A grammar-based technique for genetic search and optimization

Clayton Matthew Johnson
College of William & Mary - Arts & Sciences

A GRAMMAR-BASED TECHNIQUE FOR GENETIC SEARCH AND OPTIMIZATION

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

by

Clayton Matthew Johnson

1996

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

[illegible]

Clayton M. Johnson

Approved, August 30, 1996

[illegible]

Stefan Feyock
Thesis Advisor

[illegible]

William Bynum

[illegible]

Phil Kearns

[illegible]

Rex Kincaid
Department of Mathematics

[illegible]

Gene Tracy
Department of Physics

In memory of my father,

Thomas H. Johnson

In memory of my sister,

Kimberly Marie Day

Contents

ACKNOWLEDGMENTS

To my friends: James O'Hara, for his passion; Dennis Pattinson, for his acceptance; and Dave Piller, for his wit. Thanks for the Humanities lessons.

To my fellow graduate students, especially: Tracy Camp, for never allowing me to think too little of myself; Tracey Beauchat, for never allowing me think too much of myself, either; Phil Auld, for good conversation and an outlet for play; Felipe Perrone, for long hours contemplating the insignificant; A.B. Wakely, for keeping my head in working order; Stamos Karamouzis, for the sibling rivalry; Rance Necaise, for serving as foreman of the moving and storage crew; Tin Siladin, for offering a safe harbor during the storm; and Nicky Albanese for his continual efforts to distract me from my work.

To the faculty and staff of the Computer Science department at the College, who made allowances for the non-traditional. Additional thanks to a top-notch dissertation committee: Bill Bynum, Phil Kearns, Rex Kincaid, and Gene Tracy. Special thanks to my advisor, Stefan Feyock, for his patience, breadth of knowledge, and subtle humor in directing my research over the years.

To the Williamsburg Players, for the experience of Theater.

To my family, especially Leanne Cain and Rick Day. Loving thanks to my mother, Nancy Johnson, for her emotional and financial support during the lean times.

List of Figures

ABSTRACT

The genetic algorithm (GA) is a robust search technique which has been theoretically and empirically proven to provide efficient search for a variety of problems. Due largely to the semantic and expressive limitations of adopting a bitstring representation, however, the traditional GA has not found wide acceptance in the Artificial Intelligence community. In addition, binary chromosomes can unevenly weight genetic search, reduce the effectiveness of recombination operators, make it difficult to solve problems whose solution schemata are of high order and defining length, and hinder new schema discovery in cases where chromosome-wide changes are required.

The research presented in this dissertation describes a grammar-based approach to genetic algorithms. Under this new paradigm, all members of the population are strings produced by a problem-specific grammar. Since any structure which can be expressed in Backus-Naur Form can thus be manipulated by genetic operators, a grammar-based GA strategy provides a consistent methodology for handling any population structure expressible in terms of a context-free grammar.

In order to lend theoretical support to the development of the syntactic GA, the concept of a trace schema - a similarity template for matching the derivation traces of grammar-defined rules - was introduced. An analysis of the manner in which a grammar-based GA operates yielded a Trace Schema Theorem for rule processing, which states that above-average trace schemata containing relatively few non-terminal productions are sampled with increasing frequency by syntactic genetic search. Schemata thus serve as the "building blocks" in the construction of the complex rule structures manipulated by syntactic GAs.

As part of the research presented in this dissertation, the GEnetic Rule Discovery System (GERDS) implementation of the grammar-based GA was developed. A comparison between the performance of GERDS and the traditional GA showed that the class of problems solvable by a syntactic GA is a superset of the class solvable by its binary counterpart, and that the added expressiveness greatly facilitates the representation of GA problems. To strengthen that conclusion, several experiments encompassing diverse domains were performed with favorable results.

A GRAMMAR-BASED TECHNIQUE FOR GENETIC SEARCH AND OPTIMIZATION

Chapter 1

Introduction

The evolutionary process is the only naturally occurring adaptive search algorithm known to exist. In nature, every organism is uniquely defined by the structure of its DNA. If each individual is considered to be a point within the search space of genetic organization, the power of "evolutionary search" becomes readily apparent. For example, a single chromosome in even the simplest vertebrate organism contains tens of thousands of genes, each of which can take on one of four nucleotide base values. This corresponds to over 10^{6000} different gene combinations. The extraordinary complexity of the biological structures discovered is evidence of the system's incredible search capabilities.

The manner in which genetic search operates was initially investigated by Charles Darwin[51]. In his treatise, *On the Origin of Species by Means of Natural Selection*, Darwin introduced the concept of "survival of the fittest". Each individual differs from other organisms in a species by slight variations in genetic structure. These variations cause physical and behavioral differences, and directly affect how the individual interacts with its environment. If an organism is better suited to its environment, its chances of surviving and eventually reproducing are greater. In this manner nature continually attempts to adaptively discover optimal genetic solutions.

John Holland[118, 119, 120] was first to explore the computational nature of the evolutionary process. In *Adaptation in Natural and Artificial Systems*, Holland developed a basic methodology for viewing all adaptive systems. He also presented a procedure for solving arbitrary search problems using natural selection as a model. This type of solution strategy is known as a **genetic algorithm.**

In following the evolutionary paradigm, the genetic algorithm (GA) operates on a population of structures, each of which represents one potential solution to the stated problem. The effectiveness of each individual is determined by performing an artificial "natural selection". Highly fit individuals are then mated and allowed to reproduce by combining their structures to create offspring for the successive generation. This process is repeated iteratively until the population converges toward a single set of solutions.

In the past two decades the genetic algorithm has been successfully applied to a wide variety of problems in optimization, classification, and machine learning; however, GAs have not yet made the breakthrough into mainstream Artificial Intelligence (AI). The main reason for this discrepancy is that genetic algorithmists have typically adopted a fixed-length binary string representation in their research. The larger AI community, on the other hand, has relied heavily upon more expressive semantic representations describing complex interrelationships between problem elements.

In the past few years there have been several proposals for modifying the representation scheme of genetic algorithms. Research efforts by Smith[234] and Goldberg, Korb and Deb[87] presented techniques for applying the genetic algorithm to variable length strings. Fujiki and Dickinson[83] developed genetic operators for Lisp S-expressions. Bickel and Bickel[24] and Koza[149, 152] devised similar operators for tree structures. Antonisse and

Keller[8, 7] and Grefenstette[108, 110] also presented methods for performing genetic search on certain higher-level statements.

