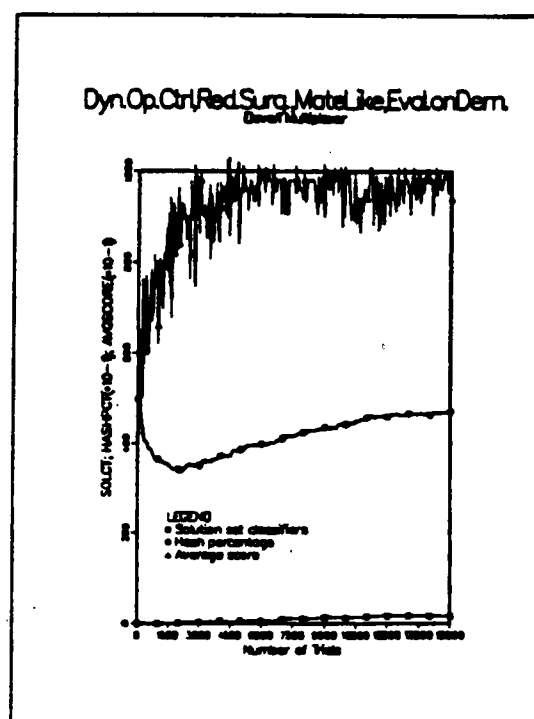
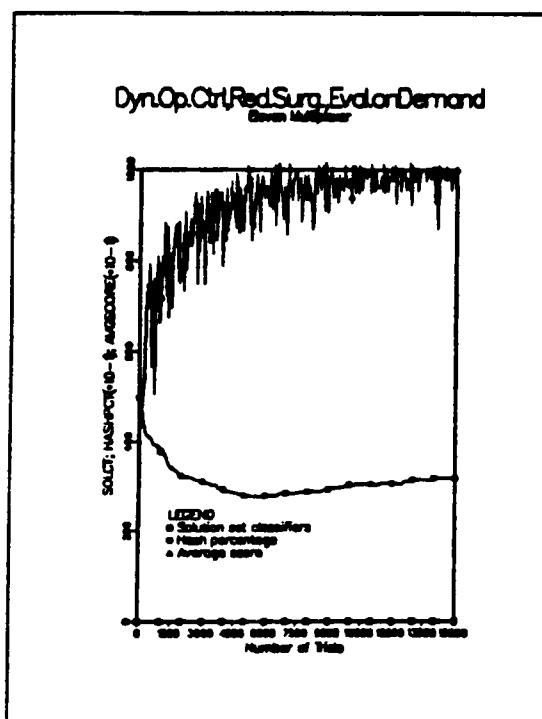
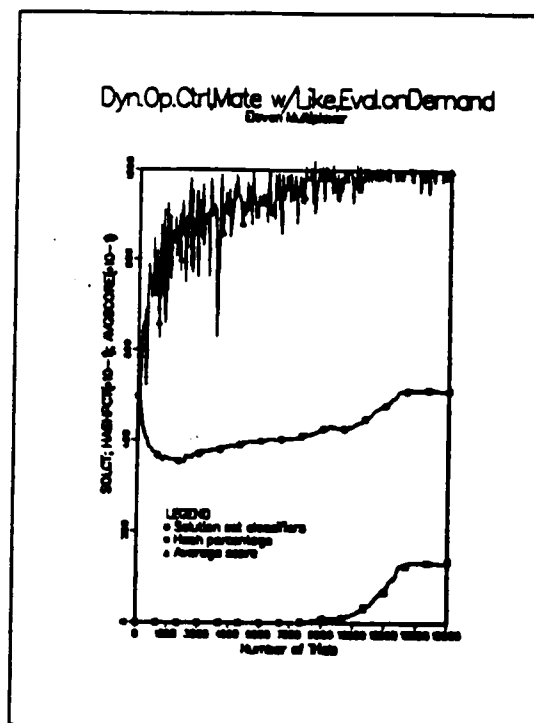
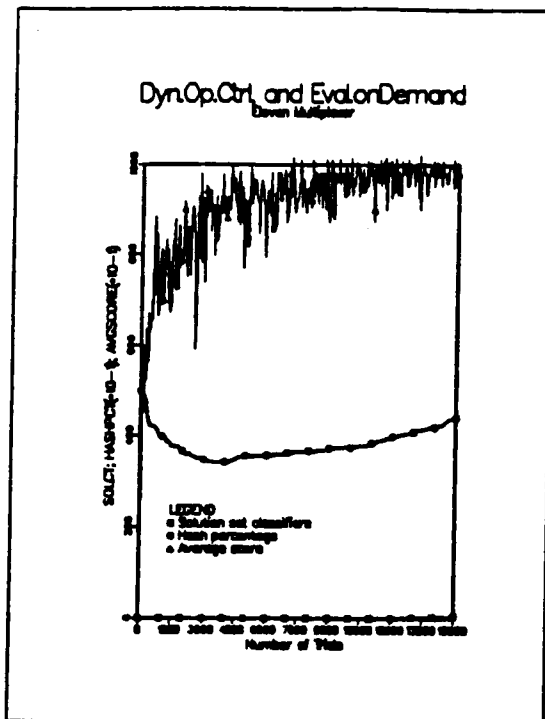
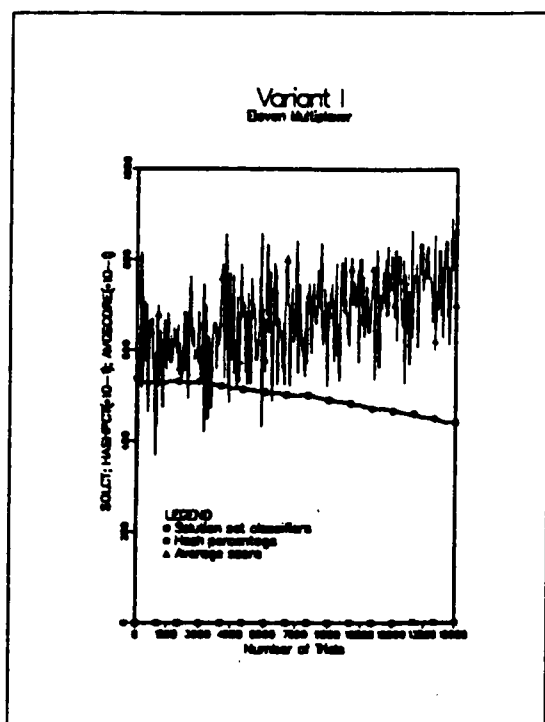
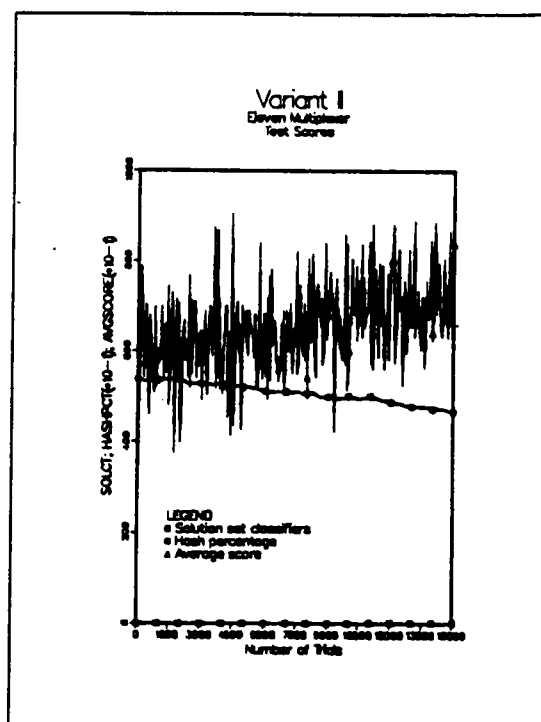


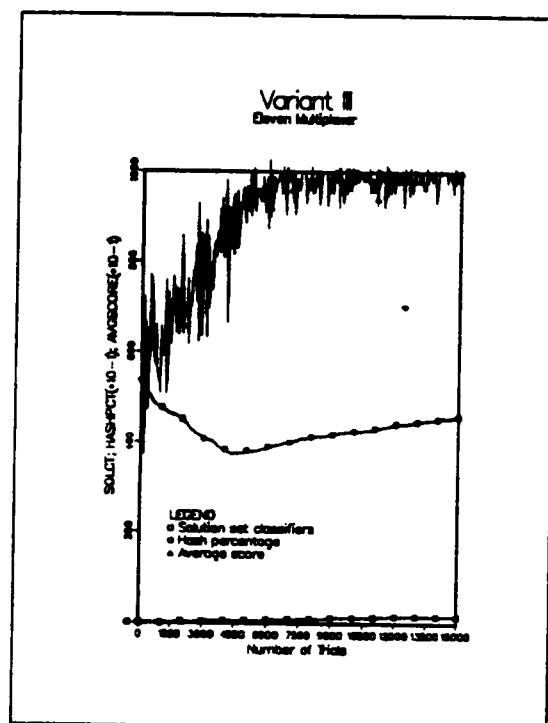
Figure B-10. Dynamic operator control and evaluation on demand for the eleven multiplexer.



