UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Terry B. Van Belle

TITLE OF THESIS: A New Approach to Genetic-Based Automatic Feature Discovery

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1995

(Signed) . . . .

Terry B. Van Belle
10517 69th St.
Edmonton, Alberta
Canada, T6A 2S7

Date: . . . .

UNIVERSITY OF ALBERTA

A New Approach to Genetic-Based Automatic Feature Discovery

BY

Terry B. Van Belle

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

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
Fall 1995

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **A New Approach to Genetic-Based Automatic Feature Discovery** submitted by Terry B. Van Belle in partial fulfillment of the requirements for the degree of Master of Science.

. . . .

Dr. J. Schaeffer (Supervisor)

. . . .

Dr. P. Dixon (External)

. . . .

Dr. J. Culberson (Examiner)

(Chair)

Date: . . . .

Abstract

Systems which take raw data and categorize them into discrete classes are ubiquitous in computer science, having applications in fields such as vision, expert systems, and game playing. These systems work by extracting features from the data and then combining the values of the features to form a judgement. While much work has been done on ways to automatically combine feature values, the task of automatic discovery of these features is recognized to be much more difficult, and so has become one of the holy grails of machine learning. Classifier systems, an outgrowth of genetic algorithms, seemed a promising approach to automatic feature discovery, but it is difficult to get the full power of the classifier system from existing implementations.

This thesis simplifies the classifier system into a variant of the genetic algorithm, called the Population Genetic Algorithm (PGA). PGAs are used to automatically discover features for tic-tac-toe and checkers endgame positions, and these features are automatically combined using Bayesian statistics to classify each position as won, lost, or drawn.

The theoretical maximum performance of the PGAs is determined by using an exhaustive enumeration technique to serve as a baseline comparison. The results indicate that while PGAs can be made to perform at near-optimal levels, the optimal solution is insufficient to perfectly classify any of the domains studied.

Acknowledgements

The following people were invaluable to me in the production of this thesis. My deepest thanks go to all of them:

Jonathan Schaeffer, my supervisor, for his guidance, encouragement, and experience,

Joe Culberson and Peter Dixon, my examining committee, for taking the time to read my thesis and offer their valuable suggestions,

Andreas Junghanns, Yngvi Bjornsson, and Mark Brockington for their input and enthusiasm,

and Edmund Dengler, who helped develop many of the initial ideas.

Contents

List of Figures

Chapter 1

Introduction

In the field of machine learning, the problem of supervised learning is a prominent research area. It can be summarized as follows: Given a set of training data with a correct response for each datum, construct a system which learns from those data so that it can give the correct response for any datum. The way in which the system must respond depends on the specific problem. If the data are categorized into a set of fixed categories, the task is to construct a system which will correctly classify any datum. If the data are accompanied by a recommended course of action, the task is to produce the correct action in all situations. The system generally determines its response by extracting relevant features, or numerical abstractions, from the data and then, on the basis of these feature values, arriving at a judgement.

Extracting and combining feature values has a wide base of applicability. A computer vision program may identify different faces based on the facial features of each. A medical expert system uses the answers to questions it asks as features to determine what sort of disease the patient has. In computer game playing, the data is the set of possible board configurations, with the system either classifying them as a win, loss, or draw, or recommending a particular move. There is even an analogous process in our visual system. It is now known that certain classes of neurons in the visual cortex will only respond to stimuli of certain orientations, movements, spatial frequencies, or retinal disparities [25].

In the field of game playing, programs must be able to search a reasonable number of positions, and they must be able to evaluate each position to determine how likely it is that the position will lead to a win. While the most successful game-playing programs for chess, checkers, and Othello[1] have relied primarily on the fast searching of a large number of positions, the applications of supervised learning to the improvement of a program's position evaluation function have provided some notable successes. As far back as 1959, Arthur Samuel was doing experiments in machine learning for checkers, ranging from rote learning to the adjustment of feature weights to evaluate positions [33, 34]. For the game of backgammon, Gerald Tesauro's Neurogammon [40] used neural nets trained by backpropagation to learn which move to recommend. Kai-Fu Lee used Bayesian learning for his Othello program, Bill 3.0 [28], and the concepts have been extended for Michael Buro's Othello program, Logisthello [7]. Neurogammon, Bill 3.0, and Logisthello have all been computer world champions.

However, virtually all attempts at machine learning for game playing have concentrated on automatic methods of *combining* various feature values extracted from the board, and relied on human expertise to determine what these features will be instead of doing this automatically. A small amount of work has been done using symbolic AI [15] for automatic feature discovery, and Tesauro has claimed that his neural nets have this capability [39], although he was not able to get master-level performance from his nets until he added hand-crafted features.

Samuel himself regarded the automatic discovery of features as a necessary extension for truly flexible machine learning of game playing, and this sentiment has been echoed several times since then [33, 40, 28, 17]. The discovery of an automatic feature generation system with reasonable performance would be a great aid to the field of machine learning because traditional feature selection requires the user to decide ahead of time what is relevant and what is not for a domain. Ideally one wishes for a machine learning system to be a "black box" in which no domain dependent knowledge is required, although this ideal is surprisingly difficult to attain in prac-

[1] Also known as Reversi.

tice, and researchers are starting to conclude that no learning algorithm will work for all domains. The advantages to automating aspects of the machine learning system, however, lie in the fact that they can be applied to problems for which there is little human knowledge, that they are free from human biases and that they may uncover more information about a field which has already been well studied with traditional techniques. Unfortunately, despite many decades of research, no practical approach has yet been discovered for the problem of automatic feature discovery, making it a sort of holy grail for computer game playing and other areas of machine learning.

One promising approach was presented during the 1980's by John Holland of the University of Michigan. His approach, known as Classifier Systems, uses a large number of simple rules in a production system to recommend a course of action based on input. The learning comes in the form of reinforcement from the environment when the system behaves correctly, and new rules are generated using a "survival of the fittest" method called Genetic Algorithms, also developed by Holland [18, 5, 23].

Despite their promise, the results of applying classifier systems to anything beyond "toy" problems have been disappointing because of the enormous complexity of the systems, their myriad of parameters, and their non-linear behaviour. Subtle design decisions can exert a surprising and often unwelcome influence on the system's performance. In cases like these it can be difficult to determine which part of the system is responsible for its failure to perform.

This thesis presents a form of automatic feature discovery using a simplified form of the classifier system. Features are discovered using a modified genetic algorithm and then combined using Bayesian statistics to produce a final classification of game positions as winning, losing, or drawn. The notion of the system as a black box is discarded in favour of a set of techniques designed to evaluate the absolute performance of the system and to diagnose and correct any weak areas. Although the focus of this thesis is on game playing, the techniques discussed here could conceivably be applied to more general areas like computer vision, expert systems, and data compression.

Chapter 2 gives an overview of genetic algorithms and classifier systems, along with a more detailed discussion of their variants and problems, and then presents Population Genetic Algorithms (PGAs) as a synthesis of the two. Chapter 3 contains a discussion of the methodological principles which guided the research in this thesis, and provides a functional breakdown of the problem, illustrating various issues and previous work done in each category. Chapters 4 and 5 contain the results of applying population genetic algorithms to the problem of automatic feature discovery for tic-tac-toe[2] and checkers endgames. Chapter 6 is an overview of possible extensions and applications for such systems.

[2]Also known as noughts and crosses.

Chapter 2

An Overview of Evolutionary Computation

Evolutionary computation refers to the set of all algorithms which are modeled after the process of Darwinian evolution. As such, it falls under the broad category of Artificial Life, or computation inspired by biological processes. Two common types of evolutionary computation are Genetic Algorithms and Classifier Systems. This chapter provides an overview of these two techniques, and presents a third alternative, Population Genetic Algorithms, or PGAs, which were developed by the author and lie in between genetic algorithms and classifier systems in terms of complexity.

2.1 Genetic Algorithms

Genetic Algorithms are the oldest form of evolutionary computation in use today, developed by John Holland and his colleagues and students at the University of Michigan in the 1970's for optimization problems [18]. They work by generating a population of possible solutions to the problem and defining the "fitness" of each solution to be its closeness to the ideal solution. A process similar to natural selection operates on solutions as they reproduce and mate, and this leads to an overall increase in the fitness of the population, hopefully culminating in the emergence of an individual

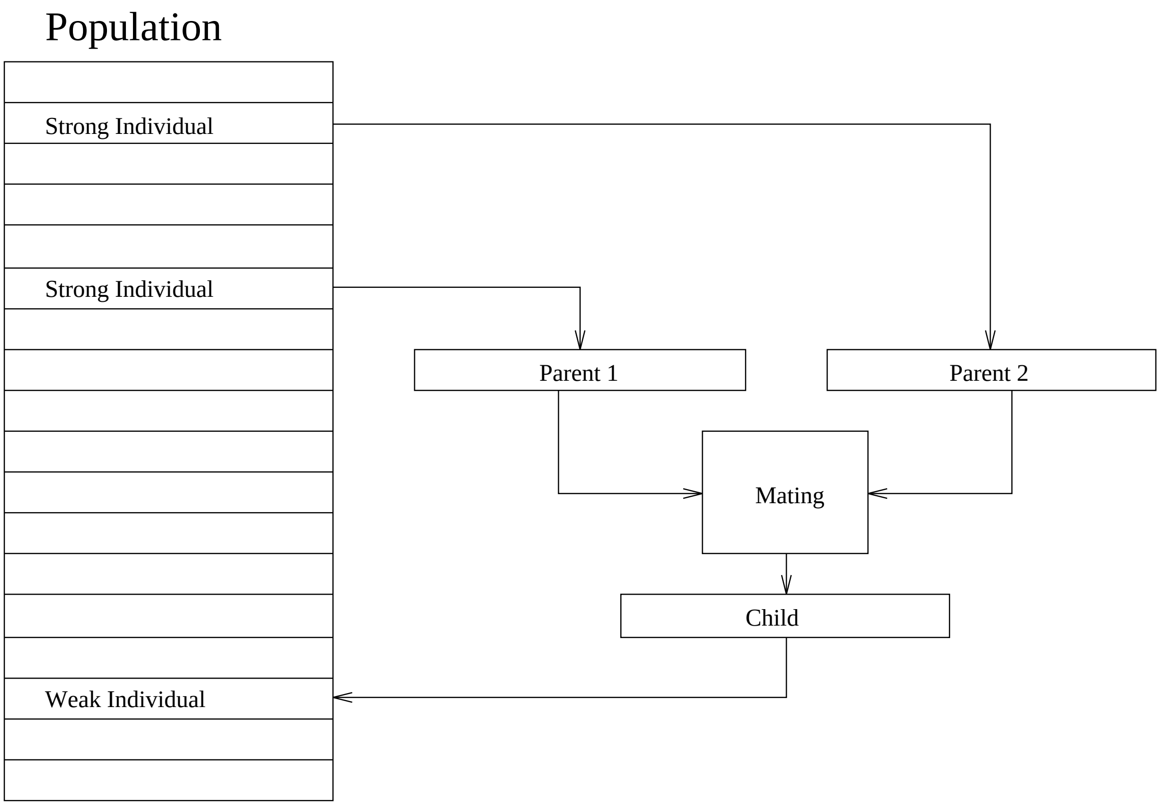


Figure 1: Structure of the genetic algorithm used in this thesis

representing the optimal solution.

The canonical genetic algorithm follows these steps:

1. Determine an encoding of the solution as a fixed-length bit string. For example if you know that the solution you are looking for is a number, you could choose a number of bits large enough to cover the range, and use a standard base 2 encoding.

2. Generate a fixed-size population of random bitstrings, which represent random solutions to the problem. In the initial population some strings will be fit (ie. represent values close to the optimal) by sheer chance, and others will be unfit. It is unlikely for a non-trivial problem that the population will contain an optimal solution.

10110110101001001
00010010011100011

10110110011100011

Crossover

10110110101001001
00010010011100011

101101100011

Messy Crossover

$1001101\boxed{1}011011110$

$1001101\boxed{0}011011110$

Mutation

$1001101\boxed{10110}11110$

$1001101\boxed{01101}11110$

Inversion

Figure 2: Some common mating techniques

3. Create a next-generation child by selecting two parents from the population and mating them. There should be a random component to selecting the parents, but it should be biased towards the fitter individuals. One possible way to do this is to make the probability of selecting an individual be proportional to its fitness. This is known as "Roulette Wheel" selection.

   There are many different techniques for mating together two parents, including crossover, messy crossover, mutation, and inversion. Crossover combines the two parents by cutting them both at a random place (but the same place for both strings) and then pasting the first half of one with the second half of the other. Messy crossover is like crossover, except that each parent can be cut at a different place. Mutation is the process of selecting random bits of the child and flipping them to the opposite value. Inversion is the process of reversing the order of symbols in an arbitrary substring of the child (see figures 1 and 2).

4. Depending on the type of genetic algorithm, either use this new child to replace one of the older individuals in the population, or add it to a new population, which replaces the old population when full.

5. If an individual with an acceptable fitness is in the population, or a preset number of generations has passed, then terminate, and present the most fit individual in the population as the solution. Otherwise, go to step 3.

Since the child individuals are made up of fit individuals from the previous generation, we expect them to be fitter on average, resulting in an increase in the average fitness of the population and, hopefully, a convergence upon the optimal solution to the problem.

2.1.1 The Schema Theorem

This expectation of convergence has been formalized in Holland's Schema Theorem [18]. A schema, according to Holland, is a subset of all possible bitstrings of the length chosen for the genetic algorithm. So, for example, the schema $**0*1*1*$ represents a subset of all 8-bit strings, with the third bit being 0, and the fifth and seventh bits being 1. The $*$ character in this context means "don't care" and can match either a 0 or a 1.

If we define the fitness of a schema to be the average fitness of all strings in the population which match the schema, then the Schema Theorem says that we can expect the number of occurrences of any particular schema to grow roughly according to its fitness divided by the average fitness of all schemata in the population. Other factors in the theorem account for the chance of a schema being destroyed by crossover or mutation.

Holland has also shown that binary string encodings are optimal for processing the maximum number of schemata per generation. This result has been widely cited as justification for only using binary encodings for any problem [19], although this is changing [16].

The intuitive argument goes as follows: Consider two different alphabets for encoding the numbers from 0 to 15. We could use the standard binary encoding $0000, 0001, 0010, \ldots, 1111$, or we could use a symbol for each number, like $A, B, \ldots, O$. The latter scheme needs only one symbol to encode a number, while the former needs four. However, while there are no similarities at all between the different letters, the genetic algorithm can exploit similarities between different binary representations. For instance, the schema $1***$ represents all numbers greater than 7, and the schema $***0$ represents all even numbers. The number of schemata per bit of information can be calculated. For an alphabet of cardinality k, it is $\sqrt[\log_2 k]{k+1}$, which is minimized, subject to $k \geq 2$, when $k = 2$.

2.1.2 Modifications to the Canonical Genetic Algorithm

Avoiding Premature Convergence

While the canonical genetic algorithm described above is most often used as an introduction to genetic algorithms, it is almost never used in practice, because roulette wheel selection leads to problems. As time goes by, the variation in the population goes down as more successful strings take over. Inevitably the population converges to multiple copies of one particular solution to the problem. If this happens too quickly, the solution which comes to dominate the population may not be optimal, or anywhere near optimal. This problem is known as premature convergence, and roulette wheel selection is particularly susceptible to it.

Furthermore, there are classes of problems in which there may be several good solutions to a problem, all of which should be found by the genetic algorithm. This quality is called *multimodality*, and we would ideally wish to have the population distributed so that each solution is represented by a number of instances proportional to its fitness. Much research has been devoted to devising parent selection schemes which accomplish this. Three of the most common approaches are crowding [11], sharing [12, 24], and local mating [10]. All three approaches attempt to maintain variation in the population by encouraging unique strings and/or checking the fecundity of particularly fit strings. Of the three, sharing has been shown to be superior [12, 24].

A more extreme approach to selection is employed by rank-based schemes such as work done by Baker [3], and Darrell Whitley's GENITOR algorithm [41, 42]. Under a rank-based scheme, the absolute fitness of a string is unimportant. Instead, a method of determining if one string is more fit than another is required. The initial population is sorted according to this criterion, and then new children are inserted into the population, pushing out the lowest-ranked strings. The probability of a string being selected is a function of its rank in the population. Since this approach does not depend on absolute fitness values, there is no chance of an individual prematurely dominating the population by virtue of its fitness being far higher than its neighbours.

Non-Binary Alphabets

Recently, Holland's conclusion that binary alphabets are optimal has come under attack [1]. The argument is that defining schema using a $\{0, 1, *\}$ alphabet implicitly biases the results towards the conclusion that binary alphabets are optimal. A higher cardinality alphabet would involve a more complex schema language. For example, if we encoded our individuals using the ternary alphabet $\{0, 1, 2\}$, then the appropriate schema alphabet to consider would be $\{0, 1, 2, *_{01}, *_{02}, *_{12}, *_{012}\}$, where, for example $*_{02}$ would match the symbols 0 and 2, but not 1. When such an expanded concept of schema is used, the results are reversed, and higher cardinality alphabets are favoured for the greatest amount of schema processing per generation.

In a separate subfield of evolutionary computation this approach has been used successfully for many types of problems. Genetic Programming, developed by John Koza [26], uses genetic algorithm techniques to evolve LISP functions to solve specific problems. These functions are usually stored as trees. Instead of the traditional crossover operator, a subtree is extracted from one function and grafted to replace a subtree of the other function.