These efforts have provided useful extensions to the genetic algorithm paradigm; however, each focuses on adapting the GA to only one specific type of structure. None of this work has presented a general-purpose strategy for performing genetic operations on an arbitrary class of expressions. Since GAs are designed to be domain-independent procedures, this lack of an underlying methodology has greatly reduced the applicability of genetic algorithms to problems expressed in conventional AI terms.

The research presented in this dissertation describes a grammar-based approach to genetic algorithms. With this technique, all structures in the GA population are strings derivable from a problem-specific grammar. Any structure which can be expressed in Backus-Naur Form (BNF), whether it be a fixed-length binary string or a complex computer program, can therefore be manipulated by genetic operators. As such, a grammar-based GA strategy provides a consistent methodology for handling arbitrary populations of structural elements.

Chapter 2 of this dissertation provides an introduction to the genetic algorithm and to schema theory. It also details the early development of the genetic paradigm. Chapter 3 presents the grammar-based genetic approach, and describes related work in GA representation. The **GE**netic **R**ule **D**iscovery **S**ystem (**GERDS**) for performing GA tasks is also detailed. Chapter 4 develops a new schema theory for syntactic GAs, and applies the grammar-based genetic algorithm to several optimization tasks. Chapters 5 presents applications of GERDS to representative problems in classification and machine learning. Finally, Chapter 6 explores the use of a grammar-based approach for the discovery of meta-rule heuristic operators, and details directions for future research.

Chapter 2

Background

2.1 The Genetic Algorithm

The genetic algorithm (GA) is a powerful search strategy based on natural selection and population genetics. In the past two decades it has been empirically and theoretically proven to provide robust search for complex problems. Because it is modeled closely after the evolutionary paradigm, however, the GA differs in many ways from more traditional search techniques.

As does nature, the genetic algorithm conducts its investigation on the basis of a population of individuals. Competition between members of this population drives the inductive process. The GA is therefore unlike more familiar hill-climbing techniques which climb a functional peak until a maximum value is encountered. The GA climbs many peaks in parallel. As such, its chance of finding a local optimum for the best solution is greatly reduced.

In keeping with the evolutionary metaphor, each individual in the GA population is defined by its gene content. In nature, a gene is a structure holding one of four nucleotide bases: adenine (A), cytosine (C), guanine (G) or thymine (T). In the genetic algorithm, a **gene** is usually a structure holding a binary value. In both the natural and artificial cases, the value of a particular gene is known as its **allele.** Groups of genes are in turn arranged into long strands called chromosomes. For GAs, a **chromosome** is defined as a string of genes (bits). The position of a gene within the chromosome is called its **locus**; genes are numbered left-to-right starting with locus 1.

In nature, the entire collection of chromosomes defining an individual is its genotype. The living organism itself is called a phenotype. In the genetic algorithm, a **phenotype** is a parameter set representing one possible solution to a given problem. The **genotype** is the same solution encoded into a chromosome. As a simple example of this distinction, if we were interested in using a genetic algorithm to find the maximum range of the quadratic function

$$f_x = x^2 - 3x + 6$$

for integers in the interval $[0, 63]$, we could use a 6-bit chromosome ranging in value from `000000` (zero) to `111111` (sixty-three) to represent the problem's domain. One possible member of the population has the genotype `101101` and the phenotype forty-five.

Every niche of the world ecology represents a different problem for the evolutionary process. Through competition and natural selection, organisms are continually being adapted to solve these problems. The algorithm behind natural search, however, remains the same for each niche. In other words, the "survival of the fittest" mechanism is domain-independent. It operates upon organisms in different niches without regard to genotypes or environment. In much the same way, the genetic algorithm works directly on a population of chromosomes without requiring additional information about the particular problem it is solving. It needs only some method for determining the merit of each potential solution. As in population genetics, this evaluation process is called a **fitness function**. Once developed, the same GA can be applied to many different problems; only the fitness function is changed for each application.

Figure 2.1 shows the operation of the simple genetic algorithm. An initial population

```
create $Population$
let $Generation = 0$
repeat
    apply fitness function to each member of $Population$
    let $Generation = Generation + 1$
    perform REPRODUCTION on $Population$
    perform CROSSOVER on $Population$
    perform MUTATION on $Population$
until $Generation = Max$
```

Figure 2.1: The Simple Genetic Algorithm

of strings is arbitrarily created for the first generation. Then, after applying the fitness function, the GA creates a new population of strings to replace the previous generation. This closely parallels natural populations in which organisms are created and eventually reproduce if they survive to maturity. After a predetermined number of generations, the algorithm halts.

The creation of new organisms in nature depends heavily upon many inherently random factors such as the choice of parents. Like other aspects of the genetic algorithm, the procedures REPRODUCTION, CROSSOVER and MUTATION follow the natural precedent for an artificial population of strings. They are each described in detail below. It is important to note that, although the GA utilizes "random operators" for string manipulation, it does not conduct a random walk through the search space. Instead, genetic search uses the fitness function as its guide.

2.1.1 Reproduction

Reproduction is an artificial "survival of the fittest" mechanism by which strings (or chromosomes) are copied according to their fitness. There are many techniques in the literature for performing GA reproduction. The most widely used method is **roulette wheel selection**, which defines the probability of selecting a string s_j for reproduction as

$$p(s_j) = \frac{f(s_j)}{\sum_{i=1}^{N} f(s_i)} \tag{2.1}$$

where f is some fitness function and N is the population size.

Turning once again to the quadratic maximization example, a genetic algorithm might initially generate the population of four strings listed in Figure 2.2, together with their

string (chromosome)		fitness function
genotype	phenotype	value
6-bit string	integer x	$f(x) = x^2 - 3x + 6$
1 0 1 1 0 1	45	1896
0 0 1 0 1 1	11	94
0 1 1 0 1 0	26	604
1 1 0 1 0 0	52	2554

total fitness: 5148

Figure 2.2: Sample GA Population

associated fitness values. Note that in this case, the fitness function is the same as the quadratic equation that is to be optimized. There is no better measure of string optimality than the function value of each string. In many other GA applications, however, no such objective function exists. In these cases a heuristic approximation is used instead.