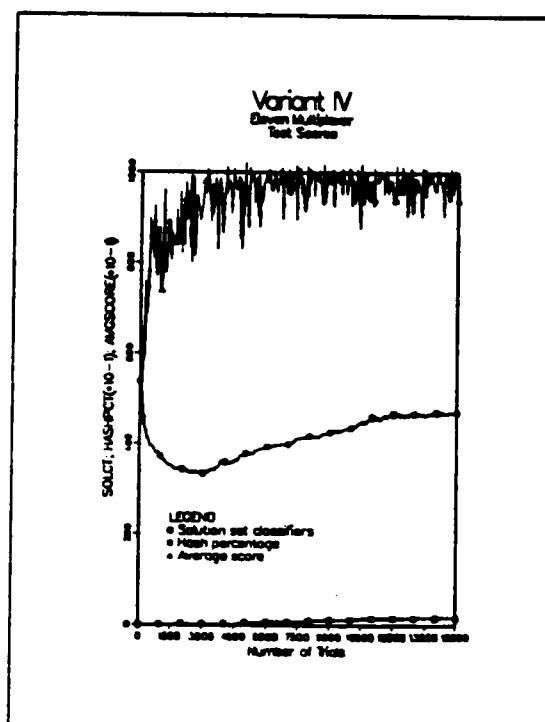
(a)



(b)



(c)



(d)

Figure B-11. Variant I, II, III, and IV for the eleven multiplexer.

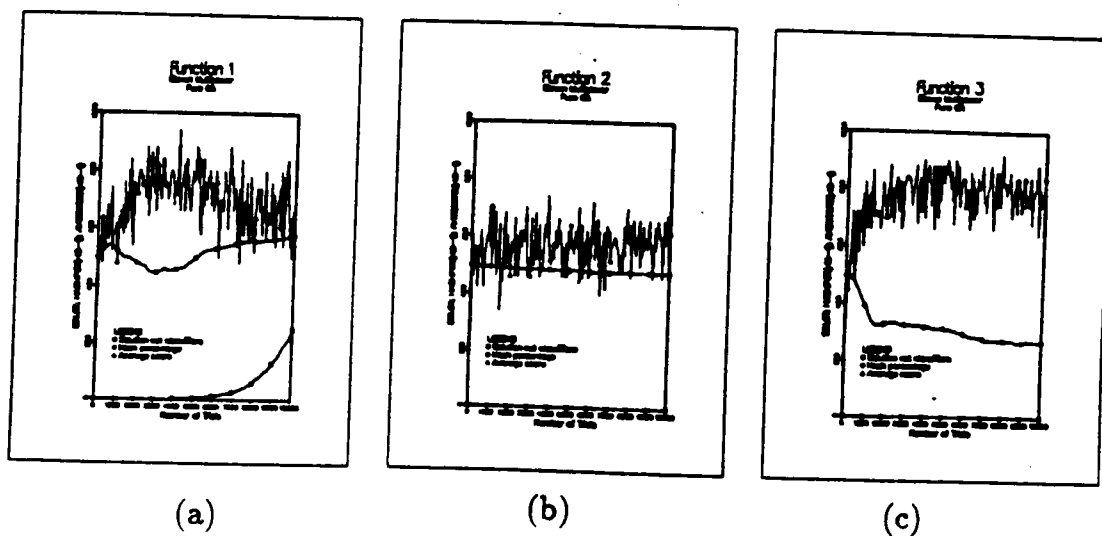


Figure B-12. Pure-GA static evaluation functions for the eleven multiplexer.

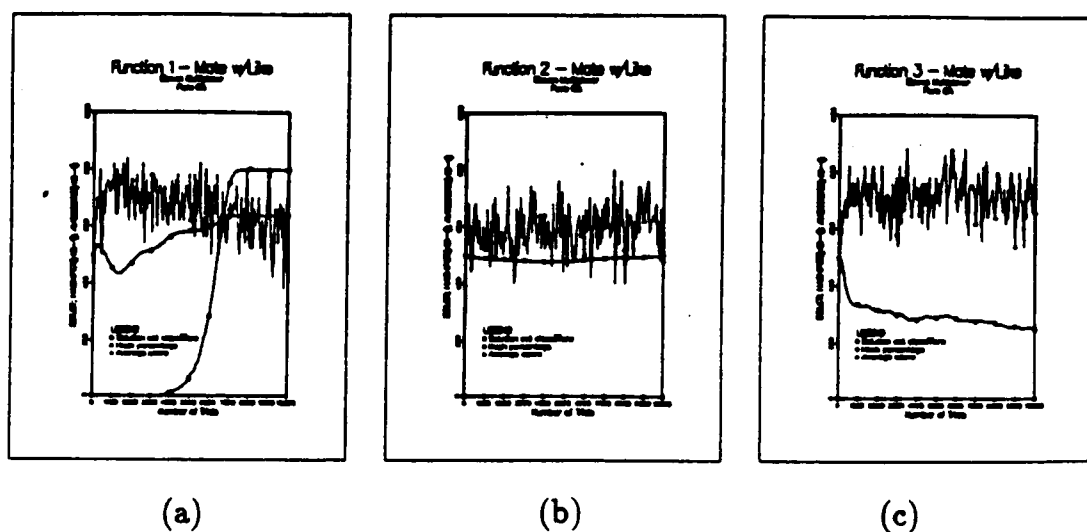


Figure B-13. Pure-GA using speciation for the eleven multiplexer.

APPENDIX C

TWENTY MULTIPLEXER DATA SETS

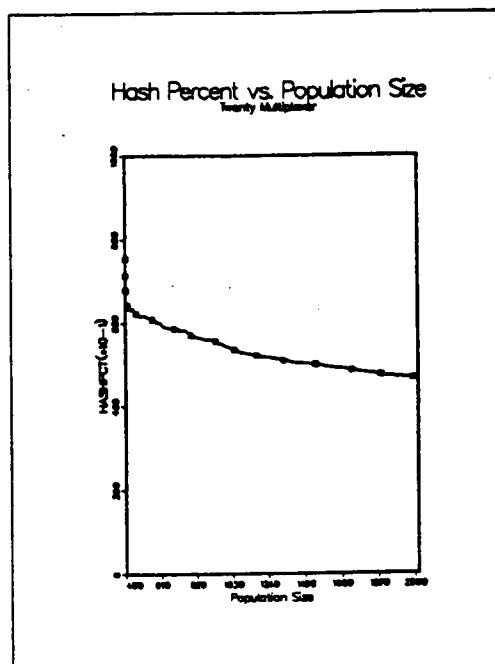
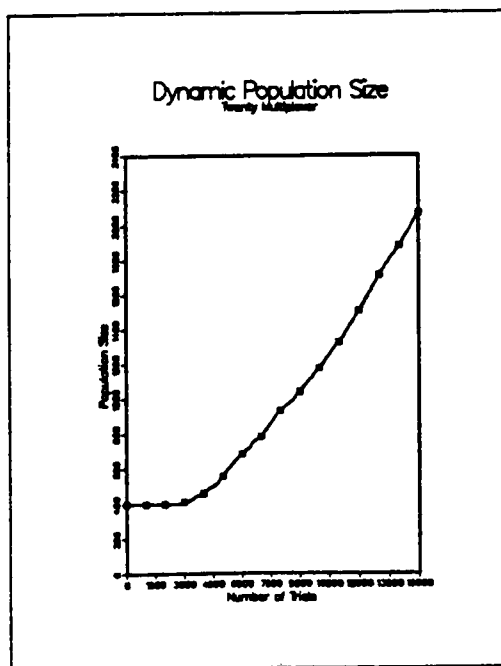


Figure C-1. Dynamic population size and hash percentage for the twenty multiplexer.