Messy Genetic Algorithms

Finally, it is possible to ignore the constraint that all strings be the same length. Such algorithms are called "messy" genetic algorithms [13]. The traditional crossover operator is replaced by a messy operator which allows each string to be cut at a different point. Then the first part of one is pasted on the second part of the other, just like in traditional crossover.

While messy encodings remove the constraint that all encodings must be the same length, they introduce the new constraint that the developer cannot attach semantic meaning to fixed bit positions, or ranges. For example, in a fixed-length encoding, the developer may wish to use bits 8 to 15 to represent an 8-bit numerical field. Under messy crossover, these bits will likely end up in a different position in the string, and the bits in positions 8 to 15 (if they exist at all) will be from some other part of the parents. Hence, position-independent semantics are necessary to intelligently decode the string into a solution.

One further potential problem with messy encodings lies in the fact that since the length of the string changes, there may be too little information in the string to completely specify a solution. At the other extreme, there may be too much information, leading to conflicts between different parts of the string.

2.2 Classifier Systems

More recently, Holland has developed a new adaptive technique known as classifier systems. Classifier systems incorporate genetic algorithms as part of their structure, and are more powerful (in fact, they have been shown to be Turing complete [18]). Whereas genetic algorithms are primarily used for simple optimization problems, classifier systems are designed to adapt to a real-time environment which offers sporadic rewards, in a fashion reminiscent of behaviourist operant conditioning.

The canonical classifier system consists of an interface to the environment in the form of bit-encoded inputs and outputs, a population of rules known as classifiers, working memory in the form of a message list, and a system for improving the population over time. The inputs are analogous to the system's senses, the outputs are like motor commands, and the classifiers and memory comprise the brain of the system (figure 3).

Classifiers have two parts to them, a condition and an action. Conditions are fixed-length strings from the alphabet $\{0, 1, \#\}$, where the # matches either a 1 or a 0. Actions are strings of the same length as conditions, and using the same alphabet, but the # symbol now acts as a pass-through: if the condition matches a bitstring, another bitstring is generated from the action part, with the # symbols filled in with corresponding bits from the matched bitstring. The standard classifier system also

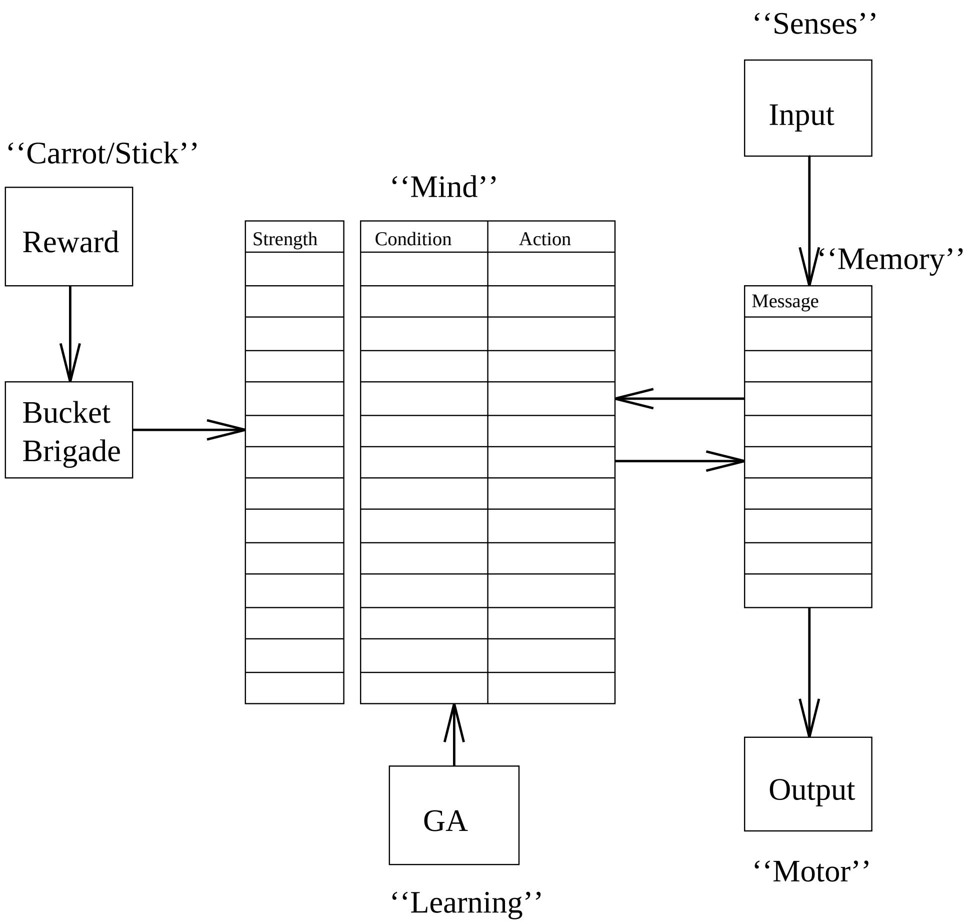


Figure 3: Structure of a classifier system

allows multiple conditions in a classifier, and for conditions to be negated. Multiple conditions must all match for the rule to match, and negated conditions must not match for the rule to match.

The strings which are given to the classifiers to match are called messages. They can either be supplied by the environment or produced by the act of a classifier matching. All messages are the same length. Inputs from the environment are encoded into binary messages, which are added to the message list. Then all of the classifiers are checked against the messages, and the classifiers that match compete to see who will be allowed to put their own messages onto the message list. The developer builds an arbitrary scheme to determine which messages will be considered output messages, usually by reserving one or two bytes as a tag. Output messages are collected, decoded, and sent to the environment.

If the environment decides that the outputs were correct, it rewards the system with a numerical payoff, which is awarded to all classifiers that were responsible for the output. Thus, over time, those classifiers which aid the system should be strengthened and those that are useless should be weakened. Periodically a genetic algorithm is run over the population to weed out the bad classifiers and generate new ones from the strongest of the population.

A large portion of the classifier system's power depends on it being able to "chain" classifiers, that is, to have one classifier produce a message which is used by another classifier. If this can be achieved, the classifier system can rise above a simple stimulus-response mode of behaviour where all classifiers only match input messages and only produce output messages. To encourage chaining, Holland developed an algorithm known as the Bucket Brigade algorithm. It works as follows: Matching classifiers compete for access to the message list by taking a portion of their strengths and offering it as a bid. Based on the size of the bids, the system stochastically decides who will be allowed to put their message on the message list, and the bids are deducted from those rules' strength. In addition, if a classifier matches another classifier's message, the matching classifier must pay the message-producing classifier a certain amount of its strength. Finally, a small tax on strength is exacted from all classifiers, so that inactive classifiers will eventually drop out of the population. The hope is that if classifier A uses classifier B's message to get rewards from the environment, then it will pass parts of its rewards to B, over time. Classifier B, in turn, may have matched a message from classifier C, and pass on a portion of its reward to C.

The second hope of classifier systems is that they will spontaneously decompose the population into a number of relatively weak general rules and a number of relatively strong specific rules. This setup is known as a "default hierarchy," and it would presumably reduce the number of rules required to handle a task by relegating default behaviour to a group of general classifiers, with more specific classifiers overriding on special occasions, a form of organization similar to Rodney Brooks' subsumption architecture [6].

2.2.1 Problems with Classifier Systems

Perhaps the most troubling problem with classifier systems is that they very rarely live up to their expressive potential by forming rule chains or default hierarchies, and those that do form are usually not maintained by the system. In David Goldberg's review of the state of the art in classifier systems [43], the number of instances of successful chaining or default hierarchy formation could be counted on the fingers of one hand. A variety of techniques have been attempted to encourage the formation and maintenance of these arrangements [31, 21], but for the most part there have been no breakthroughs.

One of the great difficulties in diagnosing and fixing problems is the sheer number of parameters involved in the average classifier system implementation. Rick Riolo's CFS-C system [32], for example, has 38 parameters, many of which are likely to interact non-linearly. Most approaches to fixing problems with classifier system performance have been oriented towards adding extra features to the algorithm, each carrying its own set of parameters, which only aggravates the problem. The truth is that there is no good theory of classifier systems which accounts for the interactions among the myriad of techniques used by researchers. To exhaustively enumerate all combinations would be prohibitively expensive, tedious, and likely only relevant to the test cases studied. Some work has been done on using a second-level genetic algorithm to optimize the parameter values for a first-level genetic algorithm [20], but little has come of this.

When considering the genetic algorithm part of a classifier system, we find that the problem of premature convergence has become a crisis. One of the great strengths of classifier systems is that the entire population is used as the solution to the problem instead of just the top-scoring individual. Unfortunately this places an additional constraint on the system in that all aspects of the correct solution must be discovered and maintained somewhere in the population. Genetic algorithms, with their emphasis on finding one solution to a given problem, not surprisingly show a strong tendency to converge on one individual only. This is unacceptable behaviour, even if that individual is the best possible classifier. To borrow from Donne, no classifier is an island.

Carrying over experience from genetic algorithms, we can see that classifier systems represent a problem with extremely high multimodality. In fact, the ideal classifier system population would be composed completely of unique classifiers, each representing one of the best problem solvers. However such a population aggravates another problem in generating a good set of classifiers: the large number of low-fitness classifiers which are generated when two completely different classifiers are mated together. Since each classifier fulfills a different role and is likely to have a different structure, it is very unlikely that the combination of the two will generate good offspring. Booker solves this problem by introducing the technique of inbreeding, where only individuals which are active at the same time are allowed to mate [4]. Instead of our ideal population, then, we settle for one composed of many subpopulations, each representing a different aspect of the solution. This effectively cuts the number of useful classifiers in a population of size n to n/s, where s is the average size of a subpopulation. This value must be large enough to provide sufficient variation for the genetic algorithm to work on.

Finally, the fact that classifier systems use bitstrings as messages and the alphabet $\{0, 1, \#\}$ for classifiers generates a subtle class of problems. As mentioned before, it is widely believed that a low-cardinality alphabet is best for processing large numbers of schemata simultaneously, as it takes advantage of naturally occurring regularities in the representation. The natural question to ask is whether we *should* be taking advantage of schemata which are likely to be serendipitous artifacts of an arbitrary encoding scheme, for such artifacts represent a double-edged sword. Not only can an advantageous encoding aid in search, but a poor encoding can significantly hamper the chances of a classifier system discovering a good solution.

Take, for example, the task of encoding a tic-tac-toe board. Each space may contain an X, and O, or a blank, so we need two bits to completely encode a square. Since two bits encode four values, we will have one left-over state, which we will call illegal. Now, if we encoded the square contents as follows:

X	$\rightarrow$	01
O	$\rightarrow$	10
Space	$\rightarrow$	00
Illegal	$\rightarrow$	11

the genetic algorithm would run into trouble in the future if the concept of 'not empty,' (ie. either X or O) was necessary. There is no combination of 0, 1, and # which can represent it, and we, the users of the classifier system, might never realize that this was the reason for the poor classifier system performance. On the other hand, if we have the foresight to see that this concept could be useful to represent, we might as well make it explicit in the alphabet by supplying a 'not empty' symbol. Suppose that we use an encoding which prevents this problem, like:

X	$\rightarrow$	00
O	$\rightarrow$	01
Space	$\rightarrow$	10
Illegal	$\rightarrow$	11

Now the pattern 0# will encode the 'not empty' condition, but even now we are introducing arbitrary biases in the genetic search. To see why, consider the effect of mutating one of these representations:

00 (X) mutates to	10 (Space) or 01 (O)
01 (O) mutates to	11 (Illegal) or 00 (X)
10 (Space) mutates to	00 (X) or 11 (Illegal)

Of the 6 possible outcomes, X is twice as likely to come out as O or Space. Thus our encoding biases the mutation operator, and hence the search, towards boards with many X's. Crossover is even worse:

00 (X) crossed with 01 (O)	gives	01 (O) or 00 (X)
00 (X) crossed with 10 (Space)	gives	00 (X) or 10 (Space)
01 (O) crossed with 10 (Space)	gives	00 (X) or 11 (Illegal)

Now fully half of the resulting outcomes represent X.

On the other hand, if we encoded the square using a higher cardinality alphabet like $\{X, O, Space\}$, not only would we have direct control over the transposition probabilities between symbols, we would be able to eliminate that annoying Illegal symbol. These examples, and other problems with binary encodings, can be found in [35].

2.3 A New Synthesis: Population Genetic Algorithms

Despite their problems, classifier systems have one great advantage over genetic algorithms: they use the entire population as a solution rather than just the fittest individual. To see why this is important, consider the task of evolving a multiple-rule production system like the one used in classifier systems, in the context of the genetic algorithm. Because only one individual in the population is chosen among many, it is necessary to encode the *entire* rule set of the system into each individual. This approach was used by S.F. Smith in his LS-1 system, whose most famous application was a Poker playing system [36]. A comparison with the canonical classifier system model, CS-1, can be found in [18].

An intuitive argument can be made for the increased problem complexity from using LS-1 as opposed to CS-1 by considering two multiple-rule production systems, each which must discover R rules. Let n be the number of rules encoded in each individual. The LS-1 approach would set $n = R$ and use N individuals. If we wish to keep the number of potential rule solutions in a population the same for both approaches, CS-1, which sets $n = 1$, would be allowed to maintain a population of NR individuals without increasing the amount of computation per generation.

If it takes b bits to encode each rule, then the search space for the LS-1 system would be bR bits long, for a total of 2^{bR} possible rule sets. The complexity of the task is thus exponential with respect to the number of rules desired. On the other hand, the CS-1 approach decouples the number of rules from the search space size, so that we are searching for R instances from the much smaller space of 2^b possible rules, a linear increase in R. While it is likely that genetic search would reduce the complexity of the LS-1 approach to less than exponential, and increase the complexity of the CS-1 approach to more than linear, clearly LS-1 represents more work.

One advantage of the LS-1 approach over CS-1 is that we can more accurately test the true fitness of an LS-1 rule, which is its performance in the context of the best possible set of co-rules. Since LS-1 treats entire rule sets as individuals, this quality is optimized for that system, while the rules of CS-1 must be tested on their own. Calculating the true fitness of a CS-1 individual would require generating all $2^{b(R-1)}$ possible sets of its of co-rules. Fortunately it is not necessary to resort to such measures, as will be seen in the next chapter.

2.3.1 Application to Data Classification

In traditional classifier systems, the mechanisms of reward and bucket brigade are used to determine the strength of each rule in the population. As we have seen, these mechanisms have failed to produce the performance one would expect from an adaptive Turing-complete engine. Thus, when considering the use of classifier systems, we do well to heed the lessons learned by previous researchers and strip out any unnecessary features and parameters.

A straightforward application of classifier systems to classifying game boards would produce rules of the following types:

- **IF** *condition* **THEN** *recommend Win*
- **IF** *condition* **THEN** *recommend Loss*
- **IF** *condition* **THEN** *recommend Draw*
- **IF** *condition* **THEN** *add message M*

Classifiers which matched the board and/or current message list would compete to determine which outcome was assigned to the board. If the classification was correct then all matching classifiers would be rewarded.

The first simplification we can make is to eliminate internal message passing. While it may be important to maintain the expressive potential of the system, we saw in the chapter on classifier systems that it is a troublesome feature to get working. Thus it is important to determine how necessary it is to a specific task, and only use it if the performance without it is unsatisfactory.

The second thing to note is that since we have all of our training data on hand at once, we can iterate through it as many times as we like, and in any order. The notion of time between payoffs becomes irrelevant as well. Thus we do not have to incrementally approximate the value of each rule over time, nor use a simplistic punishment/reward scheme. Furthermore, we do not have to maintain a high degree of system performance throughout the entire run, or worry about exposing each classifier to a sufficient amount of "experience." Thus, the whole issue of when to explore for more solutions and when to exploit the ones we have is avoided.

One final simplification can be made by recognizing that for the job of classifying data, each pattern in a rule can simultaneously provide evidence for or against a number of different categories. Thus, the notion of an explicit action part to each rule is unnecessary.

When all of these modifications are made, we are left with a learning system which is somewhat less than a classifier system, but somewhat more than a genetic algorithm. This is a new type of evolutionary computation, called the Population Genetic Algorithm, or PGA, in this thesis. Thus, a classifier system should be considered a PGA, as is the simplified classifier system used in this thesis, which generates a set of optimal binary features for the classification of data; in other words, automatic feature discovery.

Chapter 3

Formal Analysis

This chapter presents a more thorough analysis of the task of applying population genetic algorithms to the problem of supervised learning for game playing. The task is decomposed into a number of mostly independent sub-tasks, and a review of possible approaches is given, along with a justification for each approach used in this thesis. Before we start, though, it is important to give an overview of some of the issues of methodology considered in the actual research.

3.1 A Note on Methodology

In the previous section we simplified the classifier system paradigm by removing any unnecessary aspects for the problem of game board classification. However, even PGAs are susceptible to problems of complexity. In this sense, the process of troubleshooting such a system is similar to that of debugging a large and complex program.

Novice programmers, when they first encounter bugs, will often pursue the non-productive strategy of guessing at the bug's location, making a minor change to code that is not obviously wrong, and then re-running the system to see if the bug went away. Not only does this approach often not solve the problem, but even worse, it can seem to fix the problem by changing the program's state in such a way that the bug is not manifested at that particular moment. The reappearance of the same bug that was presumably "fixed" is always a demoralizing experience.