Dividing the fitness of each individual string by the summation of all fitness values in the population results in the selection probabilities depicted in Figure 2.3. A new population

string (chromosome)	fitness function value	selection %
101101	1896	36.8%
001011	94	1.8%
011010	604	11.8%
110100	2554	49.6%
	5148 total fitness	100%

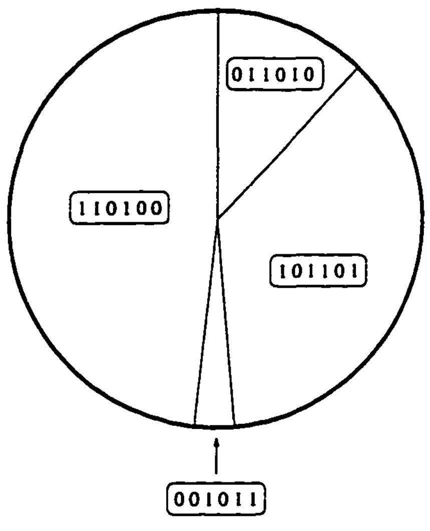


Figure 2.3: Roulette Wheel Selection

would then be created by making four spins on the roulette wheel. Since the probability of selecting 110100 is roughly fifty percent, about half of the individuals in the next generation should be copies of this string. The low-performing string 001011, however, is unlikely to survive into any successive populations. The reproduction operator enables the genetic algorithm to focus its search by emphasizing the best strings discovered.

2.1.2 Crossover

Crossover is a recombination operator for manipulating strings. After reproduction has occurred and a new population is created, crossover proceeds in three steps:

- strings in the new population are arbitrarily paired
- a crossover locus is chosen uniformly along the length of the string
- two new strings are created by swapping all genes after the crossover locus

Figure 2.4 depicts the results of applying the crossover operator to the strings 101101 and

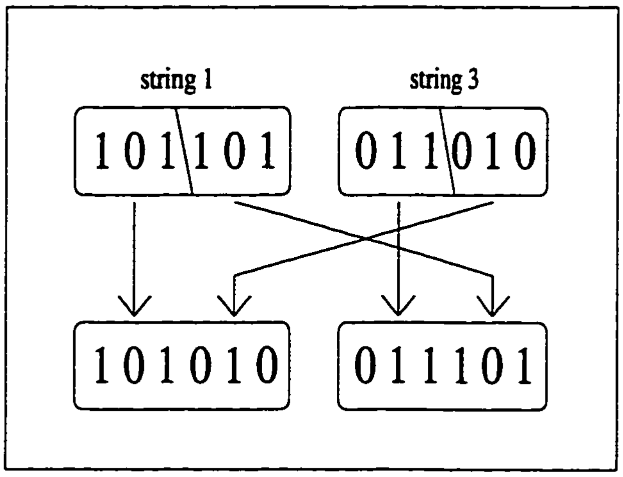


Figure 2.4: Crossover Operator

011010 in the sample population. The "\" mark indicates that the selected crossover site was between the third and fourth genes. Two new strings are then created by trading the three genes after the crossover site. In this manner the crossover operator combines highly fit population members in order to examine new points in the search space.

2.1.3 Mutation

Reproduction with crossover is an effective means of combining high-quality solutions. Occasionally, however, these operators might become overzealous and lose an important allele value. For example, every string in the population might have the allele 0 at the same locus, whereas the optimal solution might require a 1 instead. With reproduction and crossover alone, there would be no way to introduce the allele 1 into the population.

Mutation acts as a safeguard against such an event by occasionally changing the value of a gene after crossover. Figure 2.5 shows the mutation operator applied to the string

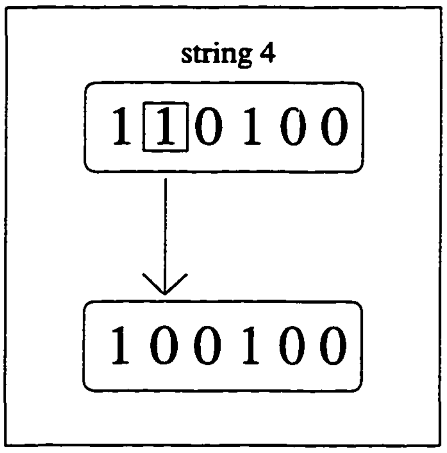


Figure 2.5: Mutation Operator

110100. The "□" symbol shows that the second gene was the mutation site, and thus the 1 was mutated to a 0. Mutation plays only a secondary role in the genetic algorithm. As in nature, the probably of mutation occurring is quite low.

2.2 Schema Processing

The mathematical development of the subsequent section is that of Holland[119].

A **schema** is a simple pattern-matching device for describing basic similarities between strings. Since its introduction into the GA discipline, it has become the basis of all genetic algorithm theory. Simply stated, a schema is a string over the ternary alphabet $\{0, 1, *\}$. The 0 and 1 symbols plays the same role in a schema as they do in a chromosome. The additional "*" symbol acts as a special "don't care" marker and can take on either binary value. It is important to emphasize at this point that "*" is a meta-symbol used only to make theoretical observations about groups of similar chromosomes. It is not used as an allele value in actual genetic search.

A string is said to match a particular schema if one of the following:

- a 1 in the string matches a 1 in the schema
- a 0 in the string matches a 0 in the schema
- either a 0 or a 1 in the string matches a "*" in the schema

holds at every position along the length of both. Two examples of schema for the quadratic maximization example are presented in Figure 2.6. Schema A matches all strings beginning