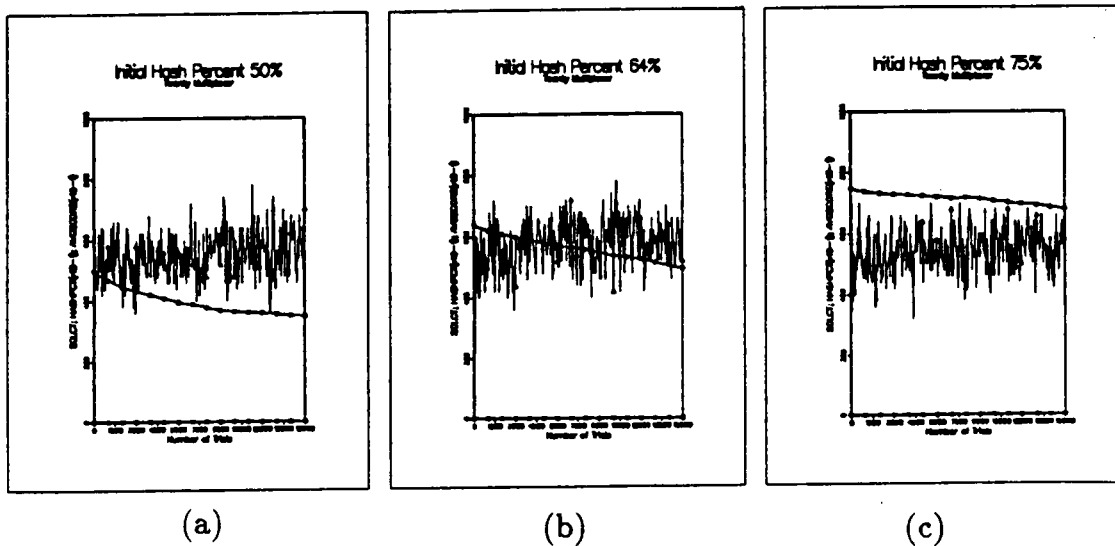


Figure C-2. Controlled initial hash percentage for the twenty multiplexer.

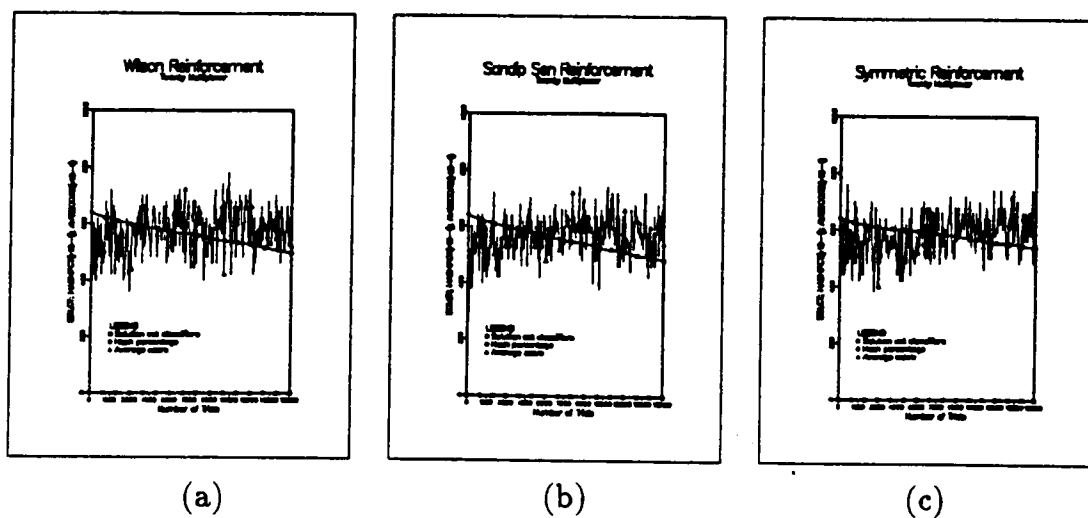
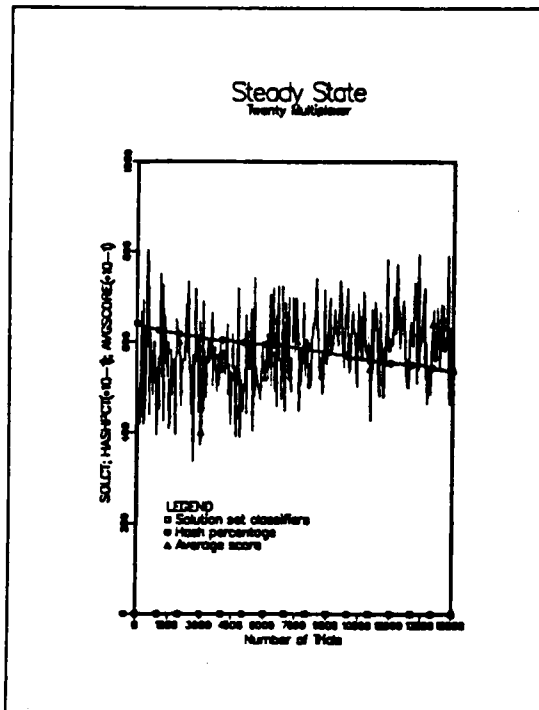
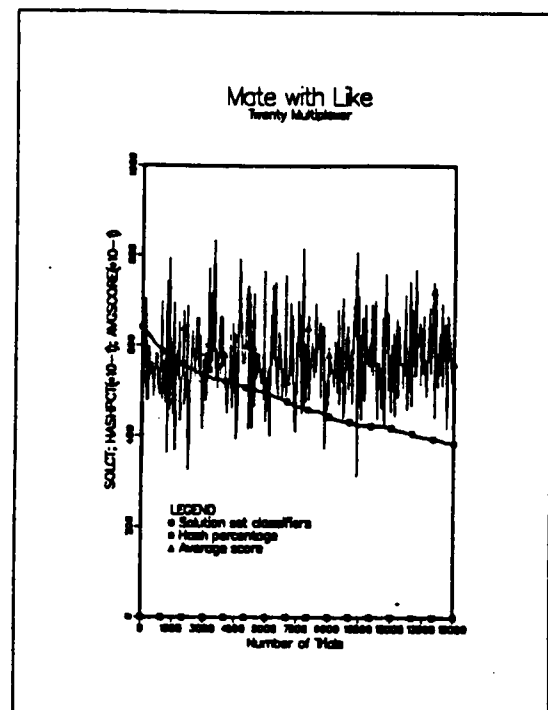


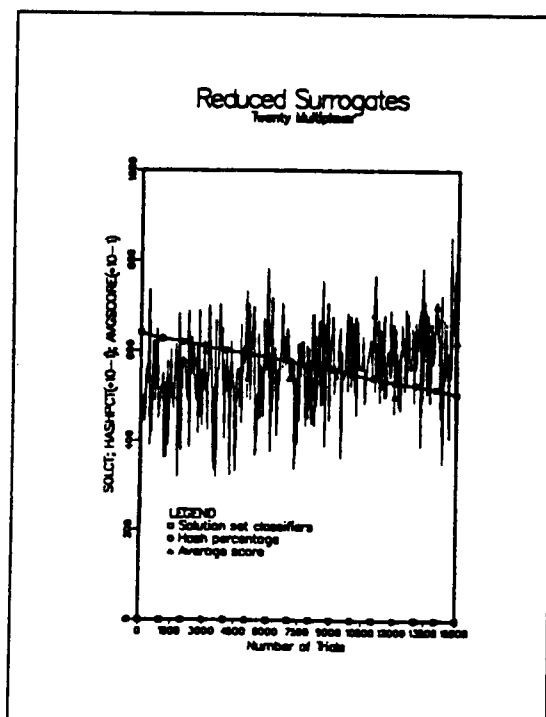
Figure C-3. Wilson, Sen, and symmetric reinforcement for the twenty multiplexer.



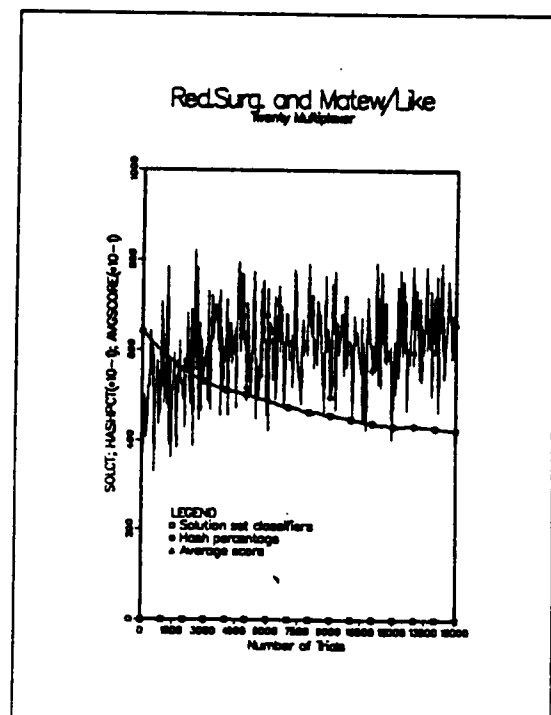
(a)



(b)



(c)



(d)

Figure C-4. Genetic algorithms for the twenty multiplexer.