More experienced programmers will approach the problem of detecting and fixing bugs in a more systematic manner. First of all they devise informal tests to determine which section of the program is misbehaving. Determining the problem area is made easier if the program is constructed in a modular way, so that the code is divided into functions which are more or less independent, with minimal interference between functions. Once the general area of the bug has been found, the programmer tests various data values against their expected values to pinpoint the first occurrence of discrepancy. Once this discrepancy has been located, it is usually easy to determine the coding error.

The methodology of such an approach can be transferred over to apply to the troubleshooting of complex systems. Instead of working with modular code to determine which module is at fault, we decompose our task into a number of functionally independent subtasks. Goldberg has used this approach to divide genetic algorithms into the subtasks of building block generation, isolation, growth, and mixing, and to divide classifier systems into the subtasks of strong/weak cooperation, rule maintenance/search, and rule accuracy/covering/generalization [24]. This thesis takes a broader approach, dividing the task of supervised learning into the subtasks of data representation, data collection, feature representation, feature selection, and feature combination.

To determine which subtask(s) is causing the lack of performance, it is important to use accurate metrics to determine the absolute performance of each subtask. Baseline comparisons are useful in this capacity to determine both the upper and lower bounds of expected performance for the subsystem performing the subtask. If we have an upper bound on performance we can determine whether our particular subsystem could do any better, and should be adjusted, fixed, or replaced by a more sophisticated approach. If we have a lower bound on performance, we can reassure ourselves that a good performance from our subsystem is not due to pure chance or an overly simple problem domain.

This analogy between code and complex systems can be extended beyond debugging/troubleshooting to the area of prevention. Code which is simple and straightforward is less likely to contain bugs, and more likely to reveal any bugs which it does contain. In the same manner, reducing the number of parameters in a complex system reduces the number of things which could go wrong and makes it more obvious where problems lie. This has been a goal throughout the research for this thesis. However, it is important to take into account any implicit parameters from an approach as well as the explicit ones. For example, we might claim that combining two values by averaging them is superior to combining them with a weighted sum because averaging requires no parameters, while a weighted sum requires two. Unless we have an *a priori* reason, however, for assuming that both parameters are equally important, we are simply hiding the weighting parameters by setting them to the value of $\frac{1}{2}$.

3.2 A Functional Decomposition

As stated before, supervised learning can be decomposed into the subtasks of data representation, data collection, feature representation, feature selection, and feature combination. Note that the functional independence of subtasks is an ideal which cannot always be met with all possible combinations of solutions to the subtasks. Possible dependencies between subtasks will be noted in the appropriate section.

3.2.1 Data Representation

The first issue at hand is what aspects of the data to provide to the learning algorithm, and in what format. Obviously if the information is not there, or is buried by large amounts of irrelevant information, then we cannot expect to get good performance. As an extreme example, a learning system for chess which gets its positions in the form of an index into some database, or as a list of move numbers from the start of the game is probably not going to be able to extract positional information.

A second question pertains to whether each position should be classified into the categories $\{Win, Loss, Draw\}$, or should be associated with a set of recommended moves. Kai-Fu Lee's Bill 3.0 [28] used the former scheme, as do most game-playing programs for the simple reason that most tree searching algorithms require a means of numerically rating the worth of positions, and any metric which indicates the probability of a position being part of the Win category will suffice.

Two notable examples of the second scheme are Gerald Tesauro's Neurogammon, [40] and Moriarty's board ranking for Othello [29]. Tesauro notes two variants on the move recommendation scheme. One can either structure the data so that for each position there is a recommended move, or in such a way that for a combination of position and move, a numerical score is returned indicating how good that move is. Neurogammon avoids the dicey problem of how to represent moves by implementing the latter. Its training database is composed of backgammon board configurations, each which contains a set of possible dice rolls, and for each dice roll, a set of possible moves using that roll along with a goodness score for playing that move. Tesauro justifies this approach because he, a backgammon expert, generates his database by hand, and such an approach coincides with the way that humans think about backgammon. Indeed, backgammon literature examples are primarily composed of hypothetical positions and dice rolls, along with a comparison of the best way to move men using that roll.

Moriarty used neural nets trained by genetic algorithms to rank moves in order to improve the performance of a standard alpha-beta search, since the alpha-beta algorithm works best if it searches the best branches of the game tree first. The output nodes of the neural net represented the possible moves in Othello, and the value at each node represented the goodness of that move. Thus, Moriarty also assigned scores to moves, as opposed to simply recommending a move.

In general, the position-based scheme can be transformed into a move-based scheme simply by assigning the score of the final position to the move which produced it. Thus, since the position-based scheme requires only that the position be codified, and not the potential moves, this is the method used in this thesis.

3.2.2 Data Collection

A problem in machine learning is where to get the training set of examples, and how to correctly classify them. Ideally we wish to obtain a representative sample of the set of all possible data, with each datum having a very high probability of being correct. Best of all would be a database which contains all possible data, correctly classified, but then we would have already solved the problem. Thus, we will settle for a database which is representative of the problem, and classified with reasonable accuracy.

Note that a raw percentage of all possible data may not be an adequate measure of how representative the database is, for two reasons. First, it is likely that some data will be more common than others. For example, at the start of the game there are likely to be a limited number of openings, so positions from the traditional lines of play are likely to appear in every game. The second reason is that while some positions are easy to classify, like a chess position where one side has an overwhelming material superiority, there are likely to be others which straddle the borderline between winning and losing. They may not form a large portion of all possible data, but they are important since play between two closely matched opponents will walk this grey area throughout the game.

The first possibility for data collection is to have a human generate the training examples, as Tesauro did. This works well for domains where there is a large body of human experience on the subject but few or no proficient computer programs. It also has the advantage of allowing the expert to decide which positions are the most representative and conducive to learning. The major disadvantages to this approach stem from the fact that humans are limited and fallible. Since humans are so slow compared to computers, it is difficult to scale up this approach, and tedious for the human. Also, even experts can make mistakes. Finally, using human generated training sets forces the implementor to incorporate domain-dependent knowledge in the system, since the human must decide how to classify each training position. An approach which can automatically generate and classify positions would correct all of these problems.

Lee's approach with Bill 3.0 was to use an older version of Bill, Bill 2.0, to generate the training set. He generated a game by selecting the first 20 moves of the game at random, and then let Bill 2.0 play against itself. He then classified each position occupied by the winning side as a winning position, and each position occupied by the losing side as a losing position. This approach takes advantage of the fact that if two perfect players played against each other, the player who started at a winning position will always occupy winning positions, and the one starting at a losing position will always occupy losing positions. While Bill 2.0 is not a perfect player, it was a world championship level program, and thus Lee felt justified in making the assumption of perfection.

Other approaches to generating test cases by allowing the program to play against itself can be seen in Samuel's polynomial evaluation learning [33], and Tesauro's application of Temporal Differencing [38]. In Samuel's work, two versions of a checkers program were set to play against each other. One version learned to play by comparing the result of its evaluation function against one generated by a deeper search, and adjusting its weights accordingly. As soon as it had reached a certain level of superiority over its non-learning opponent, it replaced the opponent, and all subsequent learning took place in the context of games against this most recent incarnation.

More recently, Tesauro has used Temporal Differencing to train neural net evaluation functions. In temporal differencing, the learning algorithm makes use of the difference between successive estimations of the value of a position as well as the eventual game outcome to adjust its weights appropriately. The amount of reliance on these estimation differences can be adjusted from 0 (weights are determined solely by the most recent difference) to 1 (weights are determined solely by the outcome of the game).

Both of these approaches have been used successfully in situations where it is impossible to generate perfect and complete information. This thesis does not deal with this problem specifically, and so to eliminate data collection as a potential source of errors, the only games studied are ones which can be solved completely. Both the training and testing set consist of all possible positions.

The game of tic-tac-toe is simple enough that it can be fully solved, and all possible positions stored in a small amount of storage. The checkers endgame databases [27] are part of the Chinook checkers program, currently the human-machine world champion. The game of checkers has been solved for all positions containing eight or fewer pieces. This database is divided into a large number of smaller databases, based on the number and type of pieces on the board, and the rank of the furthest advanced piece on each side. This provides a wide range of sizes and complexities for use as training sets.

3.2.3 Feature Representation

The subtasks of feature representation and selection move us out of the traditional position evaluation paradigms and into the field of feature discovery. Unlike hand-crafting features, the process of automatic feature discovery requires that we must supply some sort of alphabet out of which these features can be built. If the alphabet is too simple, the features will not be able to express those aspects of the board which allow it to be classified. If the alphabet is too complex, the feature selection process may have a difficult time gleaning the useful features from a sea of poor or illegal features.

At the simplest, we could use the same method to represent features as we used to represent boards, so that a set of features enumerates a set of boards. This scheme might be the best we can achieve if our data are all completely unrelated to one another, but usually we prefer to have each feature match a number of boards.

As we have seen, classifier systems encode the data as fixed-length binary strings. Introducing the # symbol to the alphabet for a string of length n means that we can represent 3^n of the 2^{2^n} possible subsets of strings of length n, or $(\frac{1}{2})^{2^n - n\log_2 3}$ of all possible subsets. However, we have previously noted that binary encodings can contain illegal strings, and introduce search biases. Other encodings are possible, such as using higher cardinality alphabets.

To illustrate the possibilities, consider the problem of encoding features for a tic-tac-toe board, ignoring for the moment the possibility of illegal boards, rotations, reflections, and so on. Using a binary encoding, we need two bits to store the 3 possible values, leaving one illegal value. If we take nine of these, we require 18 bits to fully store a board. Hence, there are $2^{2^{18}}$ possible sets of 18-bit strings, and the alphabet $\{1, 0, \#\}$ can encode 3^9 of them, or $(\frac{1}{2})^{262115}$ of them, an extremely small portion.

We can reduce the exponent somewhat by going to a higher cardinality alphabet. Using the alphabet $\{X, O, Space, \#\}$ we can represent 4^9 of the 2^{3^9} possible subsets of tic-tac-toe boards, for a fraction $(\frac{1}{2})^{19665}$. If we extend the alphabet to include symbols for *not–X*, *not–O*, and *not–Space*, we can now represent 7^9 subsets, for a fraction of $(\frac{1}{2})^{19658}$. It appears that a single feature will be unlikely to suffice to categorize a non-trivial domain, no matter how much we extend the alphabet. Fortunately, complete representative power can be gained if we use more than one feature. Going back to the simplest alphabet, we can see that it represents binary conjunctions. For example, if we number the bits of a string as $b_0, b_1, b_2, \ldots$, the string 10#1 represents the proposition $b_0\overline{b_1}b_3$. Using multiple classifiers gives us the logical-or operator, allowing us to construct propositions in disjunctive normal form, which is sufficient to represent all possible subsets of the strings we are trying to match, though not necessarily in the most efficient manner [24].

There are other extensions we could make, depending on domain-dependent knowledge. For example, position-invariant patterns might be useful, or patterns which take into account relationships between various pieces. Even before Tesauro added hand-crafted features to his neural net inputs, his notation was augmented to allow a simple representation of a point being "made", "stripped", "cleared", "slotted", and "broken" [40]. Research into automatic optimal feature representations presents another interesting challenge, above and beyond that of automatic feature discovery.

3.2.4 Feature Selection

What is a good feature?

Once we have an alphabet for representing features, we must generate a set of the best features. This entails deciding what "best" means for features, so we must define a metric for ranking features according to how well they would contribute to the solution. Ideally, we would use the metric defined for LS-1: the best features are those in the best-performing feature set. Unfortunately this requires resorting to LS-1 style feature selection, which requires a prohibitively large search space.

Approximations to this ideal feature metric fall into two categories: those that take into account the co-features of the feature we are measuring, and those that measure each feature in isolation. As an example of the first type of metric, we could measure a feature according to its best or average performance with a number of random co-feature sets. Another example of the first category, if we are using a form of population genetic algorithm, would be to take the rest of the population as the co-features, and average the population's performance over the time that the feature has been a part of it. This is the essence of the bucket-brigade algorithm in classifier systems [5], if we eliminate internal message passing. These sorts of systems have greater potential to approximate the ideal feature metric than measuring each feature in isolation. The first, however, requires a large amount of computation, and both can be unstable, as the value of each feature will fluctuate as its set of co-features changes.

The second category, measuring each feature in isolation, is likely to be simpler and more efficient than the first, since the worth of each feature does not depend on its co-features. There are several qualities which are desirable in a feature. First and foremost, it should be able to separate the data space into examples and non-examples of a given class. Above and beyond this, we would like it to match as many data as it can, and deviate as much as possible from the distribution we would expect if it were classifying data at random.

In this thesis, binary features are used, and a probabilistic approach is taken to measuring their worth. This has several advantages: probabilities are virtually parameter-free, intuitive to humans, and free from *ad hoc* judgements. They allow comparisons with other metrics, without the need to normalize one or both of the metrics. Finally, there is a large body of knowledge on how to combine and manipulate probabilities.

The following notation will be used throughout the rest of the thesis: A feature, or the event of a feature matching a datum, will be denoted by the letter $\mathbf{f}$. Data is divided into C disjoint classes, $\omega_1, \omega_2, \ldots, \omega_C$. The term ω_c will denote the cth class, or the event that a datum is a member of that class. Thus, the probability of a feature matching is $P(\mathbf{f})$, the probability of a datum being in the class ω_c is $P(\omega_c)$, the probability of a feature matching a datum given that the datum is in ω_c is $P(\mathbf{f}\,|\,\omega_c)$, and the probability that a datum is in ω_c given that a feature has matched it is $P(\omega_c\,|\,\mathbf{f})$. These probabilities can be easily estimated from the training set. An important theoretical relationship between these values is Bayes' rule [14], which states that

$$P(\omega_c\,|\,\mathbf{f}) = \frac{P(\mathbf{f}\,|\,\omega_c)P(\omega_c)}{P(\mathbf{f})}. \tag{3.1}$$

There are several different probabilistic ways in which one could measure the effectiveness of a feature. A feature which is very general will have a high probability of matching, so the generality of a feature can be determined by $P(\mathbf{f})$:

$$Generality(\mathbf{f}) \equiv P(\mathbf{f}).$$

A feature which tends to match the specific class ω_c will have a high $P(\omega_c\,|\,\mathbf{f})$, so we can use a metric like

$$Con(\mathbf{f}) \equiv \max_{\omega_c}\{P(\omega_c\,|\,\mathbf{f})\}$$

to measure how consistently that feature matches a particular class. In this thesis, we call this the *consistency* of a feature.

Another possible metric can be seen by noticing that the left hand side of equation 3.1 is changed most drastically towards 0 or 1 if $P(\mathbf{f} \,|\, \omega_c)$ is set to an extreme value, either 0 or 1. Thus, we can define the metric

$$Inf(\mathbf{f}) \equiv \max_{\omega_c}\{|P(\mathbf{f} \,|\, \omega_c) - \tfrac{1}{2}|\}.$$

In this thesis, this metric is called the *influence* of a feature.

An intuitive grasp for these concepts can be gained by looking at figure 4. The top diagram illustrates a feature with perfect Consistency. The second two diagrams show the two ways in which a feature can have perfect Influence.

Another feature metric can be deduced by looking at the distribution of classes that a feature matches. If a feature matches data at random, we expect to see the matched data distributed in the various classes according to the *a priori* probabilities of these classes. Any deviation from this expected distribution is evidence that the feature is displaying a preference towards certain classes.

This deviation can be measured by null hypothesis testing. Take the null hypothesis to be that the feature is matching data at random, according to the *a priori* probabilities of each class. Then the number of matches in each class follows a multinomial distribution. The χ^2 value of the deviation from this distribution can be estimated using Pearson's approximation [22]

$$\chi^2 = \sum_{c=1}^{C} \frac{(N^{\omega_c}_{actual} - N^{\omega_c}_{expected})^2}{N^{\omega_c}_{expected}},$$

where $N^{\omega_c}_{expected}$ is the expected number of data which match a feature and belong to category ω_c, and $N^{\omega_c}_{actual}$ is the number we actually find. If we let $N(\mathbf{f})$ be the number of data matched by feature $\mathbf{f}$, and $N(\mathbf{f} \wedge \omega_c)$ be the number of data matched by feature $\mathbf{f}$ which also belong to category ω_c, then the metric is

$$Dev(\mathbf{f}) \equiv \sum_{c=1}^{C} \frac{(N(\mathbf{f} \wedge \omega_c) - N(\mathbf{f})P(\omega_c))^2}{N(\mathbf{f})P(\omega_c)}.$$

This feature is known as the *distributional deviance*, or just deviance, in this thesis. The higher it is, the less likely it is that the feature matched the data at random.