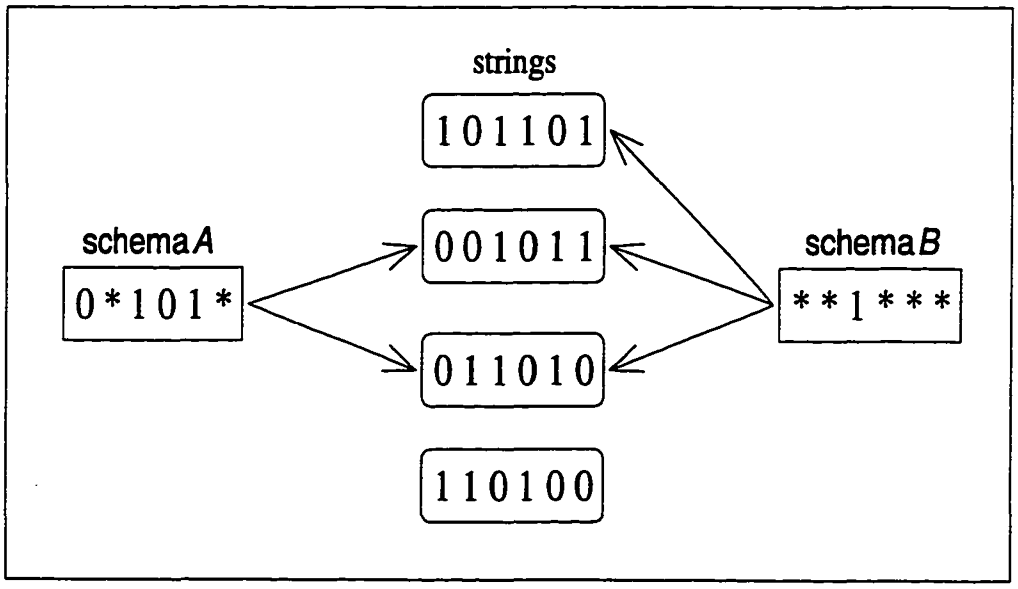


Figure 2.6: Schemata Matching

with a 0 and having the substring 101 from locus 3 to locus 5. Therefore, 0*101* matches only four different strings: the second and third strings of the sample population and the strings 001010 and 011011. Schema *B*, on the other hand, is much less specific and requires only that a 1 be present at locus 3. Since it is concerned with only one out of six possible gene values, it represents $2^{(6-1)} = 32$ different strings. A schema like 110100 matches only one string, since it does not contain a "*". The general schema ******, on the other hand, matches every possible string in the population.

The **order** $o(\mathcal{H})$ is the number of specific positions in schema $\mathcal{H}$. In other words, it is a count of all the non-"*" elements of $\mathcal{H}$. The **defining length** $\delta(\mathcal{H})$ is the difference between the first and last specific position in $\mathcal{H}$. Figure 2.7 depicts o and δ calculations

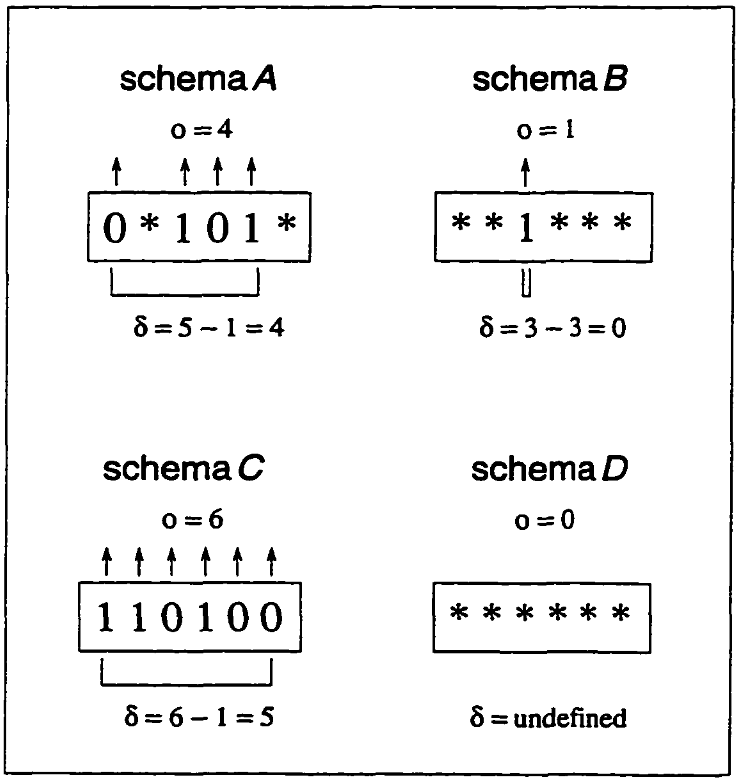


Figure 2.7: Schema Order $o(\mathcal{H})$ and Defining Length $\delta(\mathcal{H})$

for several representative schemata. Very specific schemata such as 110100 would have an order equal to the string length l. The defining length of such strings is equal to $l-1$. The general schema ****** has order 0, since it does not contain any specific information. The defining length of the general schema has no meaning, since no first or last positions exist.

Schemata are useful devices for analyzing the effects of genetic operators on a population $\mathcal{P}$ of strings. To begin, the **average fitness** $\overline{f(t)}$ of $\mathcal{P}$ at generation t is defined as

$$\overline{f(t)} = \frac{\sum_{i=1}^{N} f(s_i, t)}{N} \tag{2.2}$$

where f is a fitness function, s_i is a string in $\mathcal{P}$, and N is the population size.

The **average fitness of a schema** is the average fitness of all the strings in the population matching schema $\mathcal{H}$. Assuming $m(\mathcal{H}, t)$ represents the number of matches of $\mathcal{H}$ in $\mathcal{P}$ at generation t, schema average fitness can be defined as

$$f(\mathcal{H}, t) = \frac{\sum_{s_i \in \mathcal{H}} f(s_i, t)}{m(\mathcal{H}, t)}$$

The **fitness ratio of a schema** is the ratio of the average fitness of a schema $\mathcal{H}$ to the average fitness of $\mathcal{P}$ during generation t, and is expressed as

$$F(\mathcal{H}, t) = \frac{f(\mathcal{H}, t)}{\overline{f(t)}} \tag{2.3}$$

Turning one more time to the quadratic maximization example, Figure 2.3 shows the calculation for total population fitness as $1896 + 94 + 604 + 2554 = 5148$. Dividing the total fitness by the size of the population yields $\overline{f(t)} = 5148/4 = 1287$. Schema A in Figure 2.6 matches only string 001011 (with fitness 94) and string 011010 (with fitness 604). The schema average fitness of A would therefore be $f(A,t) = (94 + 604)/2 = 349$. The fitness ratio of schema A is then $F(A,t) = 349/1287 = 0.27$.

As stated in Equation 2.1, reproduction with roulette wheel selection copies strings into subsequent generations with probability

$$p(s_j) = \frac{f(s_j)}{\sum_{i=1}^{N} f(s_i)}$$

Since each schema $\mathcal{H}$ matches a subset of the strings in the population, it is copied from generation t with probability

$$p(\mathcal{H},t) = \frac{f(\mathcal{H},t)}{\sum_{i=1}^{N} f(s_i)}$$

After selecting N strings for the new generation $t+1$, the expected number of instances of $\mathcal{H}$ is

$$m(\mathcal{H},t+1) = m(\mathcal{H},t)N\frac{f(\mathcal{H},t)}{\sum_{i=1}^{N} f(s_i,t)}$$

Substituting $\overline{f(t)}$ from Equation 2.2 into the above expression results in

$$m(\mathcal{H},t+1) = m(\mathcal{H},t)\frac{f(\mathcal{H},t)}{\overline{f(t)}} = m(\mathcal{H},t)F(\mathcal{H},t) \tag{2.4}$$

where $F(\mathcal{H},t)$ is the schema fitness ratio from Equation 2.3.