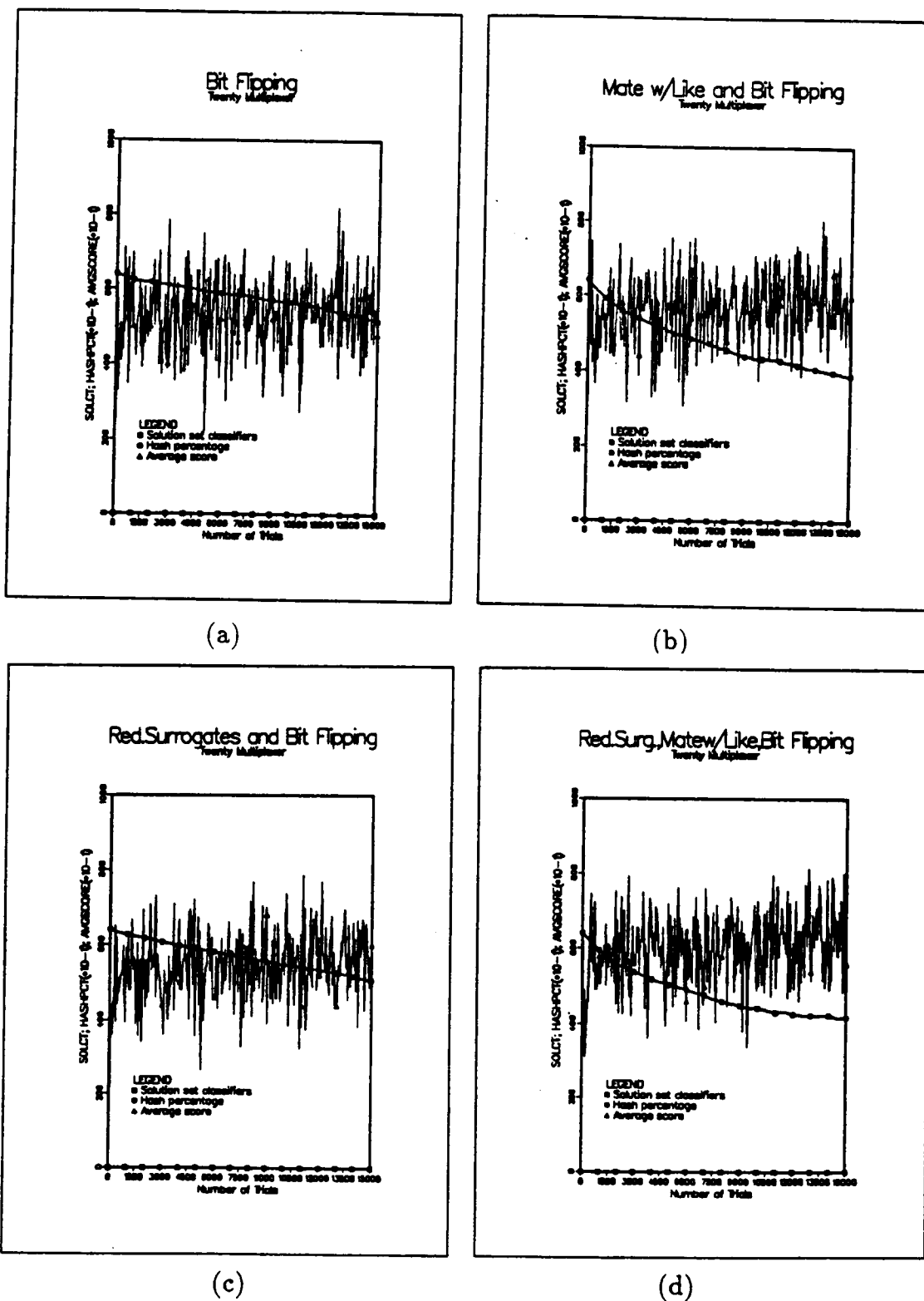
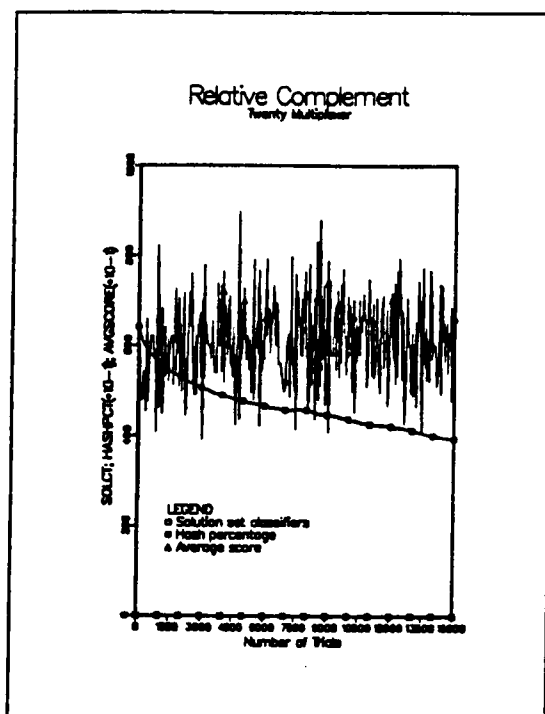
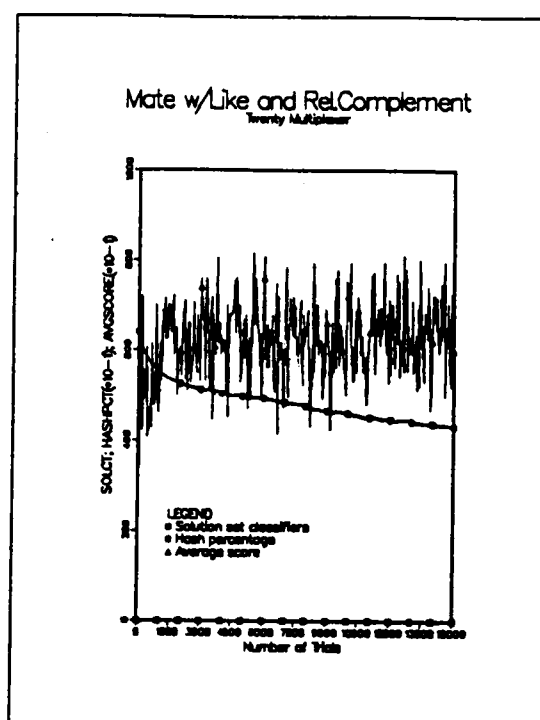


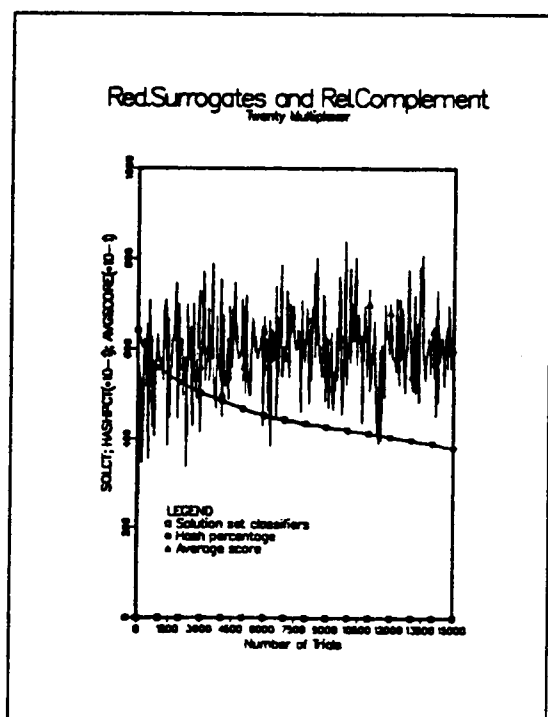
Figure C-5. Bit flipping operator for the twenty multiplexer.