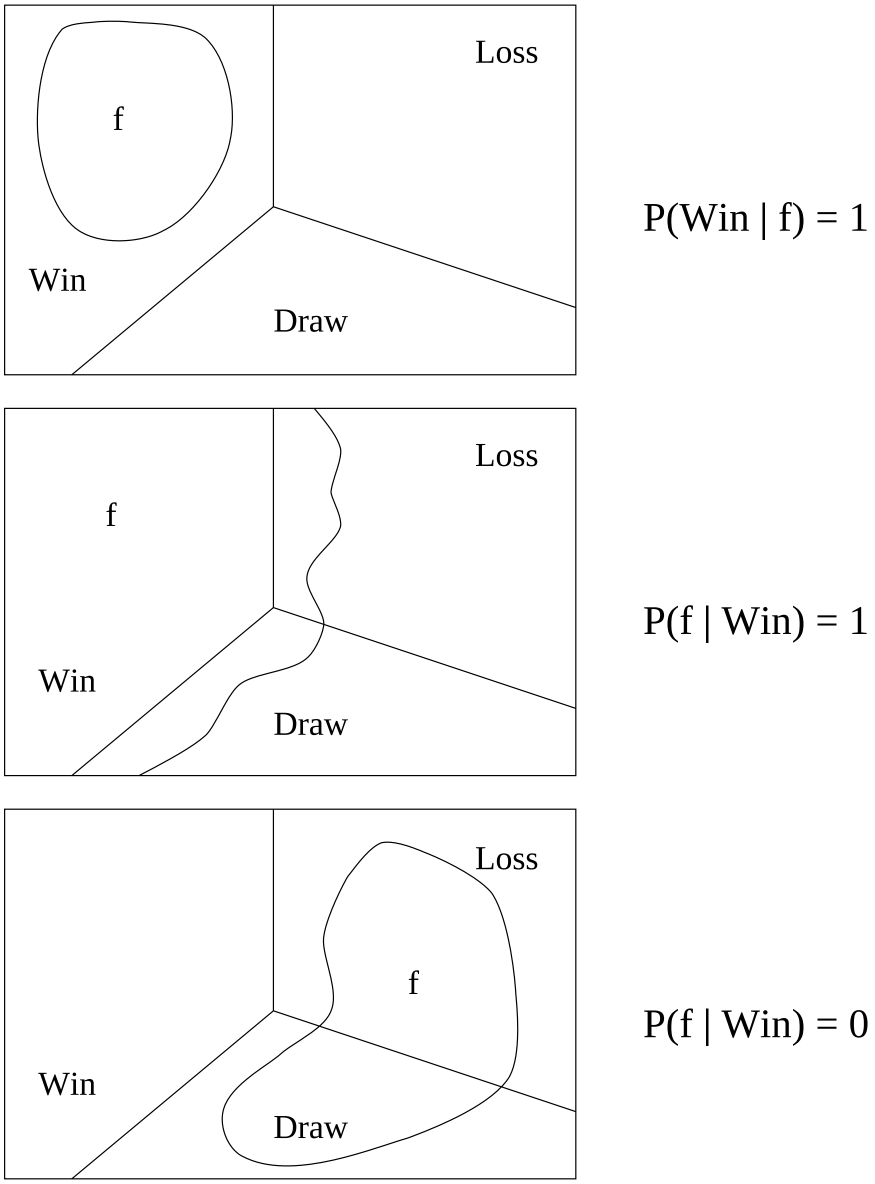


Figure 4: Optimal solutions to the consistency and influence metrics

It is important to note that statistical approximations of this sort usually come with conditions of validity. In this case, the results are only valid if the expected values are sufficiently large, say, at least 5 for 1 degree of freedom, and 10 for 2 or more degrees of freedom [22]. However, since we are using this test as a heuristic measure and not for formal hypothesis testing, the issue of sample size is less critical.

One final metric to consider would be the correlation between the event of a feature matching a datum and that datum belonging to a class. If we let X be a random variable which is 1 if the feature $\mathbf{f}$ matches, and 0 if it doesn't, and Ω be a random variable which is 1 if a datum is in the class ω_c and 0 if it is not, then we can use

$$Corr(X,\Omega) = \frac{Covariance(X,\Omega)}{\sqrt{Var(X)Var(\Omega)}}.$$

The variance of X is $NP(\mathbf{f})(1-P(\mathbf{f}))$, and similarly for Ω. The covariance between the X and Ω can be estimated by

$$Covariance(X,\Omega) = \frac{1}{N}\sum_{(X,\Omega)}(X-\overline{X})(\Omega-\overline{\Omega}) = P(\mathbf{f}\wedge\omega_c) - P(\mathbf{f})P(\omega_c).$$

Thus, we can define a *correlation* metric as follows:

$$Corr(\mathbf{f}) \equiv \max_{\omega_c}\left\{\frac{P(\mathbf{f}\wedge\omega_c) - P(\mathbf{f})P(\omega_c)}{\sqrt{N^2P(\mathbf{f})P(\overline{\mathbf{f}})P(\omega_c)P(\overline{\omega_c})}}\right\}.$$

Searching for good features

Goldberg, in his comparison of genetic algorithms with traditional search techniques [18], divides search methods into three main types: calculus-based, enumerative, and random. Calculus-based methods are typically exemplified by movement along the local error gradient in an effort to find a point which is better than all its neighbours. This is known in the literature as *hill-climbing*. Hill-climbing methods which employ movement along a gradient will fail if the fitness function is not differentiable at most of its points, since a gradient cannot then be determined. Many local optima in the the fitness landscape can also mislead the search. The fitness landscape over the space of all possible features as we have described it is likely to be very uneven, since the changing of even a single symbol in a successful feature can be enough to destroy its discriminatory properties. Thus, the fitness landscape is not even continuous, much less differentiable, and unlikely to submit to gradient-based optimization.

Enumerative techniques examine all states in turn, keeping track of the best fitness encountered. They have the advantage of being impossible to lead astray. If an optimal state exists in the landscape, enumeration will find it eventually. Unfortunately, since search time is on the order of the size of the search space, only trivial spaces can be searched in a reasonable amount of time.

However, enumeration can serve the extremely important role of a baseline technique for less robust but more efficient techniques. If we have several different search problems at our disposal, covering a range of search space sizes, then we can use an enumerative technique as a standard to tune the more efficient techniques in anticipation of applying these to the more difficult search problems. It is, of course, possible that there are qualitative differences between the simple and complex search spaces which invalidate this sort of approach, but this is beside the point. The purpose of baseline comparisons is to eliminate potential sources of error. An efficient search method may fail when applied to a large search space for two reasons: either it is a bad method, or it is bad for that particular search space. If an efficient search technique succeeds on a simple space, but fails on a more complex one, then we have eliminated the former cause and produced evidence for the latter. We can thus narrow the focus of our search for errors.

In this thesis, the PGAs are calibrated by comparing the patterns generated by them to those generated by enumeration. The enumeration technique is to generate all possible patterns and then sort them according to various fitness criteria. To break ties, a second criterion is used as a secondary sorting key. For tic-tac-toe and the smaller checkers databases, this approach is feasible.

If hill-climbing is too fragile and enumeration is too inefficient, we are left with the third possibility, random techniques. While there are other search methods which are guided by random values, such as simulated annealing, this thesis deals with genetic algorithms, which have already been shown to be more efficient than simple enumeration, and yet more robust than hill-climbing [18].

In particular, a rank-based genetic algorithm, similar to Darrell Whitley's GENITOR [41], is used. These algorithms sort the population according to fitness, and then use the *position* of an individual in the sorted list instead of that individual's fitness, to determine whether it will be selected for mating. Children are inserted at the appropriate position in the population if they are better than the worst individual, which causes the worst individual to be dropped off the end of the list.

The GENITOR algorithm has one parameter, called *Bias*, which determines how much reproductive advantage individuals at the top of the list enjoy. Bias has a lower bound of 1 (completely random selection), and as it is increased, selection pressure increasingly favours the top individuals. If we let β represent the bias, R be a random number from 0 to 1, N be the size of the population and i be the index of the individual chosen for mating, then the following formula is used by GENITOR to select individuals when $1 < \beta \leq 2$:

$$i = \left\lfloor N \frac{\beta - \sqrt{\beta^2 - 4(\beta - 1)R}}{2(\beta - 1)} \right\rfloor$$

which corresponds to a linear allocation of the number of offspring. The top-ranked individual produces, on average, β times as many offspring as the median individual. Thus, by modifying β, the convergence rate is under the direct control of the researcher.

While rank-based selection schemes throw away the information contained in the exact fitness value, these values are usually fairly *ad hoc*, so performance can actually be improved by considering rank as opposed to raw fitness. The scaling problem, for example, in which highly, but not optimally, fit individuals dominate the population, is eliminated. Certainly a rank-based approach fits more neatly with the enumeration approach discussed above. Furthermore, using a scheme which has only a single parameter follows the principle of eliminating as many parameters as possible. Finally, since a feature will never be eliminated from the population until the population is full of better individuals, rank-based populations have a stability not seen in many traditional genetic algorithms, where random genetic drift can accidentally drive even fit genotypes to extinction.

Since we are evolving a population of features, it is necessary not to have duplicate features in the population. Not only would duplicates decrease the population variation, but they interfere with the Bayesian combination function. The approach taken in this thesis is simple: the initial population does not contain duplicates, and they are disallowed from entering the population. In traditional genetic algorithm selection schemes, the chances of a feature being selected for mating would be determined both by its fitness and the number of copies it has in the population. By eliminating duplicates we capture reproductive advantage in one simple value.

A second departure in this thesis from traditional genetic algorithms is the use of variable-length genotypes, known as "messy" genetic algorithms [13]. All of the genetic encoding schemes in the next chapters are of the form:

$$(piece_1,\, square_1)(piece_2,\, square_2)\ldots$$

so a variable length genotype lends itself well to the representation. Overspecification, in which two elements conflict in what piece to place in a certain square, is resolved with a "first-come, first-serve" scheme. Underspecification, in which certain squares are not mentioned in the genotype, is handled by allowing unreferenced squares to match anything. Mating operators used are messy crossover, mutation, and inversion. Goldberg refers to this style of genetic encoding as *sparse encoding*, and mentions that it shows promise as a scheme for classifier systems [43].

3.2.5 Feature Combination

Once the features have been defined, we need some means to combine their values, also known as *conflict resolution*. Classifier system researchers have come up with several ways of resolving conflicts between output messages generated by classifiers. The simplest method is to accept the judgement of the classifier with the highest strength, but such an approach is deterministic, and so does not allow a variety of classifiers the chance to gain rewards or punishments. A stochastic version of the previous would randomly select a classifier, with the probability of being selected being proportional to the classifier's strength. Goldberg has recommended the *noisy auction* technique [43], in which each classifier has a certain amount of Gaussian noise added to its strength, and then the highest strength is selected.

All of the above methods have the disadvantage of assuming that if a set of rules conflict, one must be right and all the others wrong. Preferable would be a system in which each feature contributes to the final judgement according to its degree of surety. This is the approach of Bayesian learning, which has been successfully applied to a variety of tasks in computer vision [9], expert systems [8], and game playing [28, 7].

Bayesian learning relies on Bayes' Law, given in equation 3.1. This rule can be extended to the case of multiple binary features by introducing the vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, where x_i is set to 1 if $\mathbf{f}_i$ matches a given datum, and 0 if it does not match. Then, for a given datum:

$$P(\omega_c \,|\, \mathbf{x}) = \frac{P(\mathbf{x} \,|\, \omega_c) P(\omega_c)}{P(\mathbf{x})}. \tag{3.2}$$

The left-hand side is the probability that the datum is in the class ω_c on the basis of what features match or don't match. A judgement is arrived at by estimating the probabilities of a datum being in each class, and then simply selecting the class with the highest probability of being the correct class.

If the probability distribution of $\mathbf{x}$ is known, this equation can sometimes be used to calculate an optimal feature combination. For example, Lee used the fact that his feature value vector was approximately multivariate normal to generate the Bill 3.0 evaluation function for Othello [28]. Unfortunately, binary features do not lend themselves easily to a completely accurate probability determination, so we must approximate.

A common approximation technique, often seen in expert systems, is to assume that the features are independent given a class [14], ie.:

$$\forall \omega_c \; P(\mathbf{x} \,|\, \omega_c) = P(x_1 \,|\, \omega_c) P(x_2 \,|\, \omega_c) \ldots P(x_n \,|\, \omega_c).$$

This assumption greatly simplifies the calculation of the numerator of equation 3.2. A more stringent assumption, that the features are independent *in general* would be needed to simplify the denominator so that

$$P(\mathbf{x}) = P(x_1) P(x_2) \ldots P(x_n).$$

But, as Charniak points out, we can eliminate the denominator from this equation. Since we are simply ranking the outcomes of $P(\omega_c \,|\, \mathbf{x})$ for various ω_c and since $P(\mathbf{x})$ is the same for all of these values, we do not need to calculate it. This is fortunate, as the sheer fact that two features contribute evidence for a class is enough to produce some dependence between them [8].

Another simplification often made is to take the logarithms of both sides. This reduces the chances of arithmetic over- or underflow. Once all of these changes have been made to 3.2, we are left with the following set of discriminator functions:

$$g_{\omega_c}(\mathbf{x}) = \log P(\omega_c) + \sum_{i=1}^{n} \left[x_i \log p_i^{\omega_c} + (1 - x_i) \log(1 - p_i^{\omega_c}) \right] \qquad (3.3)$$

where $p_i^{\omega_c} = P(\mathbf{f}_i \,|\, \omega_c)$. The class ω_c whose discriminator function produces the highest value for any given $\mathbf{x}$ is deemed to be the correct class for the datum that produced the $\mathbf{x}$.

Notice that if we set n to be 0, that is, asked for a judgement without using any features, this algorithm would always select the class with the highest *a priori* probability. This agrees with our intuition that the best strategy to follow in the absence of any evidence whatsoever is to always pick the *a priori* most probable class.

These functions are linear in x_i, so interactions between features, manifested by statistical dependence, will tend to degrade performance. The degree of dependence between two features, given a class, can be measured by using the common statistical technique of null hypothesis testing using a 2-by-2 contingency table and a χ^2 test [22]. For the pair of features $\mathbf{f}_i$ and $\mathbf{f}_j$, and a class, we construct a table like so:

$$\begin{array}{|c|c|l} \hline a & b & \mathbf{f}_i \\ \hline c & d & \overline{\mathbf{f}_i} \\ \hline \multicolumn{1}{c}{\mathbf{f}_j} & \multicolumn{1}{c}{\overline{\mathbf{f}_j}} & \end{array}$$

where a, b, c, and d are the number of occurrences of each combination of presence and absence of $\mathbf{f}_i$ and $\mathbf{f}_j$, for a single class. Let the null hypothesis be that the features are independent of each other. If N is the total number of occurrences of that class, then we can use Pearson's test with Yates' correction for continuity [22] to arrive at the formula:

$$Chi(\mathbf{f}_i, \mathbf{f}_j) \equiv \chi^2 = \frac{N(|ad - bc| - N/2)^2}{(a+b)(c+d)(a+c)(b+d)}.$$

Since a class ω_c occurs $P(\omega_c)$ of the time, we can define a measure of total dependence between two features by using a weighted sum:

$$Dependence(\mathbf{f}_i, \mathbf{f}_j) \equiv \sum_{\omega_c} P(\omega_c) Chi(\mathbf{f}_i, \mathbf{f}_j). \tag{3.4}$$

If this value is greater than some predetermined cutoff, we judge two features to be too dependent.

There are a number of higher-degree polynomial approximations in existence, such as the Rademacher-Walsh, and Bahadur-Lazarsfeld approximations [14], which do not depend on the assumption of independence. They do, however, require a correspondingly high number of parameters, and hence data samples and time, to calculate. That is, a quadratic approximation using n features will require $O(n^2)$ parameters, samples, and units of time for an accurate approximation, which may be unacceptable if n is large.

Chapter 4

Experimental Results: Tic-Tac-Toe

To test these ideas, some experiments were done with tic-tac-toe and checkers. Although tic-tac-toe is quite a bit more simple than checkers or chess, it is a good domain for testing techniques, since all possible positions can be enumerated, perfectly classified, and analyzed in a reasonable amount of time. Furthermore, it turns out that even tic-tac-toe can be surprisingly complex.

All legal tic-tac-toe positions were used as training and testing data. These positions were obtained using exhaustive tree searching. Reflections and rotations were removed, and all boards were normalized so that X is to play. All positions were classified according to the eventual game outcome for X, assuming perfect play. The resulting database contains 765 positions, of which 51% are wins for X, 29% are losses, and 20% are draws.

Features were represented by a 9 symbol string from the alphabet $\{X, O, Space, \#\}$ where # is the "don't care" symbol which will match anything. The position of the symbol in the string corresponds to the tic-tac-toe board square, so that, for example, the string $X\#O\#O\#\#\#\#$ corresponds to the pattern

X	#	O
#	O	#
#	#	#

Features were combined using the Bayesian linear function defined in the previous chapter. Although there are $4^9 = 262,144$ possible combinations of the alphabet symbols, only 62,444 of them match at least one board.

4.1 Enumeration Results

To determine the theoretical best performance available to the PGA, an enumerative procedure was followed to create a population of the best possible features. All possible features which match at least one position in the database were generated, and then these features were sorted according to a particular feature metric. The top 200 were extracted and used to classify all the positions. Performance for the population was measured by the percentage of positions correctly classified.