Equation 2.4 states that the growth of a schema $\mathcal{H}$ depends only upon whether strings representing $\mathcal{H}$ have a greater fitness value on average than the population as a whole. Schemata with a value above the population average will receive increasing trials in future generations, whereas lower-than-average schemata will be sampled less frequently. At each generation, this process occurs simultaneously for *every* schema over $\mathcal{P}$[1]. Thus, many schemata are effectively processed in parallel.

The effect of crossover on $\mathcal{H}$ depends upon the individual schema itself. The crossover operator recombines strings by breaking them at an arbitrarily chosen position. A schema $\mathcal{H}$ survives crossover only if each specific position in $\mathcal{H}$ remains unaltered. Thus, unless the two strings are identical, crossover will disrupt some of the schemata over $\mathcal{P}$. As an example, schema E in Figure 2.8 can be destroyed by choosing any of the four sites between

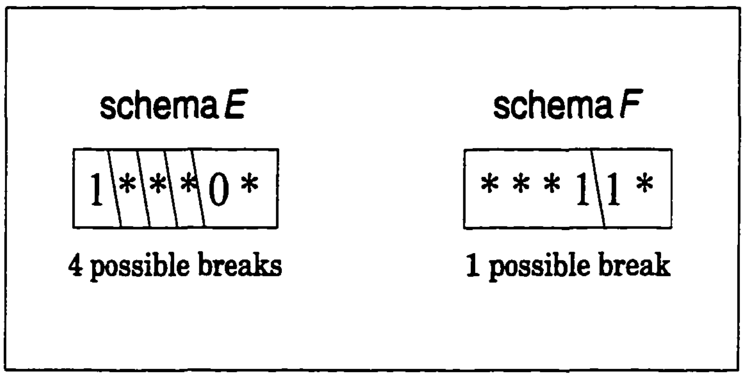


Figure 2.8: Schema Disruption due to Crossover

the 0 and the 1 as the crossover locus. Schema F can only be disrupted if the crossover point is between the two adjacent 1 genes. Close observation reveals that the number of disruptive crossover sites is the same as the defining length δ of a schema.

For strings of length l there are $l - 1$ possible crossover sites. Therefore, the probability p_d of picking a crossover locus which would disrupt $\mathcal{H}$ is

$$p_d(\mathcal{H}) = \frac{\delta(\mathcal{H})}{(l-1)} \tag{2.5}$$

Thus, the example schema E would be disrupted with a $(5-1)/(6-1) = 4/5$ probability,

[1]A schema over $\mathcal{P}$ is a schema matching a member of $\mathcal{P}$

while schema F has only a $(5-4)/5 = 1/5$ chance of being disrupted by crossover.

By extending Equation 2.5, the probability p_s of schema $\mathcal{H}$ surviving crossover would be $1 - p_d(\mathcal{H})$ or

$$p_s(\mathcal{H}) = 1 - \frac{\delta(\mathcal{H})}{l-1}$$

Assuming crossover is performed with probability p_c, schema survival is bounded by the expression

$$p_s(\mathcal{H}) \geq 1 - p_c \frac{\delta(\mathcal{H})}{l-1} \tag{2.6}$$

Multiplying Equation 2.4 by Equation 2.6 results in the expression

$$m(\mathcal{H}, t+1) \geq m(\mathcal{H}, t) \frac{f(\mathcal{H}, t)}{\overline{f(t)}} \left[1 - p_c \frac{\delta(\mathcal{H})}{l-1}\right] \tag{2.7}$$

which describes the combined effects of reproduction and crossover on $\mathcal{H}$. In words, Equation 2.7 states that schemata of short defining length and above-average fitness are increasingly sampled in future generations.

A schema $\mathcal{H}$ survives mutation only if each specific position in $\mathcal{H}$ remains unaltered. Since mutation operates by arbitrarily changing a gene with probability p_m, the chance of each position surviving its effect is $1 - p_m$. The order $o(\mathcal{H})$ represents the number of fixed positions in $\mathcal{H}$. Therefore, the survival probability p_s of $\mathcal{H}$ is

$$p_s(\mathcal{H}) = (1 - p_m)^{o(\mathcal{H})} \tag{2.8}$$

For $p_m \ll 1$, Equation 2.8 can be closely approximated by the expression

$$p_s(\mathcal{H}) = 1 - o(\mathcal{H})p_m \tag{2.9}$$

The joint effect of all three operators is obtained by combing Equation 2.7 with 2.9 with the result

$$m(\mathcal{H}, t+1) \geq m(\mathcal{H}, t)\frac{f(\mathcal{H}, t)}{\overline{f(t)}}\left[1 - p_c\frac{\delta(\mathcal{H})}{l-1}\right][1 - p_m o(\mathcal{H})]$$

Ignoring the small cross-product term

$$\frac{p_c p_m o(\mathcal{H})\delta(\mathcal{H})}{l-1}$$

the above equation can be rewritten as

$$m(\mathcal{H}, t+1) \geq m(\mathcal{H}, t)\frac{f(\mathcal{H}, t)}{\overline{f(t)}}\left[1 - p_c\frac{\delta(\mathcal{H})}{l-1} - p_m o(\mathcal{H})\right] \tag{2.10}$$

Equation 2.10 is the expression of the **Schema Theorem**, also called the **Fundamental Theorem of Genetic Algorithms**. It states that low-order, above-average schemata with short defining lengths are sampled with increasing frequency by the genetic algorithm. Schemata with such properties are given the name **building blocks** because of their special importance to the GA process.

The Fundamental Theorem clearly defines how each schema is simultaneously processed by the genetic algorithm. It makes no statement, however, about the total number of schemata over $\mathcal{P}$. It is a simple matter to count the total number of possible schemata for an arbitrary GA problem. Each position in a given schema can take on one of the three values 0, 1 or "*". For the example chromosomes of length 6, there would therefore be $3 * 3 * 3 * 3 * 3 * 3 = 3^6 = 729$ different schemata. In general, for genes with k allele values contained in chromosomes of length l, there are $(k+1)^l$ possible schemata. As stated on page 12, during schema matching each gene is compared to the corresponding schema position. It matches if the string and the schema have identical values (either a 1 or 0) or the schema contains a "*" symbol. Since each position of a string matches a schema position in two ways, a string of length l would represent 2^l different schemata. Therefore, a population $\mathcal{P}$ of size N matches n different schemata at generation t as bounded by the expression

$$2^l \leq n(\mathcal{P}, t) \leq N2^l \tag{2.11}$$

It can be seen from the above equation that the GA possesses the ability to create populations of strings in such a way as to simultaneously maximize a great number of schemata. This property of **implicit parallelism** is an important theoretical result, as it allows the genetic algorithm to use combinatorial explosion to its advantage.