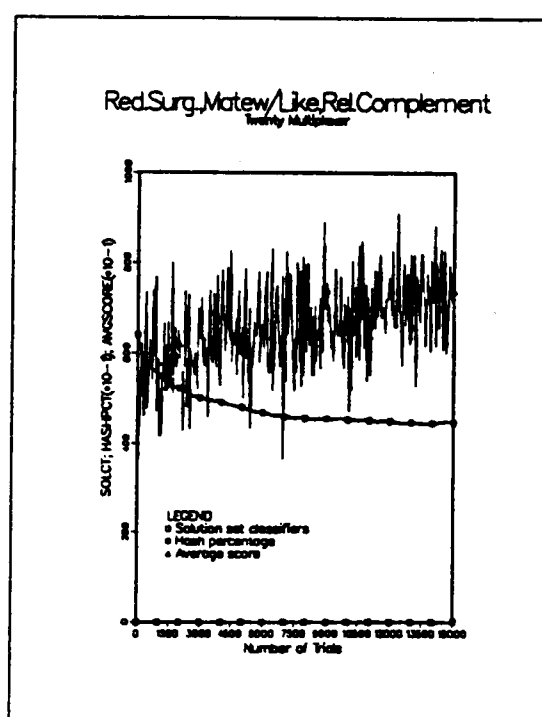
(a)



(b)

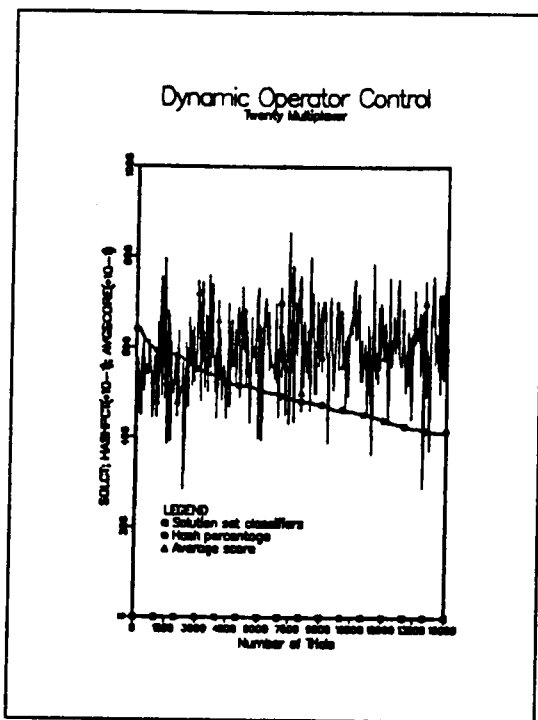


(c)

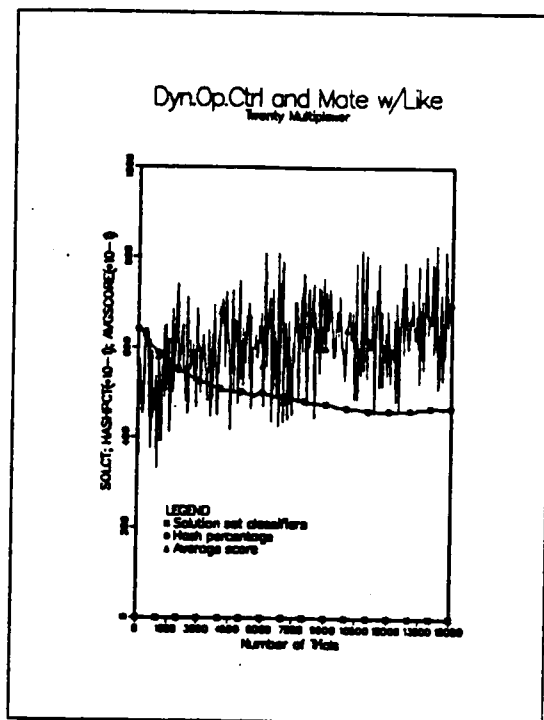


(d)

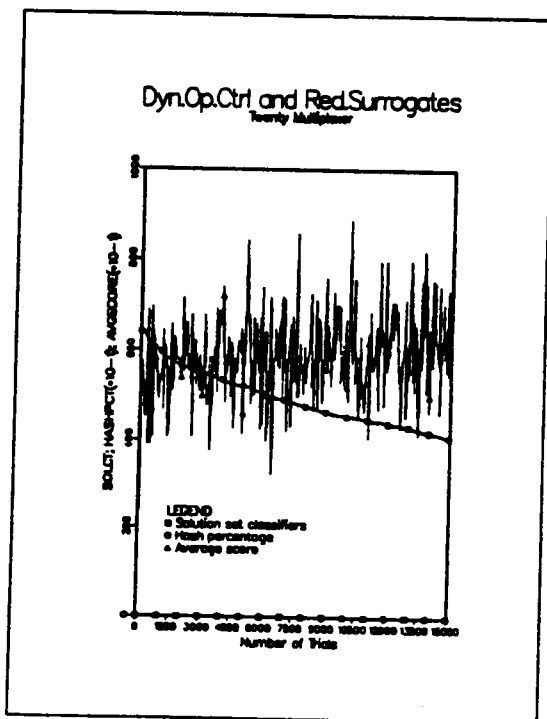
Figure C-6. Relative complement operator for the twenty multiplexer.



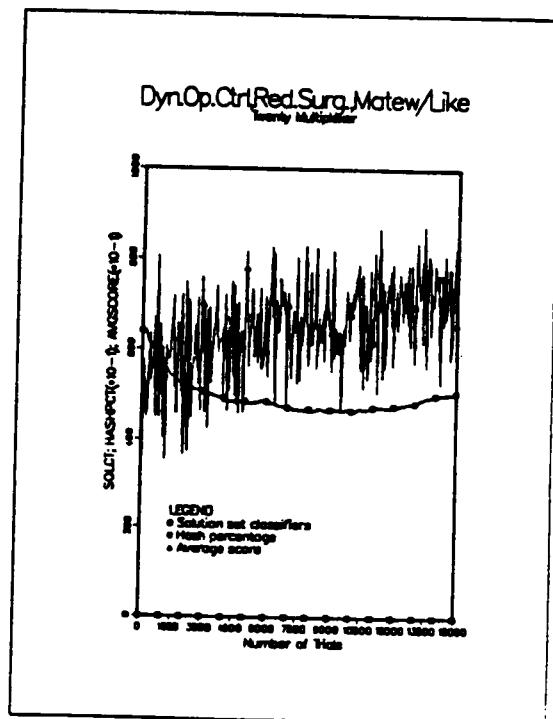
(a)



(b)

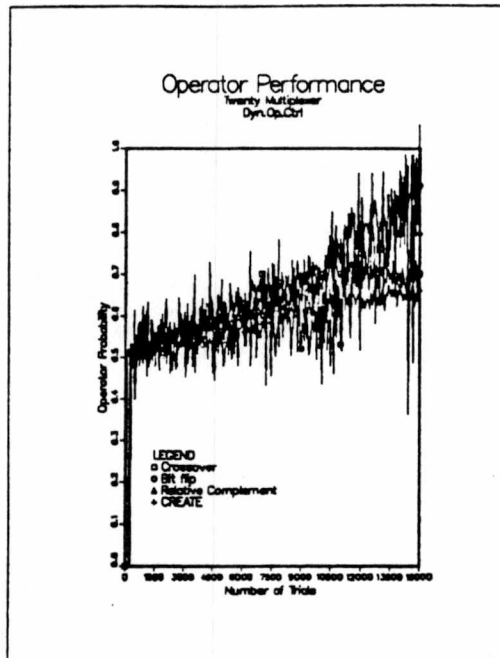


(c)

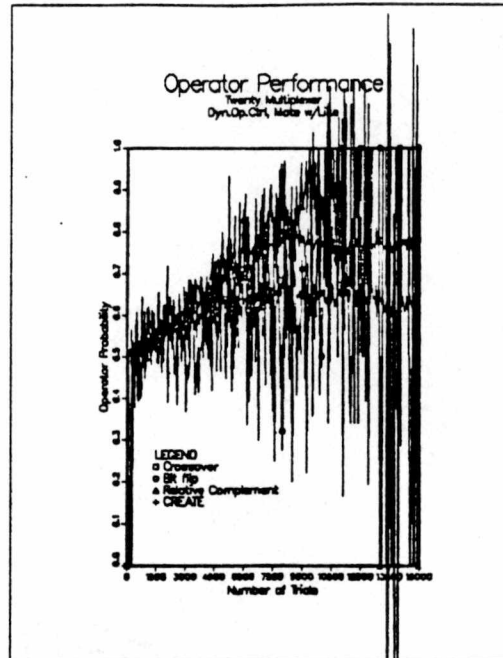


(d)

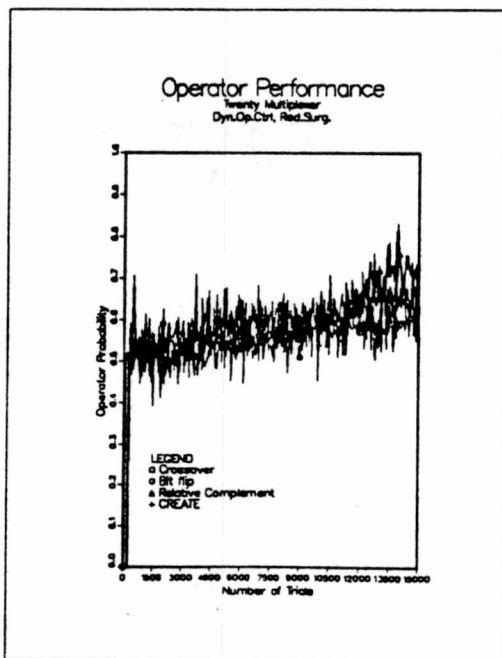
Figure C-7. Dynamic operator control for the twenty multiplexer.