In the cases of consistency and influence, it was found that far more than 200 of the possible patterns scored perfectly for that metric. In such cases, a second feature metric was used as a secondary sorting key. The sorting criteria tried were:

- Primary Key: Consistency
  Secondary Key: number of matches (Generality)

- Primary Key: Consistency
  Secondary Key: Deviance

- Primary Key: Influence
  Secondary Key: Deviance

- Primary Key: Deviance
  Secondary Key: number of matches

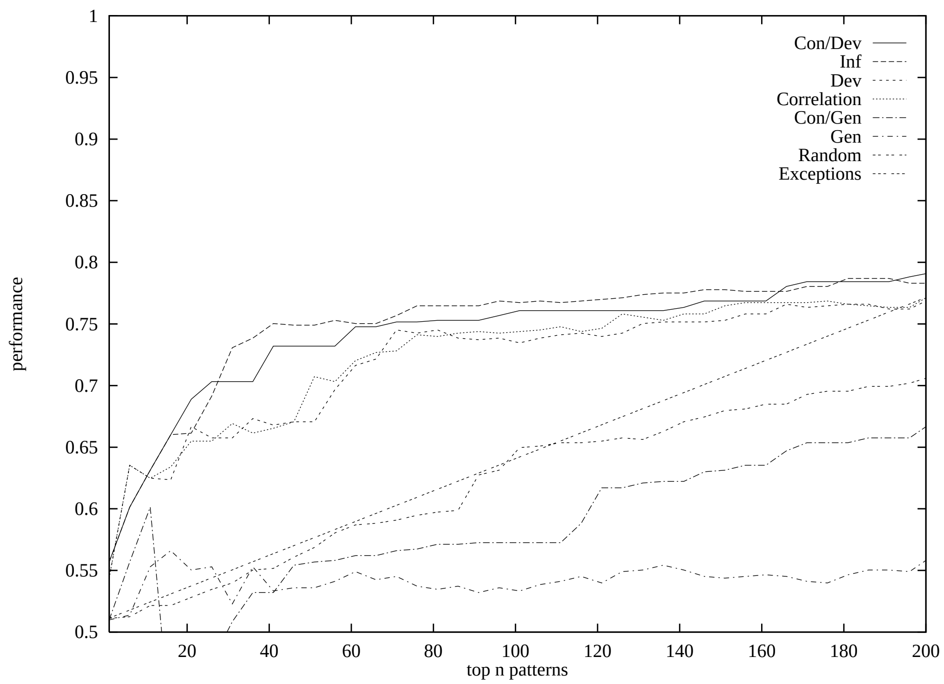


Figure 5: Performance of various feature metrics and baselines for tic-tac-toe

- Primary Key: Correlation
- Primary Key: Generality

Technically, the secondary key for deviance was unnecessary, as there is enough variation among the best high-deviance patterns to minimize the effect of a secondary sorting key.

In addition, two minimum-level baseline metrics were tested to estimate the lower bound on performance. The Random sorting criterion chooses 200 features at random without replacement from the population. The Exceptions criterion assumes the strategy of classifying the position according to the most *a priori* probable class, unless the position is found in a set of memorized exceptions. Thus, this approach amounts to rote learning. With n exceptions, for tic-tac-toe we expect a performance of $P(Win) + n/765$.

The performance of the top n features in the population is charted in figure 5 for various sorting criteria. That is, the performance when $x = 100$, for example, is the performance using the top 100 enumerated features. As can be seen, the performance for consistency/deviance, influence, deviance, and correlation initially rise quite rapidly, and then taper off, in some cases equaling that of rote learning. There does not seem to be much favouring one over the other, except that consistency and influence performances rise more quickly at the beginning. The two criteria in which generality plays a prominent part seem to do especially poorly, performing below even random features, suggesting that the sheer coverage of a feature is too simplistic to be useful. For the remainder of this thesis, the consistency criterion will refer to the consistency/deviance combination, not the consistency/generality one.

The fact that perfect performance is not achieved indicates that either tic-tac-toe is too complex a game to be captured using the number and types of features we are using, the feature combination function is not able to exploit all of the feature matching information, or a combination of the two. As we will see later, the feature combination function is responsible for much of the problem.

Initial experiments actually applied *a priori* minimum standards of deviance and number of matches to the features, so that features which did not meet these standards could not become part of the population. Figure 6 illustrates the performance of features in which the requirements $\{Deviance > 9.21,\ NumMatch > 50\}$ and $\{Deviance > 9.21,\ NumMatch > 25\}$ are applied to features. Since deviance is a χ^2 value, a value greater than 9.21 corresponds to the significance level $p < 0.01$ that the feature did not match at random. 50 matches is the lower bound on the number of samples required to have each expected frequency be greater than 10. 25 matches guarantees that this be greater than 5. Not only did these standards worsen performance, but the less stringent the requirements, the better the populations performed. This indicates that even features which only match a small number of positions contribute significantly to the performance.

Another approach used to improve performance was to change the combination function. Initially, it utilized both the positive evidence provided by a feature matching, and the negative evidence provided by a feature not matching. In the new combination function, known as the Active combination function, only the event of

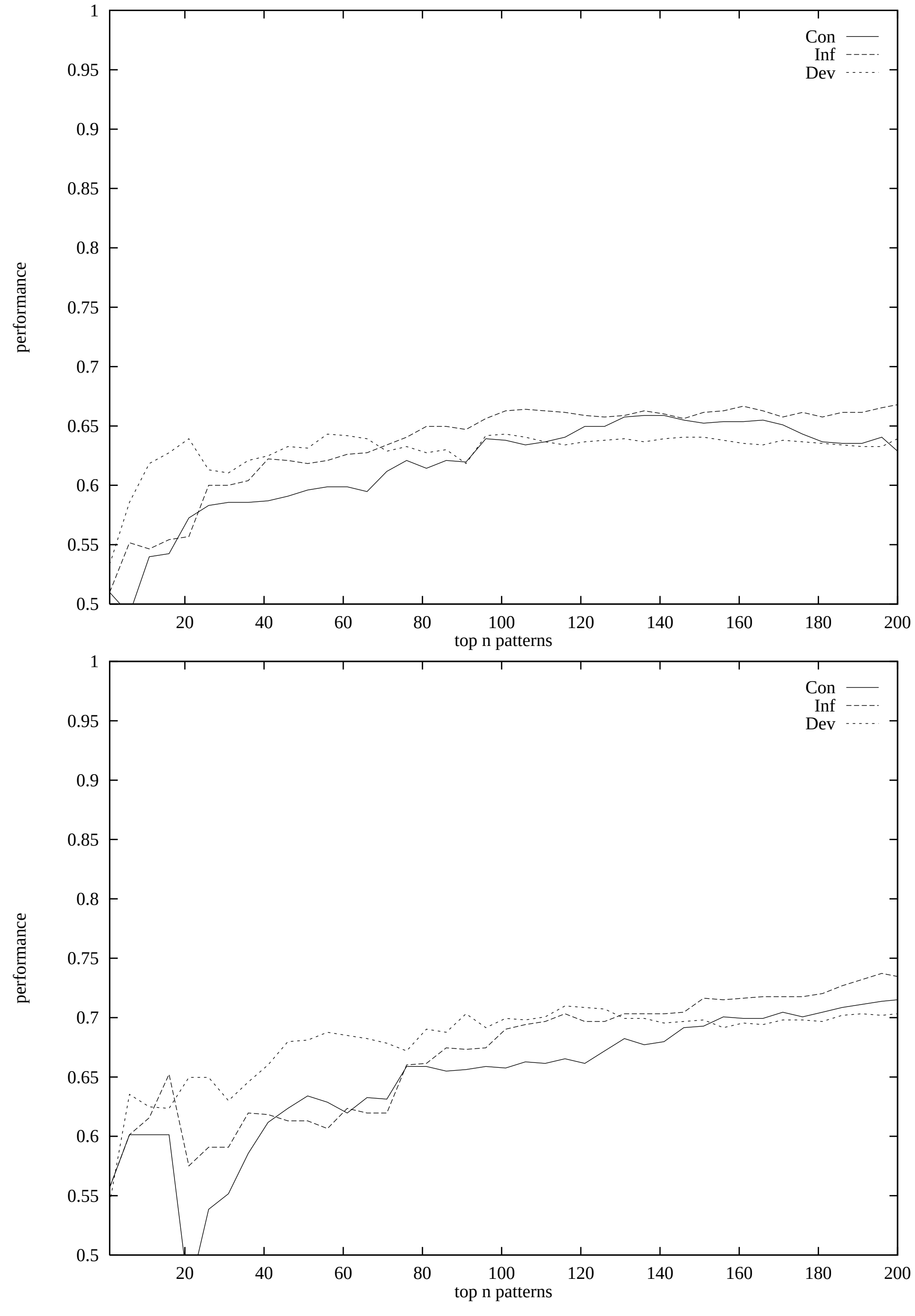


Figure 6: Effects of applying *a priori* standards to deviance and generality. The first diagram corresponds to requiring 50 matches. The second one, 25 matches. Both require deviance to be greater than 9.21

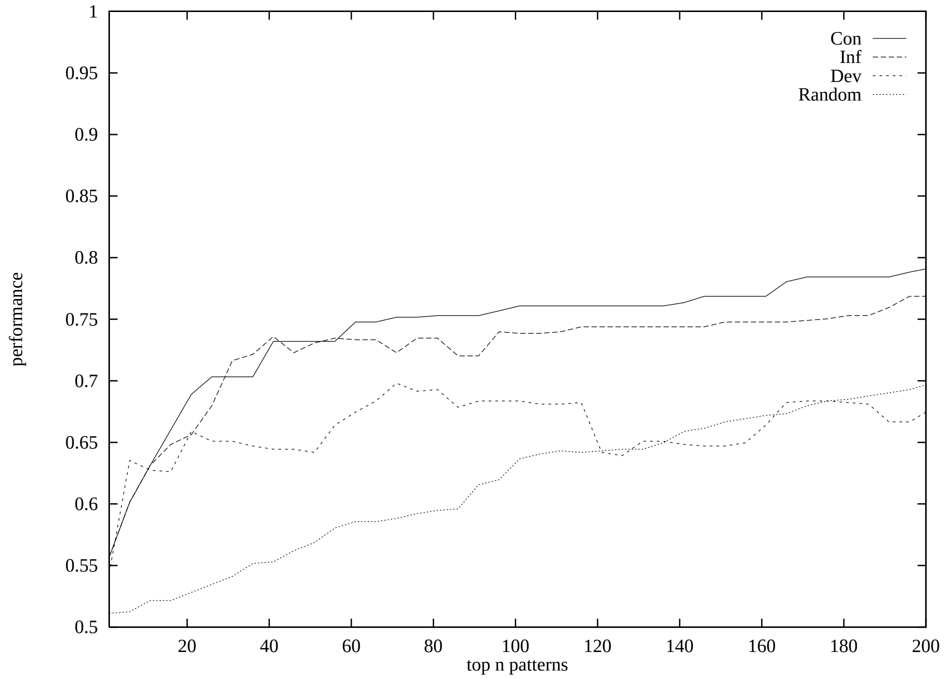


Figure 7: Effects of using an active combination function

a feature matching affected the outcome. If we remove all terms from equation 3.3 dealing with the case where a feature does not match (ie. $x_i = 0$), we arrive at the following set of discriminator functions:

$$g_{\omega_c}(\mathbf{x}) = \log P(\omega_c) + \sum_{i=1}^{n} x_i \log p_i^{\omega_c}.$$

As can be seen in figure 7, this did not affect the performance of the Consistency metric, and worsened the performance of Influence and Deviance.

One technique which did improve performance considerably was the removal of dependence from the population. Recall that our main assumption in determining a Bayesian combination function was that all features were independent, given a class. If this condition is violated, performance can be expected to degrade. In equation 3.4 we defined a metric to measure the dependence between two features. We can produce a measure of the dependence in a population by simply averaging the dependence of all pairs of features.

Dependence was removed from the population by changing the last step of the population generation process. Instead of taking the top 200 features from the sorted

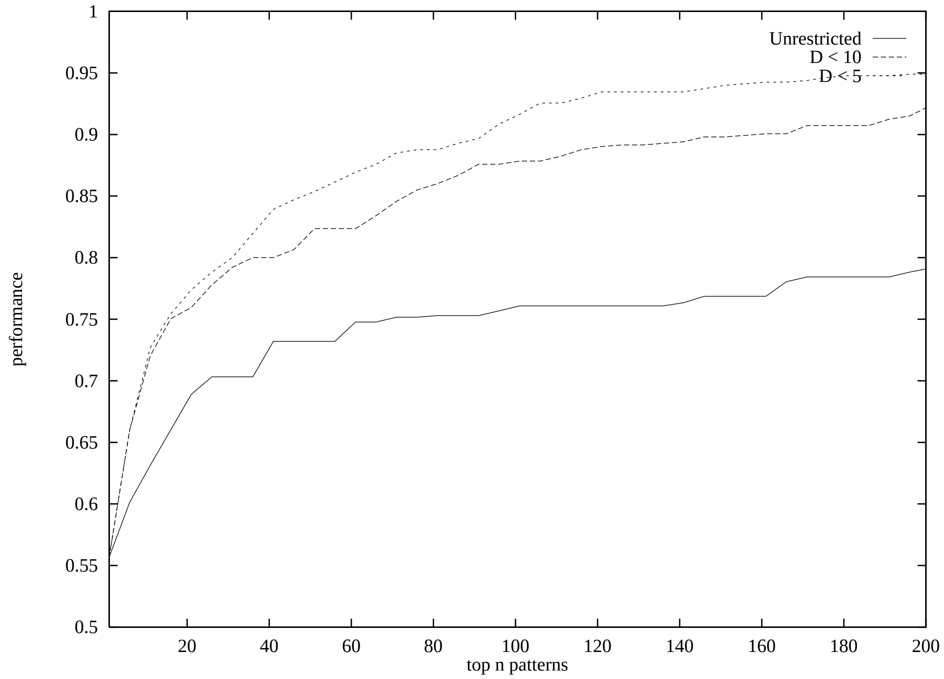


Figure 8: Dependence culling on the consistency population at levels 5 and 10

list, feature i was only allowed into the population if its dependence with each feature from $1 \ldots i-1$ was below a certain maximum, D_{max}. This approach is called dependence culling in this thesis. To generate 200 consistency patterns for $D_{max} < 10$, 1260 patterns were examined. For $D_{max} < 5$, 2911 patterns were examined. As can be seen from figure 8, dependence culling dramatically improves population performance.

To see the relationship between dependence and performance in a population, the average dependence versus relative performance can be plotted. Relative performance is the percentage of positions matched, but scaled so that the performance from 0 features is set to 0. In the case of tic-tac-toe, this would be the probability of the most probable class, ie. Win, so that we have

$$scaled\_perf(perf) = \frac{perf - P(Win)}{1 - P(Win)}.$$

The results show that there is a definite relationship between average population dependence and performance, and decreasing dependence increases performance. However, as the Generality data point indicates, dependency is not the sole criterion responsible for poor performance (figure 9).

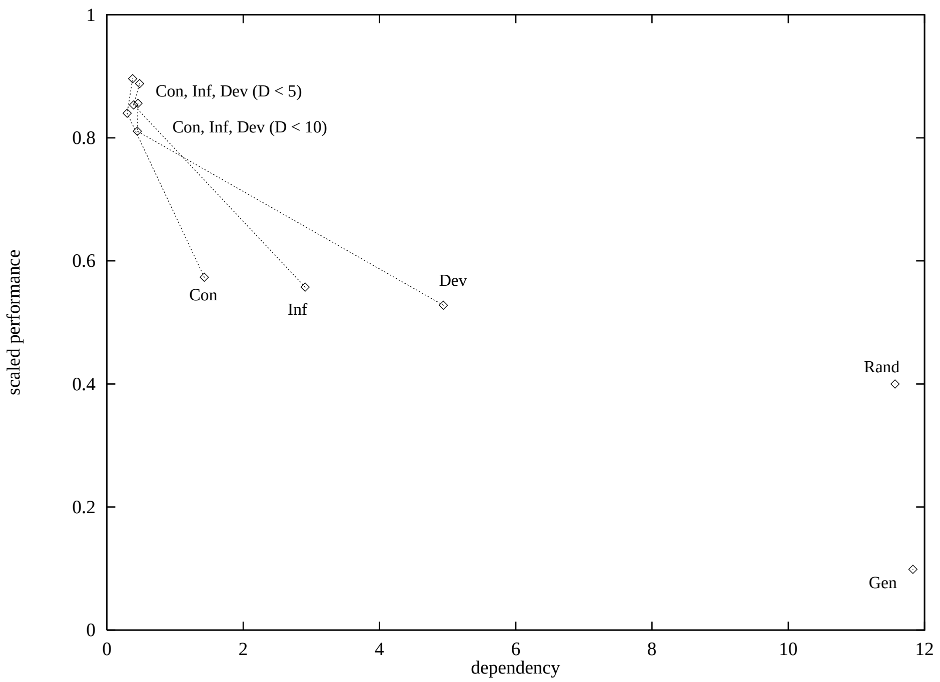


Figure 9: Population dependence vs. relative performance. Dotted lines illustrate the movement towards increased performance due to dependence culling

4.1.1 Qualitative Results

An examination of the best patterns for the best performing population shows a mixture of intuitive and not-so-intuitive features. Since all positions are normalized so that it is always X's turn to go, three O's in a row always signifies a loss for X. The various rotations of this pattern occupy the top two positions, as well as 4th and 7th place. Third place is two X's and a blank, corresponding to a win.

On the other hand, the enumeration process is as diligent about distinguishing draws as it is about distinguishing wins and losses. Several of the top 10 patterns are variations on this theme:

#	O	#
#	#	
#	X	O

which corresponds to a draw. Results such as this serve to illustrate that the difference between a human's and a computer's idea of what is interesting can often be quite pronounced.

4.2 Population Genetic Algorithm Results

As mentioned in the previous chapter, the genetic algorithm used in this thesis is rank-based, similar to Whitley's GENITOR program [41], which means that the decision of which individuals to select for mating is determined by their position in a list sorted by fitness. The ranking criteria used were exactly the same as those used for the enumeration results.