2.3 Development of Genetic Algorithms

In the early sixties a number of biologists such as Fraser[82] and Barricelli[16] were running computer simulations of genetic systems in an attempt to better understand natural processes. Fraser's work even employed binary strings, a fitness function and a reproduction operator. There was however no mention in these early works that natural search could be applied to artificial problems.

A few years later, Holland[118] developed his *adaptive system theory*, describing in part how the genetic process could be applied to arbitrary search problems. His early work stressed the importance of selection as a "survival of the fittest" mechanism. He also emphasized using a population of search points, and mentioned recombination operators such as crossover and mutation.

Bagley[13] was the first to coin the phrase "genetic algorithm". For his dissertation, Bagley constructed a GA for finding an evaluation function for *hexapawn*, a game in which two players start with three pawns apiece and attempt to cross a three by three chess board. Bagley's GA contained the three standard genetic operators of reproduction, crossover, and mutation. However, these were applied to *diploid chromosomes*, which consist of two joined strings. Each gene along a diploid chromosome takes on the value of the most dominant allele at each locus.

Rosenberg's[206] work involved the biological simulation of a population of single-celled animals. As part of his research, he developed an adaptive crossover mechanism in which *linkage factors* where carried along with allele values to mark the probability of crossover occurring at each locus. His work was very similar to later optimization and root finding tasks. At about this time, Holland[119] was mathematically formulating the underlying processing power of the genetic algorithm with his schema theory.

Hollstein[127] was the first to apply genetic algorithms to mathematical optimization problems. His GA used crossover, mutation and other genetic operators, and was able to find optima much more rapidly than the traditional hill-climbing techniques. Hollstein also investigated five different selection methods and eight mating techniques borrowed from horticulture and animal husbandry practices.

Frantz[81] studied the positional effect of genes on function optimization. Specifically, he considered functions with a strong **epistatic** nature; that is, functions in which important genes were separated by relatively large distances along the chromosome. Frantz used roulette wheel selection, simple crossover and simple mutation, and was able to show that tight linkage between genes (low epistasis) increased the rate of population improvement.

For his dissertation, DeJong[62] applied Holland's schema theory to five problems in function minimization ranging in scope from a smooth, unimodal function of two variables to a function characterized by discontinuity, great multi-modality and high dimensionality. By varying the population size, selection technique, and mutation and crossover probabilities, he was able to quantify the performance of genetic algorithms in terms of both interim operation and final convergence. His work demonstrated by experiment the robustness of the GA across many optimization problems.

Chapter 3

Rule Representation

3.1 Binary Representation

There are several difficulties inherent in the simple binary encoding schemes used by standard genetic algorithms. Since the number of chromosome interpretations is not always a power of two, GAs often contain redundant information which may unevenly weight the search process while at the same time reducing the effectiveness of recombination operators. As an example, let us turn again to the simple quadratic function from the previous chapter. If we were interested in finding the maximum range of

$$f_x = x^2 - 3x + 6$$

for integers in the extended interval $[0, 64]$, we would require a 7-bit chromosome to represent the phenotypes 0000000 through 1000000. As can be seen, the sixty-three chromosomes from 1000001 to 1111111 would not have a meaningful genotype interpretation. In other words, almost half of the possible strings in the population would have no discernible fitness value.

Using binary-valued chromosomes can also make it difficult to solve problems whose solution schemata are of high order and defining length. Consider, for example, the problem of finding the maximum range value for the function

$$f_x = x(x+3) \bmod 32$$

in the interval $[0, 63]$. As shown in Figure 3.1, there are four optimal solutions to this

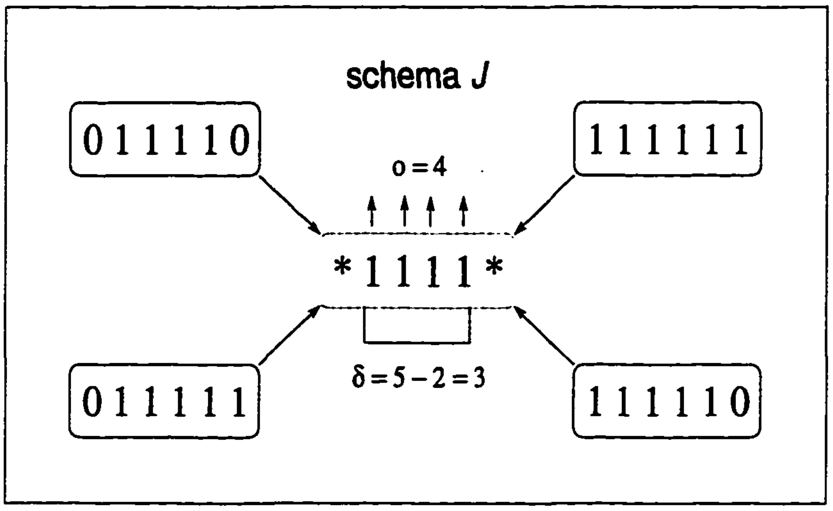


Figure 3.1: Schema for $f_x = x(x+3) \bmod 32$

problem, each of which matches the schema J. Since the defining length $\delta(J) = 3$ and there are five possible crossover loci, there is a $3/5 = 0.60$ probability of crossover breaking apart the solution schema. Similarly, since the order $o(J) = 4$ and there are six total genes, there is a $4/6 = 0.67$ probability of mutation disrupting schema J. Since both recombination operators are more likely to destroy the solution schema than to preserve it, it is unlikely that schema J will be optimally represented in the population using genetic search.

Under the traditional genetic algorithm paradigm, mutation operates upon individual bits within a chromosome. Since the probability p_m of mutation occurring is typically quite low, many chromosome-wide mutations are unlikely to occur. Turning again to the quadratic maximization problem of page 22 for the interval $[0, 64]$, it is highly improbable that the second best solution 0111111 could be mutated to the optimal 1000000. Point-by-point mutation on binary genes does not foster new schema discovery in cases where several allele values must be simultaneously altered.

Binary encoding also makes it impossible for genetic algorithms to use variables in the traditional sense. Without semantic information to call upon, the GA cannot express complex interdependencies between features in the environment, thereby making it difficult to add built-in knowledge and world models. Machine learning applications requiring heuristic discovery or explanation-based reasoning processes are therefore severely limited.