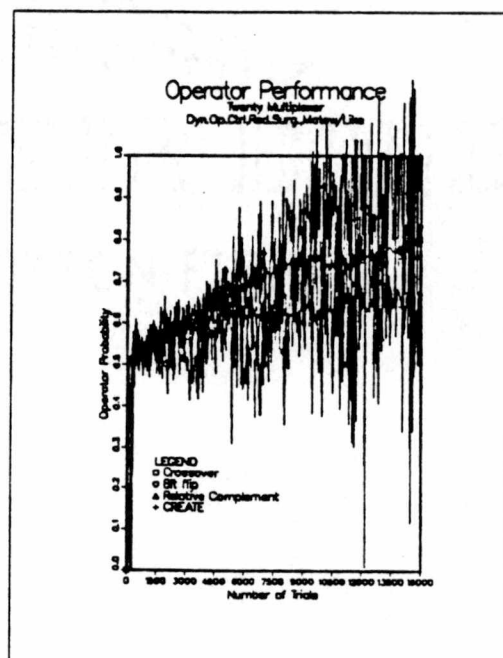
(a)



(b)

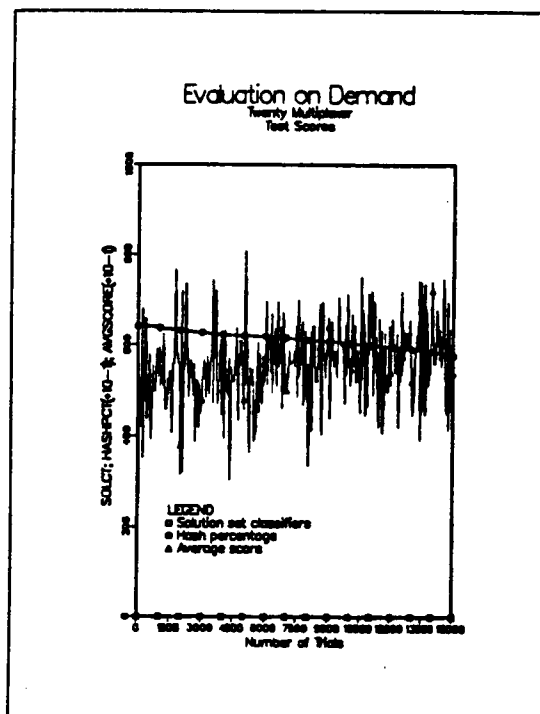


(c)

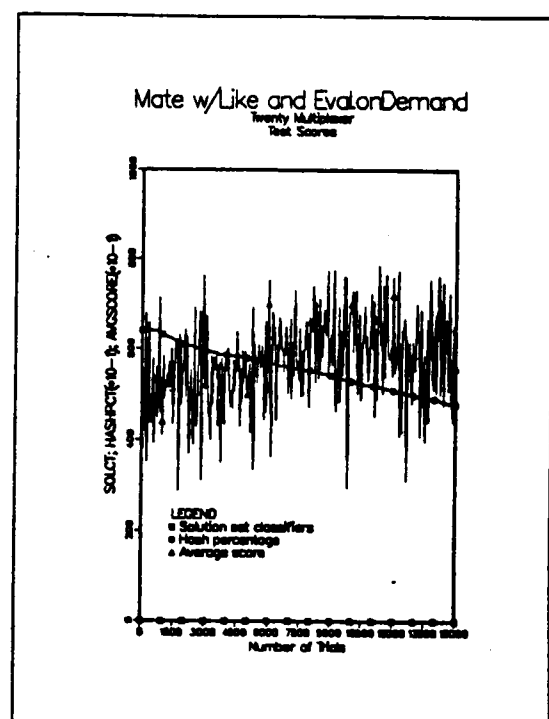


(d)

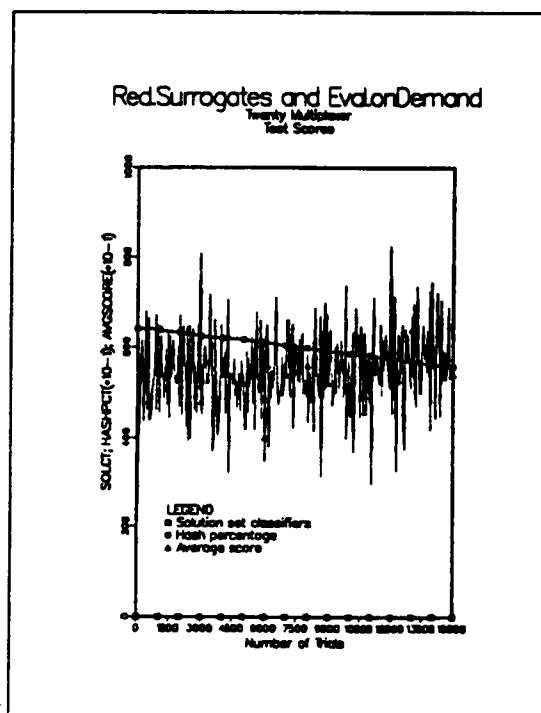
Figure C-8. Dynamic operator control operator performance curves for the twenty multiplexer.



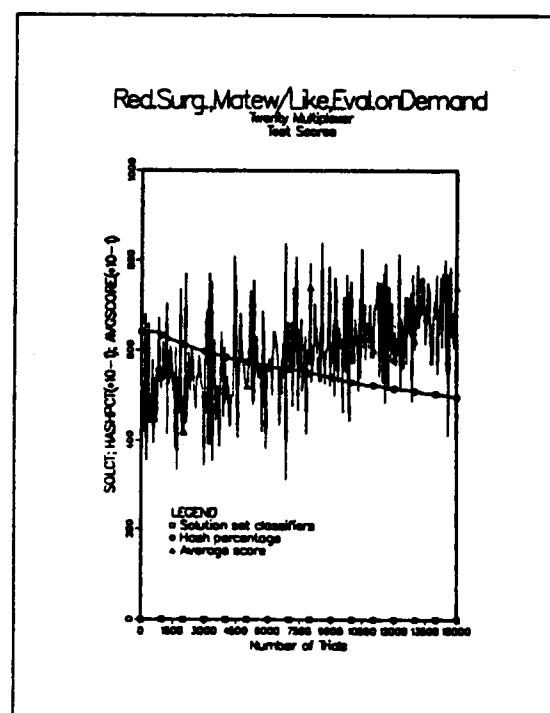
(a)



(b)



(c)



(d)

Figure C-9. Evaluation on demand for the twenty multiplexer.