To maintain variety in the population and stave off convergence towards one individual only, duplicates were not allowed into the population. In domains which are continuous with respect to some parts of the genotype, convergence, even premature convergence, is still possible if a cluster of individuals which are different but similar and all lie about one of the optima is allowed to grow. Take, for example, the domain of real numbers, using a standard binary encoding. If there is an optimal solution at x, then we can generate a large number of near-optimal solutions simply by taking the binary representation of x and setting the least significant bits to random values.

However, our feature domain is sufficiently discontinuous that changing even one symbol is likely to produce a drastically different set of matches. Consider the difference, for example, between

O	#	#
#	O	#
#	#	O

and

O	#	#
#	X	#
#	#	O

The first corresponds very highly with a loss for X. The second represents nothing in particular and does not rank high on any of the metrics.

Individuals in the genetic algorithm population are varying-length strings built from a high-cardinality alphabet. Each element in the string consists of the tuple $(\{X, O, Space, \#\}, \{1 \ldots 9\})$. This string was decoded by scanning from left to right, and placing the appropriate piece in the appropriately numbered square. Conflicts over squares were resolved in a "first-come, first-serve" manner, and any squares not explicitly addressed were defined to contain #. Operations performed on these individuals were messy crossover, mutation, and inversion. Messy crossover and inversion boundaries took place between tuples only, and the mutation operator completely redefined a tuple.

To generate each new feature, two parents were selected by rank using a bias of 1.5, which corresponds to the top-ranked feature producing approximately 1.5 times as many children as the median feature, and then messy crossover was applied to combine them. The probability of mutation was set to 0.01 per tuple, and the probability of performing an inversion was set to 0.1. The populations contained 200 individuals, the genetic algorithms were run for 200 generations, and in each generation 200 children were produced and tested. This means that 40,000 features were examined during each run, which is 15% of the entire search space. Each PGA run was repeated five times to determine the amount of variation in performance. The results can be seen in figure 10. While the deviance population performances indicate that most of the enumeration patterns were learned, this is definitely not the case for the consistency patterns. The sudden decreases in performance of the consistency population are indicative of a set of high-dependence features which are not in the top 200, and consequently are not seen in enumerated populations.

It is important to note that the task of discovering the optimal features via PGA is more difficult for a two-metric sorting criterion like consistency/deviance, than for deviance alone. As we saw with the enumeration results, many features have a perfect consistency or influence, and so all the variation in the top 200 patterns comes from deviance. During the genetic algorithm runs for consistency or influence, features which had perfect scores for these metrics quickly took over the population. The task was thus reduced to one of optimizing deviance with the constraint that any children which had less than perfect consistency/influence were disallowed into the population. Clearly genetic algorithm techniques which handle constraints more gracefully, such

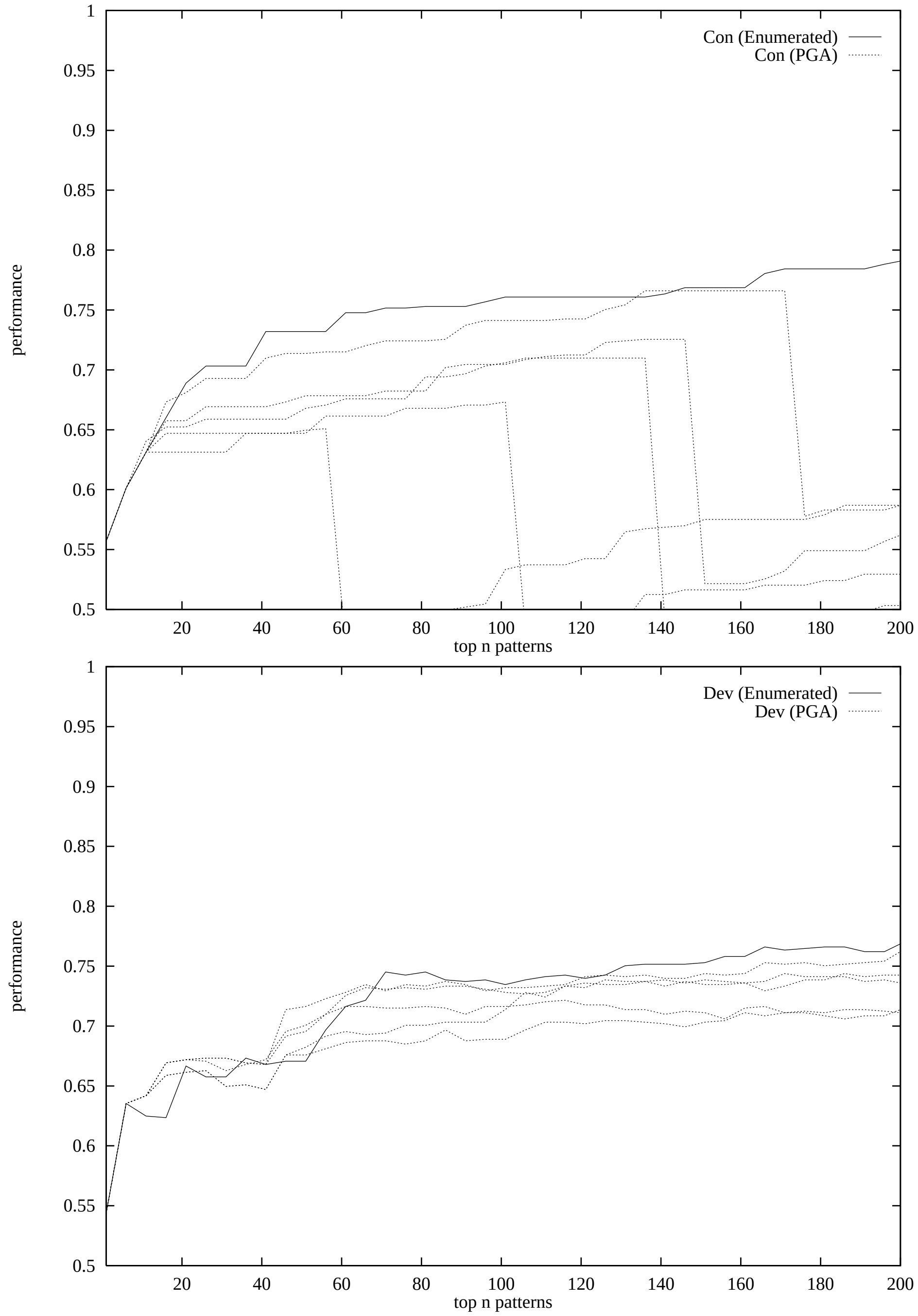


Figure 10: PGA and enumeration performance for consistency and deviance

as in [30], would be useful.

Implementing the equivalent of the dependence culling used in enumeration proved difficult. When enumerating the population, the greedy approach was taken, guaranteeing that all features would be independent, and that in case of conflict the highest-ranked feature took precedence. For rank-based genetic algorithms, guaranteeing the independence of all features is not so easy.

The algorithm used consists of two parts. First, the initial random population is created in such a way that an individual is not added to that population unless it is independent of all the individuals already added to the population. Then, when a child is created, the program creates a set of all individuals in the population which are too closely dependent on the child. If this set is empty, the child is added in the normal fashion. Otherwise, the child replaces the member of the set with the highest dependency, provided the child has a higher fitness.

The first part of this algorithm can produce problems because blocks of features can form which cover the dependency landscape in such a way as to make it difficult for any more features to be added. Thus, if 100 consecutive failed attempts were made to add random features to the initial population, then the rest of the features were added without the restriction of minimum dependence. The results of this dependence culling for consistency can be seen in figure 11. As with the enumerated population, performance improves substantially, but not to the level of the enumerated population.

In an attempt to expand the expressiveness of the features, an expanded alphabet was attempted, which contained the elements $\{X, O, Space, not{-}X, not{-}O, not{-}Space, \#\}$, where a not- symbol would match anything but that symbol. The search space for this approach becomes $7^9 = 40,353,607$, which is too large to enumerate, but well within the reach of the PGA. $40,000$ generated patterns represents 0.1% of the search space.

Unfortunately, expanding the alphabet did nothing to improve the population's performance, and performance was extremely erratic. An examination of the generated features explained why. With the expanded alphabet, synonymous features

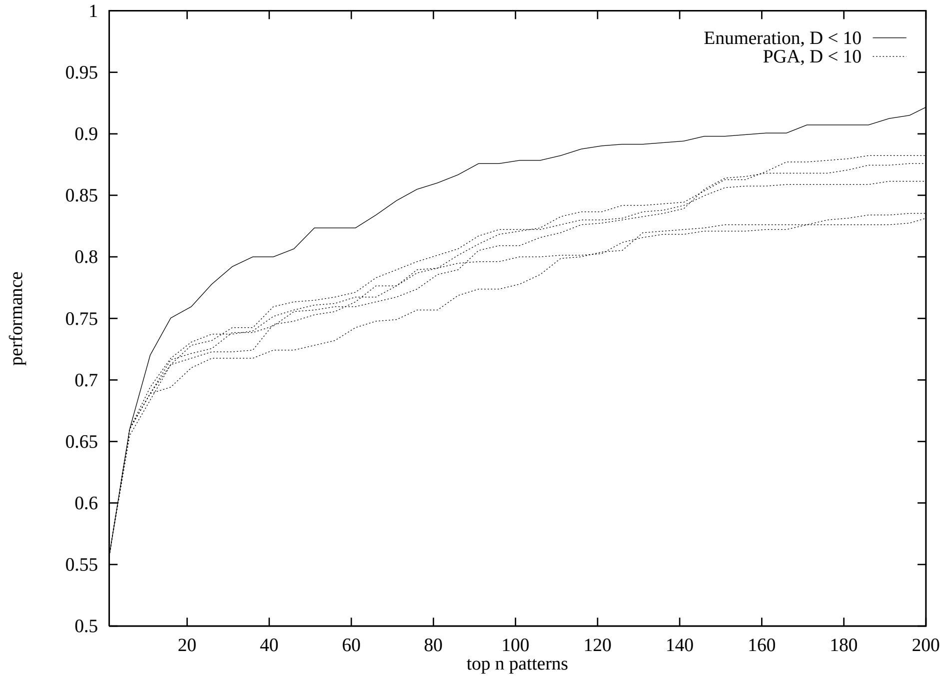


Figure 11: Effects of dependence culling on a consistency population: PGA and enumeration

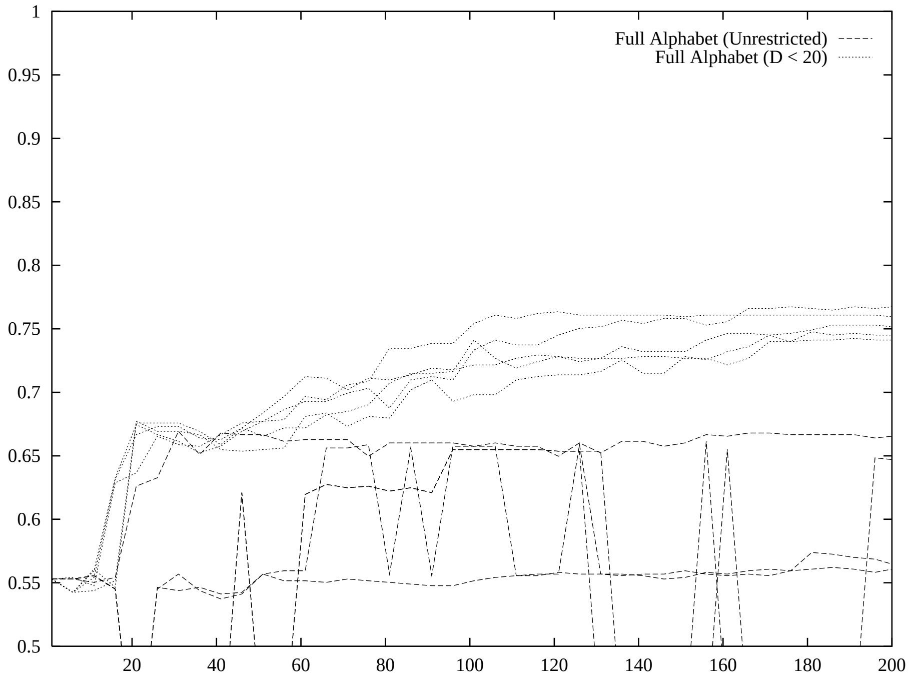


Figure 12: PGA performance using full-alphabet deviance features

became extremely common. That is, often two features would display superficial differences, but match exactly or nearly exactly the same set of positions. Thus, population dependence was extremely high. Dependence culling fixed the problem, and produced results similar to, but no better than, that of the smaller alphabet (figure 12). This suggests that improvements to the feature representation scheme must come from other directions.

4.3 Summary

An examination of various feature metrics indicated that the consistency, influence, deviance, and correlation metrics produced roughly the same performance, although consistency and influence produced better results when a limited number number of features were used. Feature metrics which used generality tended to do poorly.

Population performance was compromised by dependence between features in the population. The problem of too much dependence overwhelmed even the potential for better features from an expanded feature alphabet. Dependence can be eliminated somewhat from the population by the use of dependence culling.

PGA performance was close to that of the enumerated population, provided only one criterion was being optimized. Two-key fitnesses in which one key was perfect for many of the features caused problems for the PGA. A more sophisticated constraint satisfaction GA technique may solve this problem.

Despite the apparent simplicity of the game of tic-tac-toe, it appears that features constructed from simple template-like alphabets and combined using a linear Bayesian function are insufficient to completely categorize all positions. While this is a somewhat surprising conclusion, it is important to realize that no search was performed in the categorization. Humans regularly use a small amount of search in playing tic-tac-toe ("If I put an O here, then she can put her X there..."), and the ease in which tic-tac-toe yields to search has contributed to its reputation as a "trivial" game.

Chapter 5

Experimental Results: Checkers

The next domain attempted was a set of checkers endgame databases, generated for use by the Chinook checkers program [27]. Chinook is the current man-machine world champion, and its endgame database contains all positions, perfectly classified as winning, losing, or drawn, for eight pieces or less, a total of 440 billion positions. The database is divided into many smaller databases according to the number of each type of piece, and the most advanced checker for each side.

These smaller databases are named by six numbers: the number of black kings, the number of black checkers, the number of white kings, the number of white checkers, the rank of the farthest-advanced black checker, and that of the farthest-advanced white checker. Thus, a database called 1002.02 would contain all positions for one black king and two white checkers, where the the most advanced white checker is on the third rank, the ranks being numbered from 0 to 7.

The advantage of this scheme is that these subdatabases cover a large range of sizes and complexities, so that new techniques can be tried on the smaller databases before attempting the larger ones. The disadvantage of the subdatabases is that for the most part they are largely biased towards one class or other. Many are composed entirely of one class, and the remainder more often than not contain less than 10% of one class and more than 80% of another.

The three databases tested were 1002.02, 0022.60, and 1111.50. The rules of checkers mandate forced captures, and positions in which black's move is a forced capture are not contained in the databases, but calculated using a small amount of search. These positions were not included in the learning process since it would be difficult to classify them without performing some tactical analysis. The statistics for the three databases are:

Database	Size	% Win	% Loss	% Draw
1002.02	988	0	96	4
0022.60	612	36	22	42
1111.50	12,468	15	5	80

The composition of features is somewhat more sophisticated for checkers than for tic-tac-toe. They have the following structure:

$$(Binding, Anchor)(piece_1)(piece_2, \Delta row_2, \Delta column_2)(piece_3, \Delta row_3, \Delta column_3)\ldots$$

Each $piece_i$ is one of the set:

$$\{BlackKing, BlackChecker, BlackAny, WhiteKing, WhiteChecker, WhiteAny\}$$

where $BlackAny$ matches either black piece, and $WhiteAny$ matches either white piece. The Δrow and $\Delta column$ contain the position of that piece relative to $piece_1$. Thus we have a variable-length movable piece template.

The extent to which this template is allowed to move is determined by the $(Binding, Anchor)$ tuple, which determine the allowable squares for $piece_1$. $Binding$ is one of the set

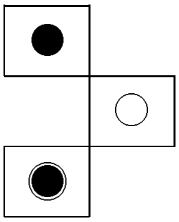
$$\{Absolute, Free, RowOnly, ColumnOnly\}$$

and the form of $Anchor$ depends on the value of $Binding$. If $Binding = Absolute$ then $piece_1$ must appear on one particular square, contained in $Anchor$, for the feature to match. $Binding = RowOnly$ or $ColumnOnly$ represents the restriction that $piece_1$ must appear on a given row or column contained in $Anchor$ for the feature to match. If $Binding = Free$ then there are no restrictions on the placement of $piece_1$ and $Anchor$ is ignored.

As an example, the feature

$$(RowOnly, 3)(BlackChecker)(WhiteChecker, -1, +1)(BlackKing, -2, +0)$$

corresponds to the following:



where the black checker is restricted to occur on the third rank only.

To eliminate exact synonyms from the population, each feature was accompanied by a 128 bit signature, which was a combination of hash values for each position that the feature matched. Thus, if two features had different signatures, they were guaranteed to match two different sets of positions. Since it is possible that two different sets of positions will generate the same signature, there was no guarantee that the reverse of this statement was true. However, in practice 128 bits proved sufficient to distinguish between various position sets.