Finally, binary representation serves to isolate genetic algorithms from mainstream Artificial Intelligence. Most AI applications such as Expert Systems rely heavily upon more expressive representations describing cognitive aspects of a particular domain. It is difficult to abstract useful information from bit string chromosomes and then translate this into the high-order representations prevalent in Artificial Intelligence research. It is even more difficult to effectively encode a semantic knowledge representation into binary chromosomes. Genetic-based techniques are therefore commonly overlooked by the larger AI community.

3.2 Rule Representation

Using a higher level knowledge representation offers an easy solution to the above problems. In the past, there have been several proposals for modifying the binary representation scheme of genetic algorithms. Each of these efforts, however, has focused on adapting the GA to one problem-specific structure.

The representation scheme presented in this dissertation provides a general-purpose strategy for applying the GA to an arbitrary representation. This is made possible by adopting a grammar-based approach to genetic search. Using this technique, all members of the GA population are strings derivable from a problem-specific grammar. Therefore. any structure which can be expressed in Backus-Naur Form (BNF) can be manipulated by genetic operators. As such, a grammar-based GA strategy provides a consistent methodology for handling any population structure that can be expressed in terms of a context-free grammar. As will be shown in subsequent chapters, the class of such problems is large.

3.2.1 Grammar-Based Approach

Figure 3.2 shows a sample grammar for generating simple if-then decision rules for playing

```
 1:      <rule> → ( if <condition> ( put-X ) )
 2: <condition> → <statement>
 3: <condition> → ( and <statement> <exp> )
 4: <condition> → ( and <statement> <exp> <exp> )
 5: <condition> → ( and <statement> <exp> <exp> <exp> )
 6: <statement> → ( is-empty <square> )
 7:       <exp> → ( <predicate> <square> )
 8: <predicate> → <attribute>
 9: <attribute> → is-X
10: <attribute> → is-O
11: <attribute> → is-empty
12:    <square> → <position>
13:  <position> → top-left
14:  <position> → top-center
15:  <position> → top-right
16:  <position> → middle-left
17:  <position> → middle-center
18:  <position> → middle-right
19:  <position> → bottom-left
20:  <position> → bottom-middle
21:  <position> → bottom-right
```

Figure 3.2: A Sample Rule Grammar for Tic-Tac-Toe

the game of Tic-Tac-Toe. When generating rules from the above grammar, it is assumed that the player is always X; therefore, **Put-X** appears as the action of every rule. Each rule states that if certain board conditions are encountered on the player's turn, an X should be placed in the square tested by the first **is-empty** predicate.

For Figure 3.2 and all subsequent grammars, non-terminal symbols are shown in brackets, while terminal symbols are in boldface. Each integer to the left of a rule is a **production label** which serves to uniquely identify each production in the grammar. Production 1 is always assumed to be the start symbol expansion.

In order for the genetic algorithm to use a grammar effectively, each individual in the population must be accompanied by a *derivation trace*:

Definition 3.1 *A* **derivation trace** $\mathcal{X} = x_1x_2..x_n$ *has the following syntax:*

<derivation trace>	→	**1 0**
<derivation trace>	→	**1** <subtrace series> **0**
<subtrace series>	→	<subtrace series> <derivation subtrace>
<subtrace series>	→	<derivation subtrace>
<derivation subtrace>	→	<production label> **0**
<derivation subtrace>	→	<production label> <subtrace series> **0**
<production label>	→	positive integer

Each trace is therefore represented as a list of integers corresponding to the labels of the productions which were used when generating a rule. This list is additionally annotated by zeros to mark the completed expansion of each production during a leftmost derivation through the grammar. Each production therefore contributes two pieces of information to

```
let $Rule$ = left-hand side of production 1
let $Trace = Rule$
repeat
    let $L$ = left-most non-terminal in $Rule$
    let $P$ = arbitrarily selected label of a production with left-hand side $L$
    let $R$ = right-hand side of production $P$
    let $S = s_1s_2..s_n$ be the non-terminals in $R$ from left to right
    let $Rule = Rule$ with $L$ replaced by $R$
    let $Trace = Trace$ with $L$ replaced by the string $Ps_1s_2..s_n0$
until no non-terminals in $Rule$
```

Figure 3.3: Rule and Derivation Trace Creation

the trace - a unique integer label marking its application and a zero marking its completion. As will be seen, this zero-embedded representation allows the genetic operators to create new, syntactically correct rules while preserving the "building block" nature of the GA search.

rule	derivation trace
<rule>	<rule>
(**if** <condition> (**put-X**))	1 <condition> 0
(**if** (**and** <statement> <expression>) (**put-X**))	1 3 <statement> <expression> 0 0
(**if** (**and** (**is-empty** <square>) <expression> (**put-X**))	1 3 6 <square> 0 <expression> 0 0
(**if** (**and** (**is-empty** <position>) <expression>) (**put-X**))	1 3 6 12 <position> 0 0 <expression> 0 0
(**if** (**and** (**is-empty middle-right**) <expression>) (**put-X**))	1 3 6 12 18 0 0 0 <expression> 0 0
(**if** (**and** (**is-empty middle-right**) (<predicate> <square>)) (**put-X**))	1 3 6 12 18 0 0 0 7 <predicate> <square> 0 0 0
(**if** (**and** (**is-empty middle-right**) (<attribute> <square>)) (**put-X**))	1 3 6 12 18 0 0 0 7 <attribute> <square> 0 0 0
(**if** (**and** (**is-empty middle-right**) (**is-O** <square>)) (**put-X**))	1 3 6 12 18 0 0 0 7 8 10 0 0 <square> 0 0 0
(**if** (**and** (**is-empty middle-right**) (**is-O** <position>)) (**put-X**))	1 3 6 12 18 0 0 0 7 8 10 0 0 12 <position> 0 0 0 0
(**if** (**and** (**is-empty middle-right**) (**is-O middle-left**)) (**put-X**))	1 3 6 12 18 0 0 0 7 8 10 0 0 12 16 0 0 0 0 0

Figure 3.4: Derivation of a Sample Rule and Trace

Every rule and its corresponding trace are initially constructed following the algorithm presented in Figure 3.3. Each rule is initially set to the right-hand side of production 1. The left-most non-terminal symbol L in the rule is then replaced by the right-hand side of an arbitrarily selected production whose left-hand side matches L. This process continues until all non-terminal symbols have been removed from the rule.