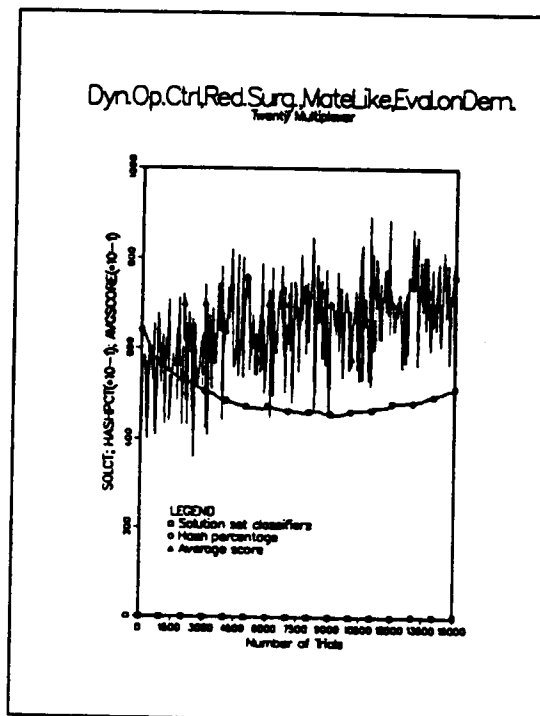
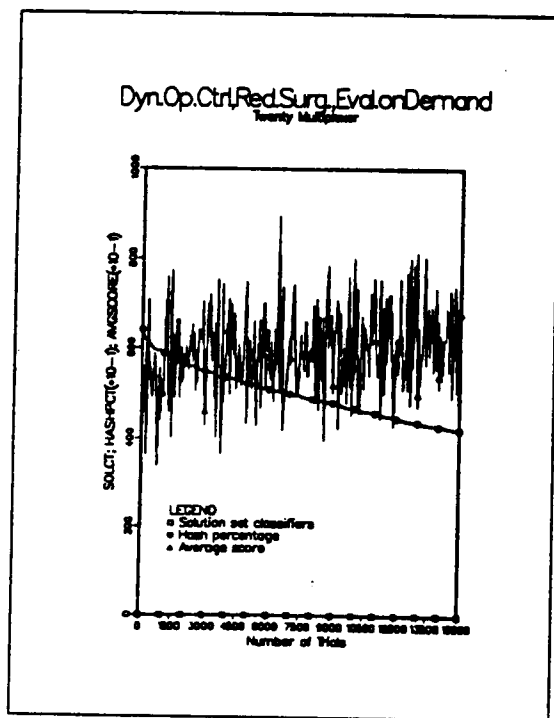
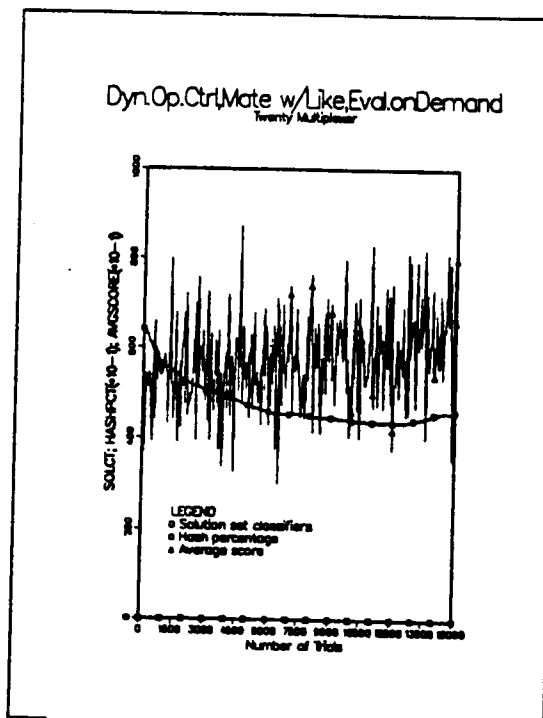
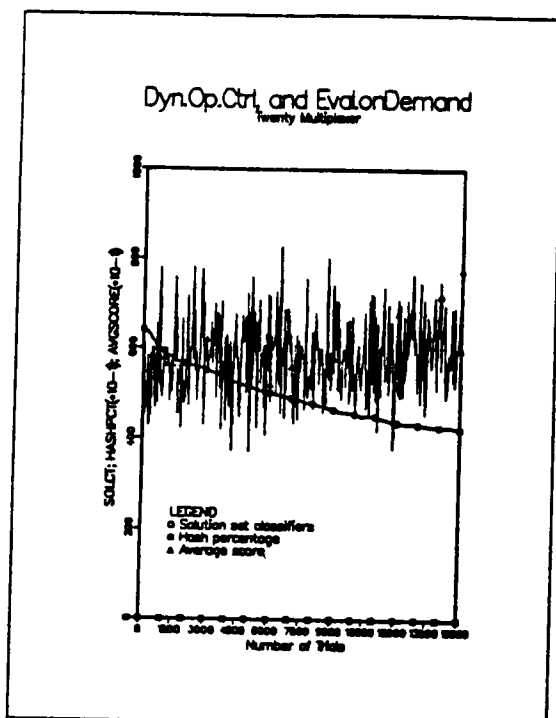
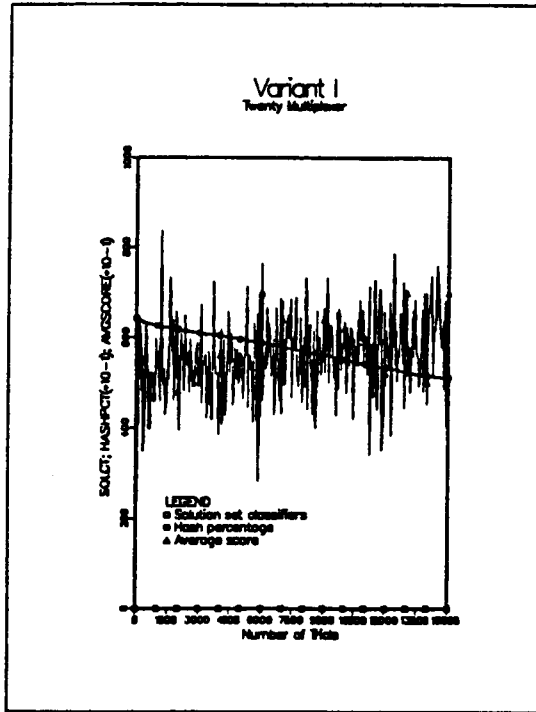
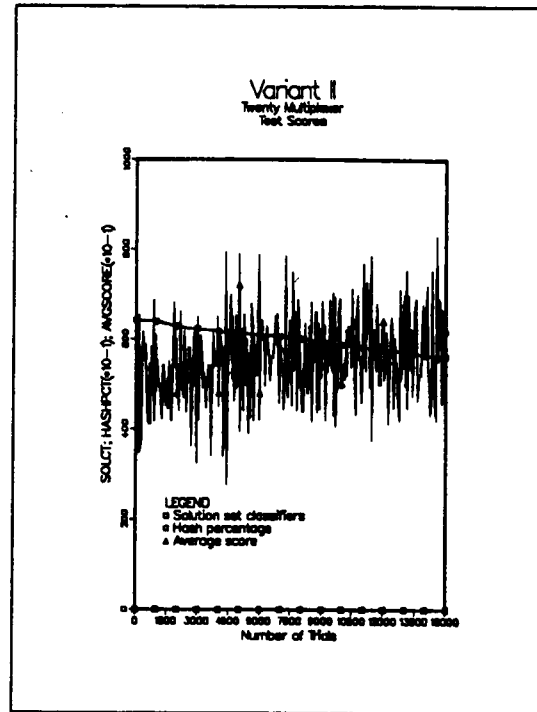


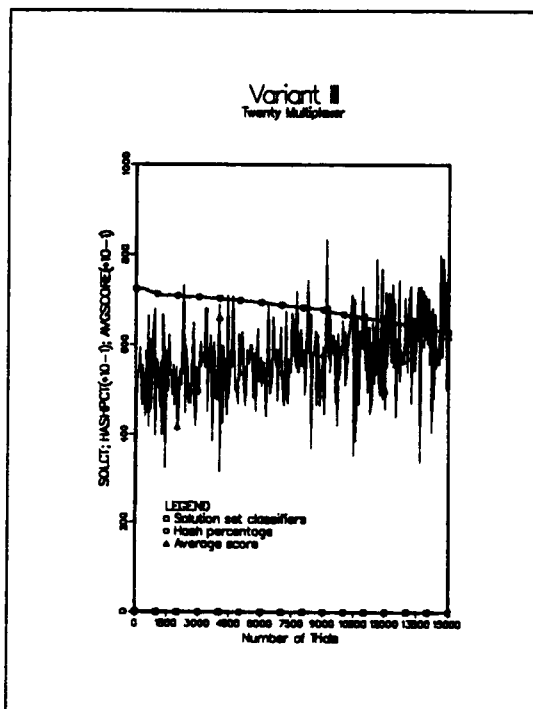
Figure C-10. Dynamic operator control and evaluation on demand for the twenty multiplexer.



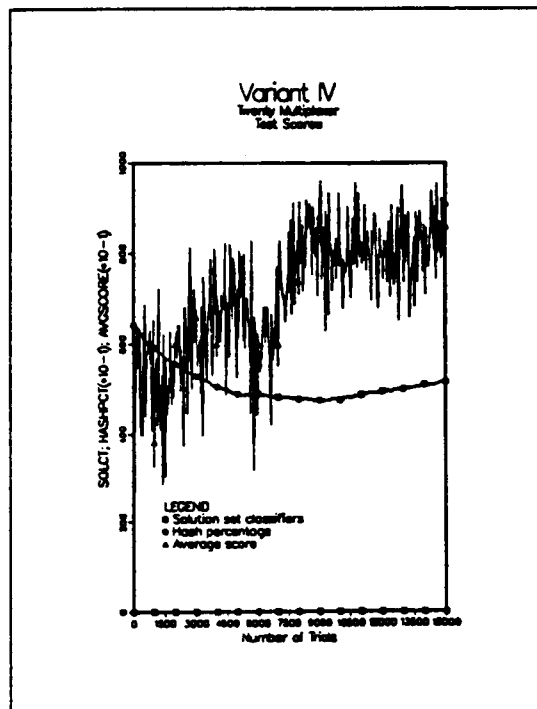
(a)



(b)

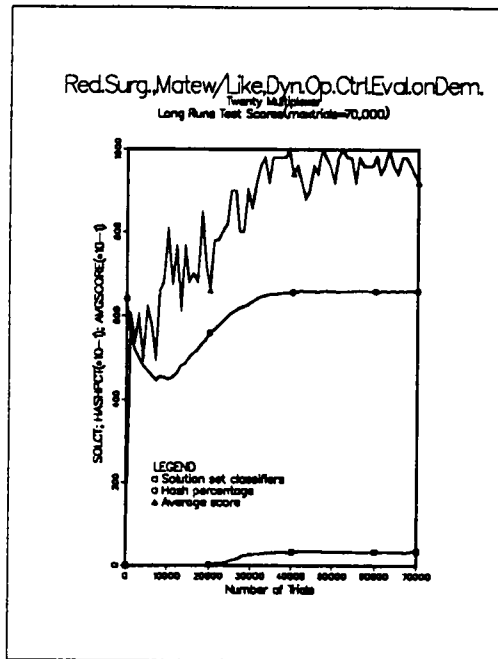


(c)