5.1 Enumeration Results

The results of applying enumeration to determine the theoretical best 200 features can be seen in figures 13 to 15. In the case of the 1002.02 database, consistency gives the best performance. The influence and deviance criteria give identical populations, which perform marginally worse than consistency in achieving perfect performance. Once perfect performance is reached, the remaining features for all criteria become superfluous and cause dependency interference. An examination of the best consistency features reveals that for the most part they only correspond to one board. Thus, with the given encoding scheme there does not seem to be any better strategy than to enumerate exceptions to the loss class, which contains 96% of the positions to begin with.

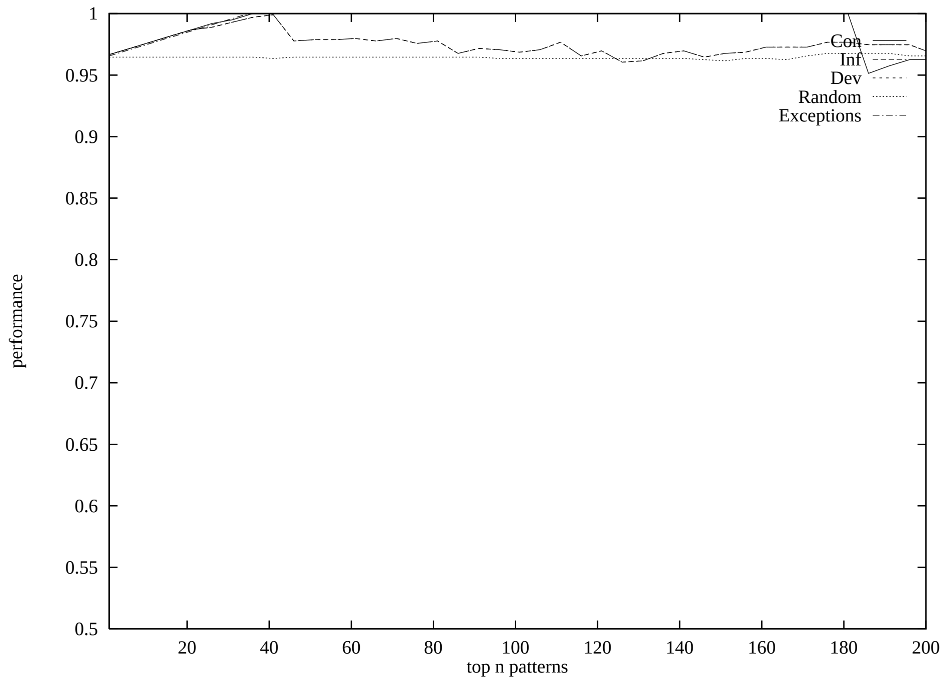


Figure 13: Performance of various metrics for the 1002.02 database

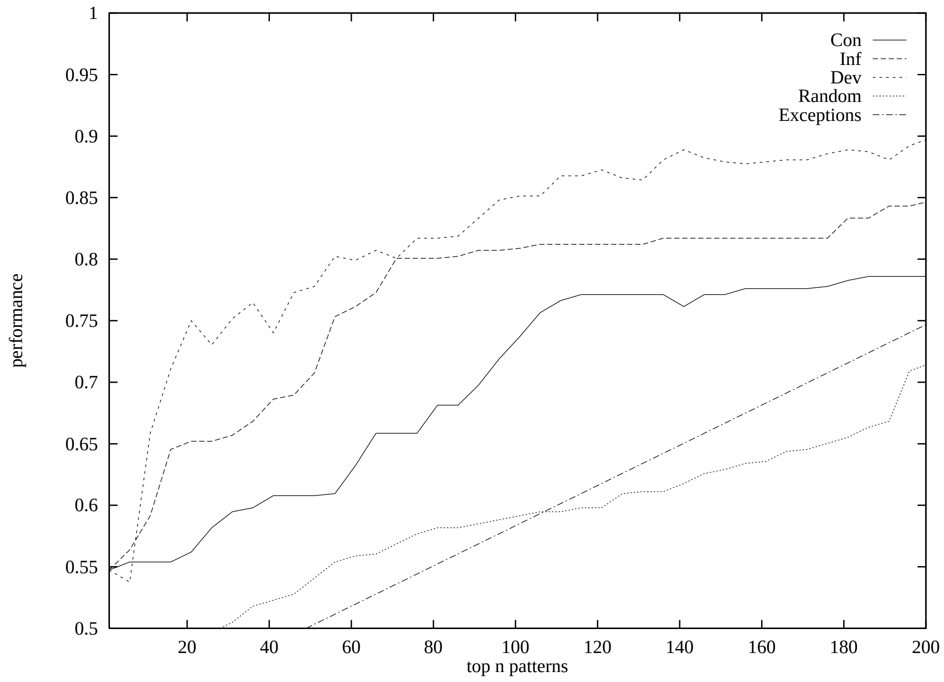


Figure 14: Performance of various metrics for the 0022.60 database

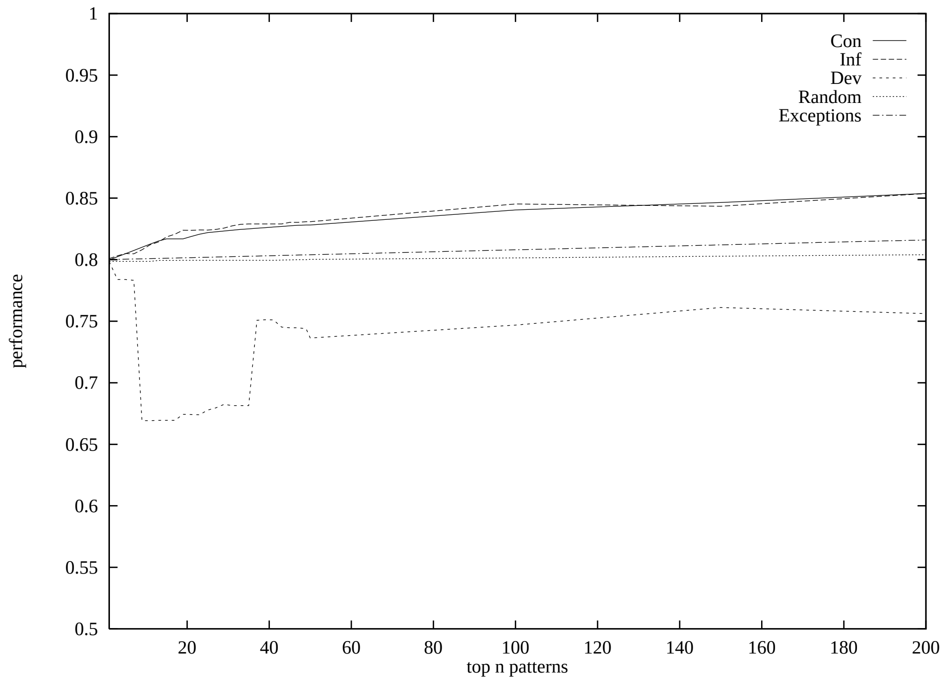


Figure 15: Performance of various metrics for the 1111.50 database

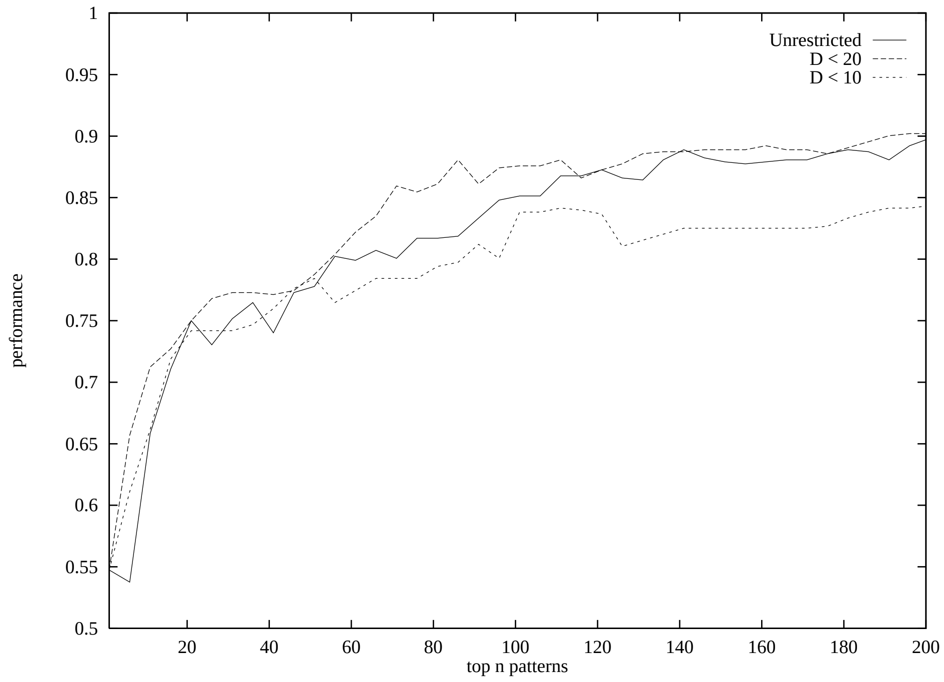


Figure 16: Effects of dependence culling for deviance on 0022.60

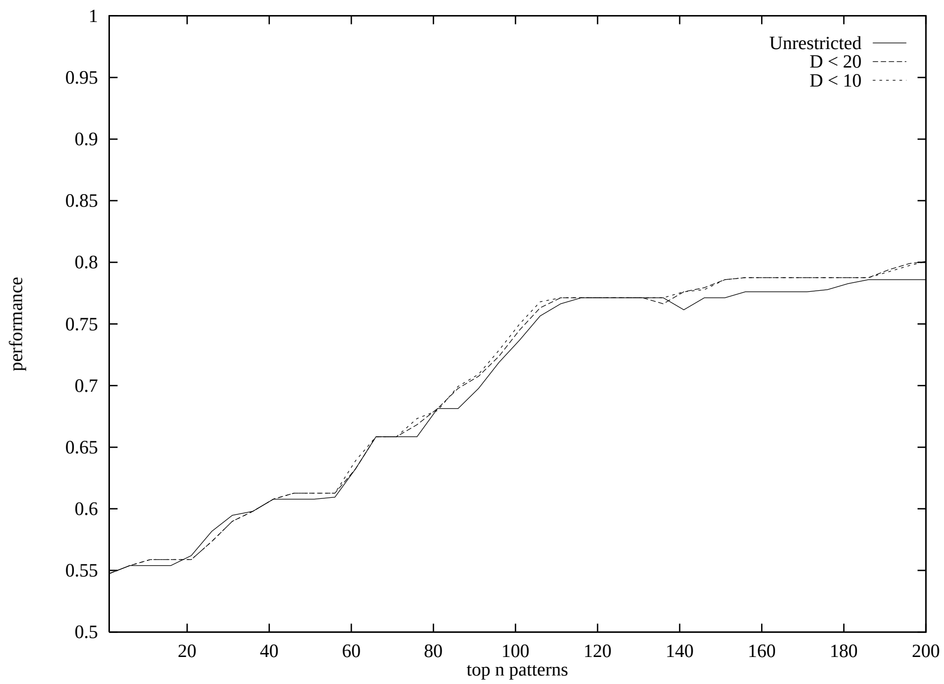


Figure 17: Effects of dependence culling for consistency on 0022.60

The results of the 0022.60 database are more interesting, and for the first time we have best performance coming from the deviance criterion. Dependence culling, on the other hand, does not improve the performance of any of the metrics, and even decreases the performance of the deviance population (figures 16 and 17) presumably due to the elimination of useful features. Unlike tic-tac-toe, dependence does not seem to be a big problem for the 0022.60 features.

In 1111.50, population dependence has degraded the deviance population's performance, and dependence culling on the population removes this deficiency (figure 18). The performance of even the consistency and influence populations, however, is somewhat disappointing, even after culling. To a certain extent, it is to be expected that it would be more difficult to compress 12,468 positions into 200 patterns than it would be to compress the 612 positions of the 0022.60 database. On the other hand, the current representation scheme is clearly inadequate for the job.

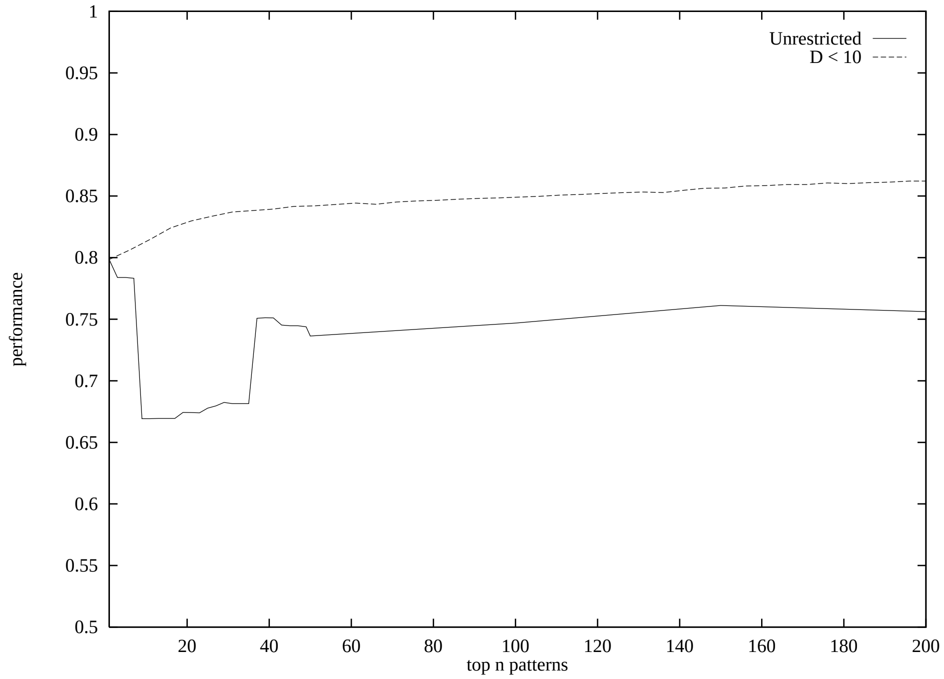
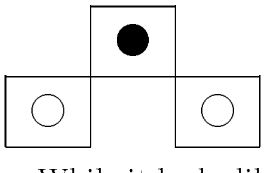


Figure 18: Effects of dependence culling for deviance on 1111.50

5.1.1 Qualitative Results

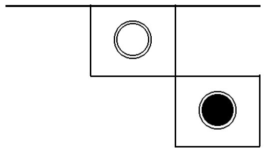
The features generated by enumeration on checkers databases were considerably less intuitive than those in tic-tac-toe, largely due to the fact that the classifications of checkers endgame positions can be difficult for humans to understand. Nevertheless, some interesting features could be determined.

For the 0022.60 database, for example, the most useful pattern was the following:

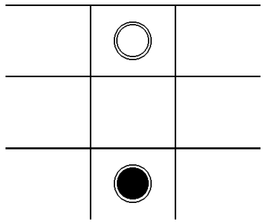


which was allowed to match freely. While it looks like black has the advantage (black moves down, black always has the next move), it is actually a loss for black. Since the 0022.60 database only contains white checkers on their home row, the pattern contains an implicit bottom edge of the board below the two white checkers. Black cannot jump either white checker, and cannot avoid getting jumped in the next turn. Whether this is considered a brilliant understanding of the problem domain or a shameless exploitation of a database artifact is left to the reader to conclude.

For the 1111.50 database, the top features are a group which contain variations on the following configuration:



where the white king must be on the top rank. This corresponds to a win for black because the black king can be moved like this:



which traps the white king and allows the black king to jump the white checker if necessary.

5.2 Population Genetic Algorithm Results

The genetic algorithm used for checkers was essentially the same as for tic-tac-toe, except that the bias was changed to 1.9. Experimenting with different biases indicated that since the population converged onto something reasonable regardless of the bias, the use of a strong selection pressure caused it to converge more quickly.

Individuals in the PGA were again composed of variable-length strings, now of either $(piece, row, column)$, or $(Binding)$. Note that tuples in the genome contain absolute row and column numbers instead of relative. In the process of decoding a genotype into a feature, the first tuple of the genotype contributed the *Anchor* values, and all subsequent pieces added to the feature were converted to be relative to this first piece. *Binding* was determined by the first *Binding*-type tuple encountered, and the rest were ignored. If no binding tuple was processed, then the binding was taken to be *Free*. The decoding process also kept track of the number of pieces of each type and stopped decoding any piece types that exceeded the maximum allowed by

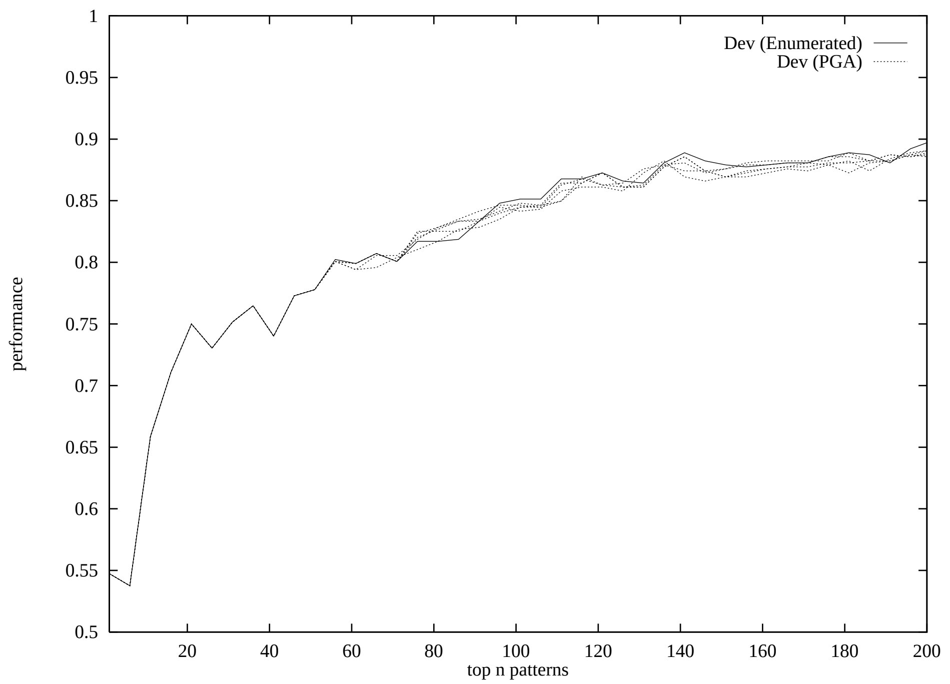
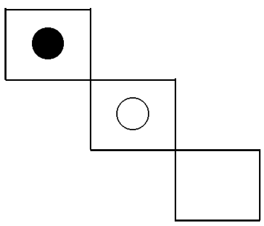


Figure 19: PGA and enumeration performance for deviance on 0022.60

the database. For a four-piece database, the number of possible legal patterns would be $4\ bindings \times 7^4\ templates \times 32\ squares = 307,328$ so that $40,000/307,328 = 13\%$ of the feature space was searched. The results for deviance on 0022.60 can be seen in figure 19. The PGA performance illustrates that most of the features were discovered.

In an effort to improve the expressiveness of the features, the feature set was expanded to allow one piece which matched an empty square to be added to the feature. The hope was that a feature like



which is clearly advantageous for black, would be found. Unfortunately it was discovered later that 0022.60 does not contain any of these situations, and in general any situation in which black has a forced capture had been eliminated from the database already. Empty squares were not originally part of the feature alphabet since the majority of squares in a checkers endgame are empty, and so most combinations of empty

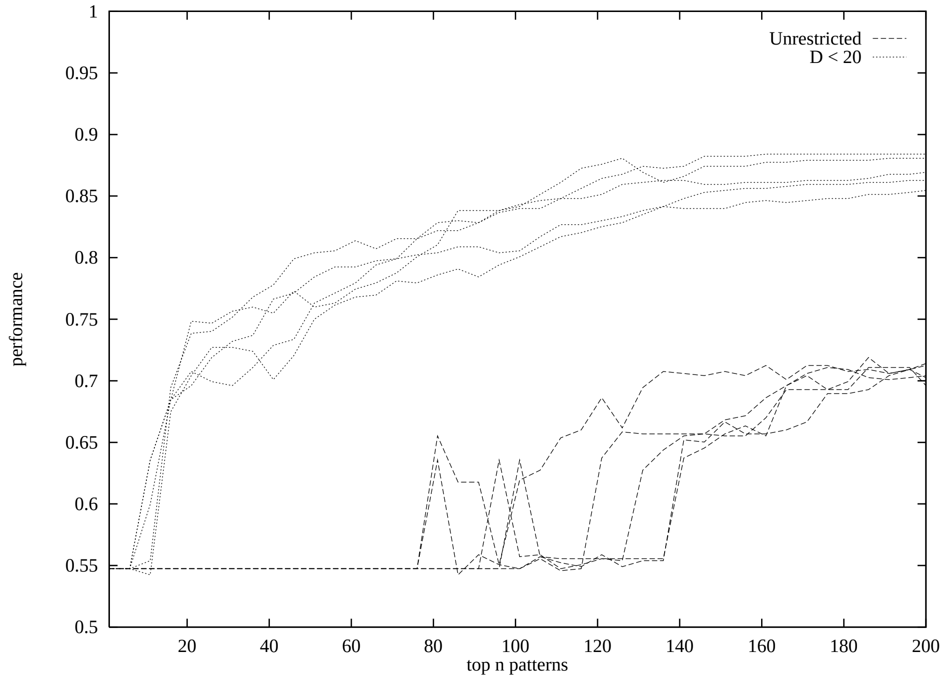


Figure 20: PGA performance when one empty square is added to features. Deviance metric on 0022.60

squares would turn out to be synonyms of each other, in the meantime expanding the search space by 2^{32-n} where n is the number of pieces on the board.

The problem of synonyms is manifested in the PGA results, even when the number of empty squares allowed in the feature was restricted to 1. The genetic algorithm simply took the best normal feature and tacked an empty square onto it at various positions to achieve more of the highly fit individuals. Adding dependence culling at level 20 improved the performance (figure 20), but not any more than could be attained from features which did not have the empty square.

5.3 Summary

One problem with many checkers endgame databases is their heavy bias towards one particular category. In cases such as these, it is usually most efficient to simply enumerate all exceptions to that category, which is the strategy employed by the enumerated patterns for 1002.02. For 0022.60, a small and well-balanced database, deviance turned out to be the best feature metric. For the larger 1111.50, the deviance metric produced features with large amounts of dependence, which were eliminated with dependence culling. This indicates that no one feature metric is best for all databases, and several should be tried. Perfect performance could only be attained with a heavily unbalanced database like 1002.02. In the case of 1111.50, only a moderate amount of performance could be achieved, which again reinforces our view of the inadequacy of the representation alphabet used.

The PGA was run using the deviance criterion for 0022.60, which produced results very close to those of enumeration. Expanding the feature alphabet to include the *Empty* symbol produced large amounts of dependency, which was eliminated from the population using dependence culling, with results comparable to that of more restricted alphabets.

It is difficult to predict how the PGA would perform on still larger databases. Since feature representation seems to be a problem, the amount of compression possible would depend on the amount of regularity in the database which can be captured by the feature alphabet. The general trend, however, seems to indicate that 200 features would produce poorer performance for larger databases.

Chapter 6

Final Remarks

6.1 Future Enhancements

6.1.1 Message Passing

In section 2.3 we simplified the classifier system to produce the population genetic algorithm. One interesting area of research would involve replacing components of the classifer system, one at a time, and observing how much this aids the performance of the PGA. For example, if we could reintroduce message passing into PGAs, their expressive power would increase substantially for the cost of a small increase in search space.

One possible approach would involve adding two more elements to the genotype: (*SendMessage* M) and (*ReceiveMessage* M). The former would add message M to a global message list whenever the whole feature matched, and the latter would require message M to be in the message list for the feature to match.

This implies that there would be some sort of temporal sequential ordering among the events of features matching. If the data of the database correspond to successive positions in, say, a chess game, then a natural scheme would involve using the message list to carry information from one position to the next. If the data follow no particular order in the database, then we can still employ a message list by starting with an empty message list for each datum, and then running through the feature set multiple times to allow different features in the set to send and receive messages for that datum.

The technique of Bayesian networks has already been developed to handle chains of features, and presumably the results would carry over to the domain of message passing in PGAs. If we assume that there are m possible messages, equation 3.2 can accommodate message passing extensions by introducing the binary vector $\mathbf{y}$ which has length $n+m$, so that $y_i = x_i$ for $1 \leq i \leq n$ and for $n+1 \leq i \leq n+m$, $y_i = 1$ if message $i-n+1$ is in the message list, and 0 if it is not.

The notions of consistency, influence, deviance, and the rest can be extended if we expand our notion of the feature matching a datum to include the event of the feature becoming involved in a chain of matches which culminates in a datum matching.

6.1.2 Meta-Features

One concept explored in this thesis involved determining the "goodness" of a feature by investigating its consistency, influence, and so on. The ultimate measure of the goodness of a feature was how it performed in the company of other features selected on the same basis. In this sense, we engaged in a second, higher-level classification scheme, in which the features became the data, and feature metrics like consistency became the meta-features. This suggests that, just as one feature was insufficient to adequately classify data, one feature metric may be insufficient to determine what constitutes a good feature, and so we should investigate methods of meta-feature combination.

One obvious approach would be to use a weighted combination of consistency, influence and deviance to rank features according to their usefulness. We thus come up against the same problems that confront researchers who wish to combine regular features.

Another approach is to employ a multiobjective genetic algorithm to optimize along all meta-features simultaneously. Pareto optimization [37] is a technique for optimizing several different fitnesses by distinguishing between dominated and non-dominated individuals. One individual dominates another individual if it is at least as good along every fitness criterion, and better for at least one criterion. Non-dominated individuals are those which are not dominated by any other individuals, and some recent work has been done to create genetic algorithms which will perform Pareto optimization [37].

It should be noted that the use of Pareto optimization for tic-tac-toe or checkers with either consistency or influence as a criterion would have produced the same results as obtained using the sorting approach of the thesis. Take, for example, the case of using consistency and deviance. Since there are many features with perfect consistency, all non-dominated individuals would have this quality, and so the non-dominated individual would be the one with the best deviance. If this was removed, the non-dominated individual would now be the one with the second-best deviance, and so on. Removing successive layers of non-domination would result in a population sorted, first according to consistency, and then according to deviance, as is done in this thesis.

6.1.3 Hybrid Schemes

A large population of binary features combined using a linear function will, by the central limit theorem, approximate a Gaussian distribution. Thus, many simple binary features can be treated as one large Gaussian one, and incorporated with other, more conventional features as an enhancement to them. As has already been noted, if the other conventional features are also Gaussian distributed, they can be combined optimally using an easily computable quadratic function.

Bayesian statistics are not the only possible method of combining binary features. Any linear combination of features can be modeled using a single-layer perceptron, so expanding the complexity of the perceptron to a multi-layer perceptron or Adaptive Logic Networks [2] might improve performance. Since ALNs take binary inputs and are much faster than multi-layer perceptrons, they are an especially attractive candidate.

6.2 Applications

6.2.1 Compression

One of the intended uses for the automatic feature discovery techniques of this thesis was to compress Chinook endgame databases. These contain 440 billion positions and occupy 6 Gigabytes of storage, so any reasonable compression scheme which allows for fast retrieval of a position/value pair would be extremely useful. A compressed database using PGAs would consist of a list of features and a set of weights for each feature and class. A set of misclassified positions along with their correct classifications could also be included, or the user could rely on search to uncover and correct any errors in classification. This type of compression could be used for other domains as well.

6.2.2 Game Playing

The obvious application of PGAs to game playing is in generating either part or all of a game-playing program's evaluation function. Performing a large number of pattern matches may prove too time consuming, however, to be practical for a tournament-style program. On the other hand, automatically generated populations of features are easier to read and understand than, for example, the pattern of weights in a neural net. Thus, PGAs can serve as a tool of analysis, either pointing out significant features which can be implemented more efficiently in a conventional manner, or pointing out similarities in positions misclassified by a conventional approach.

6.3 Conclusion

The purpose of this thesis was to provide a first cut at using genetic algorithms for automatic feature recognition. As such, it outlined the general problems to be solved and pointed out which areas in particular require further attention. The results indicate that for the practical application of PGAs to pattern classification, much remains to be done. On the other hand, the processes of functional decomposition and baseline comparisons more than proved their worth. It is common, when undertaking the task of applying genetic algorithms to a problem, to assume that the genetic algorithms will be the major source of error. Enumeration results on the games of tic-tac-toe and checkers indicated that most of the problems lay elsewhere, in the areas of feature representation and feature combination. This counter-intuitive conclusion would likely have been missed if the genetic algorithms were simply treated as a black box. Thus a greater understanding of the problem at hand was achieved.

Bibliography

[1] J. Antonisse. A new interpretation of schema notation that overturns the binary encoding constraint. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 86–91, George Mason University, June 1989.

[2] W. Armstrong and J. Gecsei. Adaptation algorithms for binary tree networks. *IEEE Trans. on Systems, Man and Cybernetics*, 9:276–285, 1979.

[3] J. Baker. Adaptive selection methods for genetic algorithms. In John Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pages 101–111, 1985.

[4] L.B. Booker. *Intelligent Behavior as an Adaptation to the Task Environment.* PhD thesis, University of Michigan, 1982.

[5] L.B. Booker, D.E. Goldberg, and J.H. Holland. Classifier systems and genetic algorithms. *Artificial Intelligence*, 40:235–282, 1989.

[6] R.A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, 1986.

[7] M. Buro. *Techniken für die Bewertung von Spielsituationen anhand von Beispielen.* PhD thesis, University of Paderborn, Paderborn, Germany, October 1994. In German.

[8] E. Charniak. The Bayesian basis of common sense medical diagnosis. In *Proceedings of the AAAI-83*, pages 70–73, Washington, D.C., August 1983.

[9] C.K. Chow. An optimum character recognition system using decision functions. *IRE Transactions on Elec. Comp.*, EC–6:247–254, December 1957.

[10] R.J. Collins. *Studies in Artificial Evolution.* PhD thesis, University of California at Los Angeles, 1975.

[11] K.A. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, University of Michigan, 1975.

[12] K. Deb and D.E. Goldberg. An investigation of niche and species formation in genetic function optimization. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 42–50, George Mason University, June 1989.

[13] K. Deb and D.E. Goldberg. mGA in C: A messy genetic algorithm in C. Technical Report 91008, IlliGAL, Department of General Engineering, University of Illinois at Urbana-Champaign, September 1991.

[14] R.O. Duda and P.E. Hart. *Pattern Classification and Scene Analysis.* John Wiley and Sons, Toronto, 1973.

[15] T.E. Fawcett and P.E. Utgoff. Automatic feature generation for problem solving systems. In Derek Sleeman and Peter Edwards, editors, *Machine Learning: Proceedings of the Ninth International Workshop (ML 92)*, San Mateo, California, 1992. Morgan Kaufmann.

[16] S. Forrest. Genetic algorithms: Principles of natural selection applied to computation. *Science*, 261(5123):872–878, August 1993.

[17] P.W. Frey. Algorithmic strategies for improving the performance of game-playing programs. In D. Farmer, A. Lapedes, N. Packard, and B. Wendroff, editors, *Evolution, Games, and Learning*, Amsterdam, 1986. North-Holland.

[18] D.E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning.* Addison-Wesley Publishing Company, Inc., New York, 1989.

[19] D.E. Goldberg. Zen and the art of genetic algorithms. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 80–85, George Mason University, June 1989.

[20] J.J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Trans. on Systems, Man, and Cybernetics*, 16(1):122–128, 1986.

[21] J.J. Grefenstette. Credit assignment in rule discovery systems based on genetic algorithms. *Machine Learning*, 3:225–245, 1988.

[22] W.L. Hays and R.L. Winkler. *Statistics: Probability, Inference and Decision.* Holt, Rinehart and Winston, Inc., New York, 1971.

[23] J.H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, Michigan, 1975.

[24] J. Horn, D.E. Goldberg, and K. Deb. Implicit niching in a learning classifier system: Nature's way. Technical Report 94001, IlliGAL, Department of General Engineering, University of Illinois at Urbana-Champaign, March 1994.

[25] D.H. Hubel and T.N Wiesel. Brain mechanisms of vision. *Scientific American*, 241:150–162, 1979.

[26] J.R. Koza. *Genetic Programming: On the Programming of Computers by means of Natural Selection.* MIT Press, Cambridge, Massachusetts, 1992.

[27] R. Lake, J. Schaeffer, and P. Lu. Solving large retrograde analysis problems using a network of workstations. In H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk, editors, *Advances in Computer Chess VII*, pages 135–162, University of Limburg, Maastricht, Netherlands, 1994.

[28] K.F. Lee and S. Mahajan. A pattern classification approach to evaluation function learning. *Artificial Intelligence*, 36:1–25, 1988.

[29] D.E. Moriarty and R. Miikkulainen. Evolving neural networks to focus minimax search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, volume 2, pages 1371–1377, Seattle, Washington, 1994.

[30] J.T. Richardson, M.R. Palmer, G. Liepins, and M. Hilliard. Some guidelines for genetic algorithms with penalty functions. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 191–197, George Mason University, June 1989.

[31] R.L. Riolo. Bucket brigade performance: II. Default hierarchies. In *Genetic Algorithms and Their Applications: Proceedings of the Second International Conferance on Genetic Algorithms*, pages 196–201, Hillsdale, New Jersey, 1987.

[32] R.L. Riolo. CFS-C: A package of domain independent subroutines for implementing classifier systems in arbitrary, user-defined, environments. Technical report, Logic of Computers Group, Divison of Computer Science and Engineering, University of Michigan, November 1988.

[33] A.L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal*, 3:210–229, 1959.

[34] A.L. Samuel. Some studies in machine learning using the game of checkers, II. *IBM Journal*, 11:601–617, 1967.

[35] D. Schuurmans and J. Schaeffer. Representational difficulties with classifier systems. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 328–333, George Mason University, June 1989.

[36] S.F. Smith. *A Learning System Based on Genetic Adaptive Algorithms*. PhD thesis, University of Pittsburgh, 1980.

[37] N. Srinivas and K. Deb. Multiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248, 1994.

[38] G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.

[39] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, March 1995.

[40] G. Tesauro and T.J. Sejnowski. A parallel network that learns to play backgammon. *Artificial Intelligence*, 39:357–390, 1989.

[41] D. Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–123, George Mason University, June 1989.

[42] D. Whitley and T. Hanson. Optimizing neural networks using faster, more accurate genetic search. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 391–398, George Mason University, June 1989.

[43] S.W. Wilson and D.E. Goldberg. A critical review of classifier systems. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 244–255, George Mason University, June 1989.