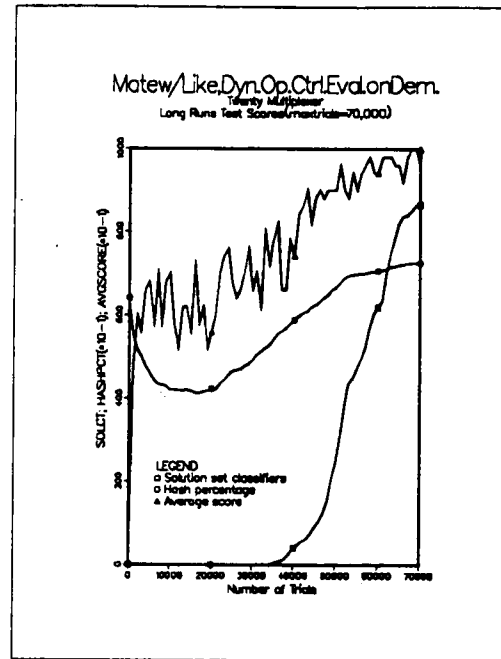


(d)

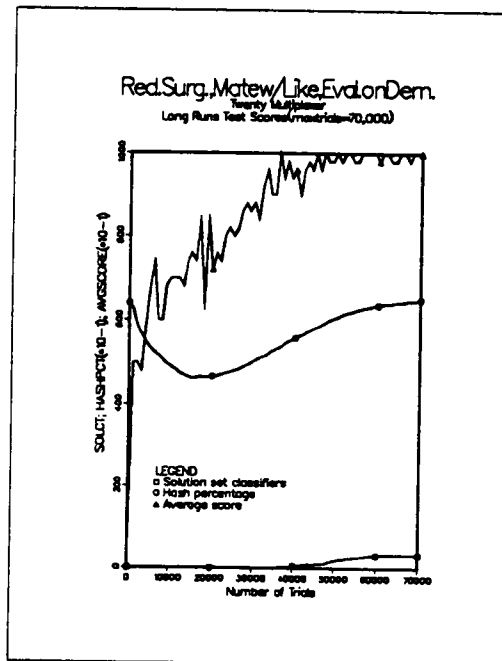
Figure C-11. Variant I, II, III, and IV for the twenty multiplexer.



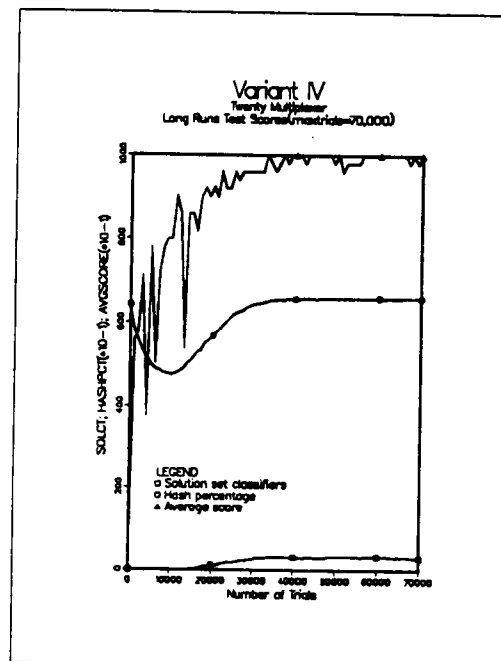
(a)



(b)



(c)



(d)

Figure C-12. Long runs(maxtrials=70,000) for the twenty

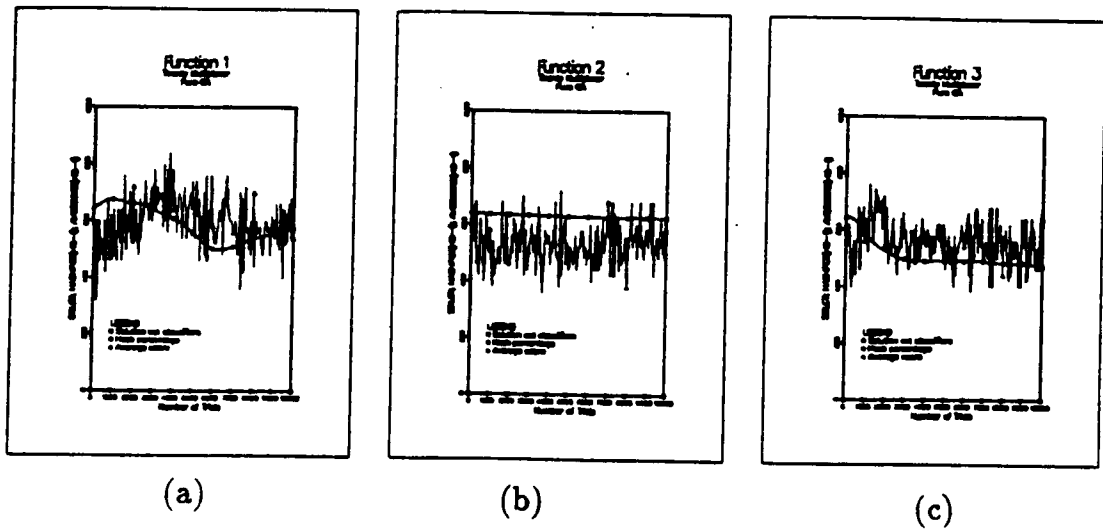


Figure C-13. Pure-GA static evaluation functions for the twenty multiplexer.

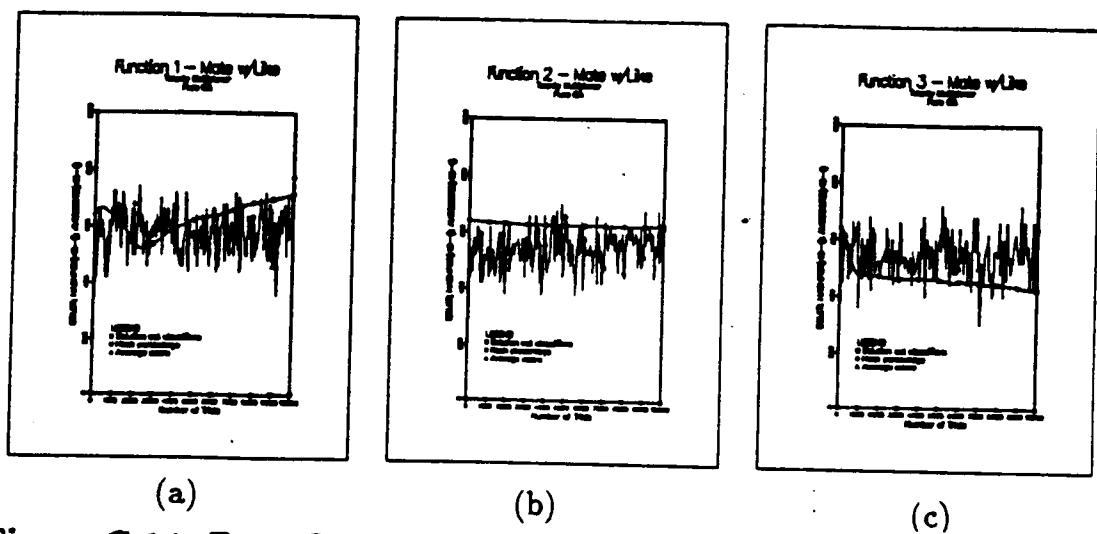


Figure C-14. Pure-GA using speciation for the twenty multiplexer.

VITA

Lori A. Wang was born on May 10, 1956 in Springfield, Massachusetts but she grew up mostly in central Florida and the southeastern United States.

In the winter of 1982, she returned to the University of Tennessee as a re-entry undergraduate student. On September 17, 1982, her son, Michael Zhong-Jei Wang, was born. In June 1986, she graduated with highest honors from the University of Tennessee with a B.A. in computer science.

After graduation, she gained valuable experience and knowledge as systems manager for Technology for Energy Corporation's corporate VAX. While at TEC she entered graduate school part-time to pursue a master's degree in computer science. Later, she received a graduate assistantship with the university's Office of Information Systems where she had the privilege and pleasure of meeting many of the administration's PC users.

She is now employed with UTK as a subcontractor with Oak Ridge National Laboratory, where she works for the Office of Risk Analysis. She is presently applying neural networks as a tool in the statistical analysis of toxicological data and in the study of chemical structure-activity relationships(QSAR). She plans to further her education by working part-time on a doctorate in environmental toxicology.