The corresponding derivation trace is constructed in a similar manner; however, each L is replaced by a list containing only the non-terminal symbols contained in the expansion. This list is additionally headed by the label of the selected production, and is delimited by

Chromosome	
genotype derivation trace	phenotype production rule

1 4 6 12 16 0 0 0
7 8 9 0 0 12 15 0 0 0
7 8 10 0 0 12 19 0 0 0 0 0
	(if (and (is-empty middle-left)
	(is-X top-right)
	(is-O bottom-left))
	(put-X))

1 2 6 12 17 0 0 0 0 0	(if (is-empty middle-center) (put-X))

1 5 6 12 21 0 0 0
7 8 9 0 0 12 19 0 0 0
7 8 10 0 0 12 13 0 0 0
7 8 11 0 0 12 20 0 0 0 0 0
	(if (and (is-empty bottom-right)
	(is-X bottom-left)
	(is-O top-left)
	(is-empty bottom-center))
	(put-X))

1 3 6 12 18 0 0 0 7 8 10 0 0 12 16 0 0 0 0 0	(if (and (is-empty middle-right) (is-O middle-left)) (put-X))

Figure 3.5: Sample Population of Tic-Tac-Toe Rules

a zero expansion marker. Figure 3.4 illustrates the creation of an arbitrary rule from the above grammar, as well as the construction of its corresponding derivation trace. Figure 3.5 shows one possible population of rules arbitrarily derived from the Tic-Tac-Toe grammar.

At first glance, the derivation of rules from a problem-specific grammar might seem to place an added burden on the genetic algorithm. In effect, however, traditional GAs must perform similar tasks in order to access valid population structures. The if-then rules created by the Tic-Tac-Toe grammar directly correspond to population phenotypes. Genotypes are the derivation traces describing how each phenotype was uniquely generated. In other words, genotypes are obtained from phenotypes during rule creation.

Traditional genetic algorithms, on the other hand, use simple bit-strings as genotypes. These strings must be decoded by a fitness function in order to access problem parameters. In other words, phenotypes are obtained from genotypes during rule evaluation. In both cases, translation between population phenotypes and genotypes is necessary. The only difference lies in when and how this translation occurs.

For example, the rules derived from the Tic-Tac-Toe grammar of Figure 3.2 can also be represented in binary. Figure 3.6 shows how an 18-bit string could be used as the genotype

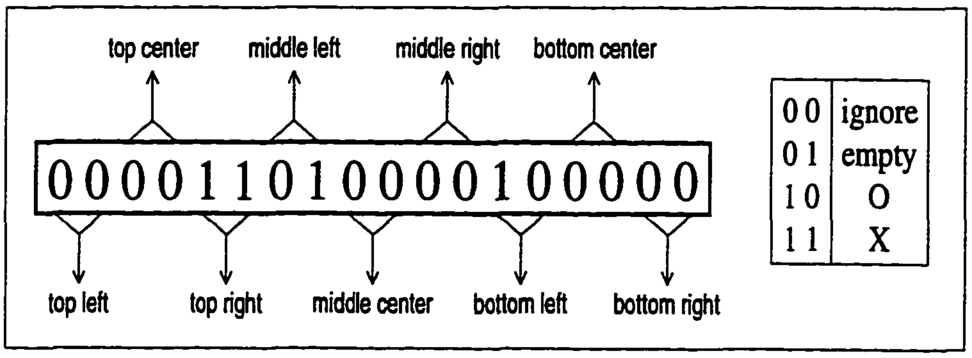


Figure 3.6: Binary Encoding For Tic-Tac-Toe Rules

for the first member of the rule population. Every board position is described by two bits which state whether the square contains an X, a O or is empty. Ignore simply means that no information about that square is required by the rule. A fitness function for such bit-strings would be required to translate this genotype before game board evaluation. As will be shown in Chapter 4.4, it is important to note that most grammar-derived rules cannot be effectively represented in binary form.

3.2.2 Rule Crossover

As explained in Chapter 2, the standard crossover operator functions by exchanging genetic material between two individuals. In traditional GAs, a position or locus is used to determine how much information each parent passes on to its offspring. The genes before this locus in the first parent are combined with the genes after this locus in the second in order to create a new population member.

As shown in Figure 3.7, a modified version of this operation is needed to perform

Crossover of Parents $\mathcal{X}$ and $\mathcal{Y}$ to Produce Offspring $\mathcal{X}'$ and $\mathcal{Y}'$

```
let $\mathcal{X}$ = derivation trace of first parent
let $\mathcal{Y}$ = derivation trace of second parent
let $P_C$ = arbitrarily selected member of $I_{\mathcal{XY}} - TLS_G$
let $i$ = index of arbitrarily selected occurrence of $P_C$ in $\mathcal{X}$
let $j$ = index of arbitrarily selected occurrence of $P_C$ in $\mathcal{Y}$
let $S_{\mathcal{X}}$ = derivation subtrace headed by $P_C$ at index $i$
let $S_{\mathcal{Y}}$ = derivation subtrace headed by $P_C$ at index $j$
let $\mathcal{X}'$ = child of $\mathcal{X}$ with $S_{\mathcal{X}}$ replaced by $S_{\mathcal{Y}}$
let $\mathcal{Y}'$ = child of $\mathcal{Y}$ with $S_{\mathcal{Y}}$ replaced by $S_{\mathcal{X}}$
```

Figure 3.7: Rule Crossover Algorithm

crossover on grammar-based rules. The first step in this process is to determine the *intersection* of the parents' derivation traces.

Definition 3.2 *Given a derivation trace $\mathcal{X}$, the* **production label set** $L_{\mathcal{X}}$ *is defined as* $\{y \mid \exists i\,[\mathcal{X} = x_1..x_i..x_n \text{ and } y = x_i]\}$. *The* **intersection of derivation traces** $\mathcal{X}$ *and* $\mathcal{Y}$ *is then equivalent to the set* $I_{\mathcal{XY}} = L_{\mathcal{X}} \cap L_{\mathcal{Y}}$.

Since zero is only a marker and does not label any production, $I_{\mathcal{XY}}$ is equivalent to the set of all non-zero elements common to both traces.

Set difference is then performed between $I_{\mathcal{XY}}$ and a grammar-specific *Terminal Label Set.*

Definition 3.3 *The* **Terminal Label Set** *of a grammar G is the set TLS_G of labels of productions of the form $P_i \rightarrow s_1s_2..s_n$, such that every s_i is a terminal symbol.*

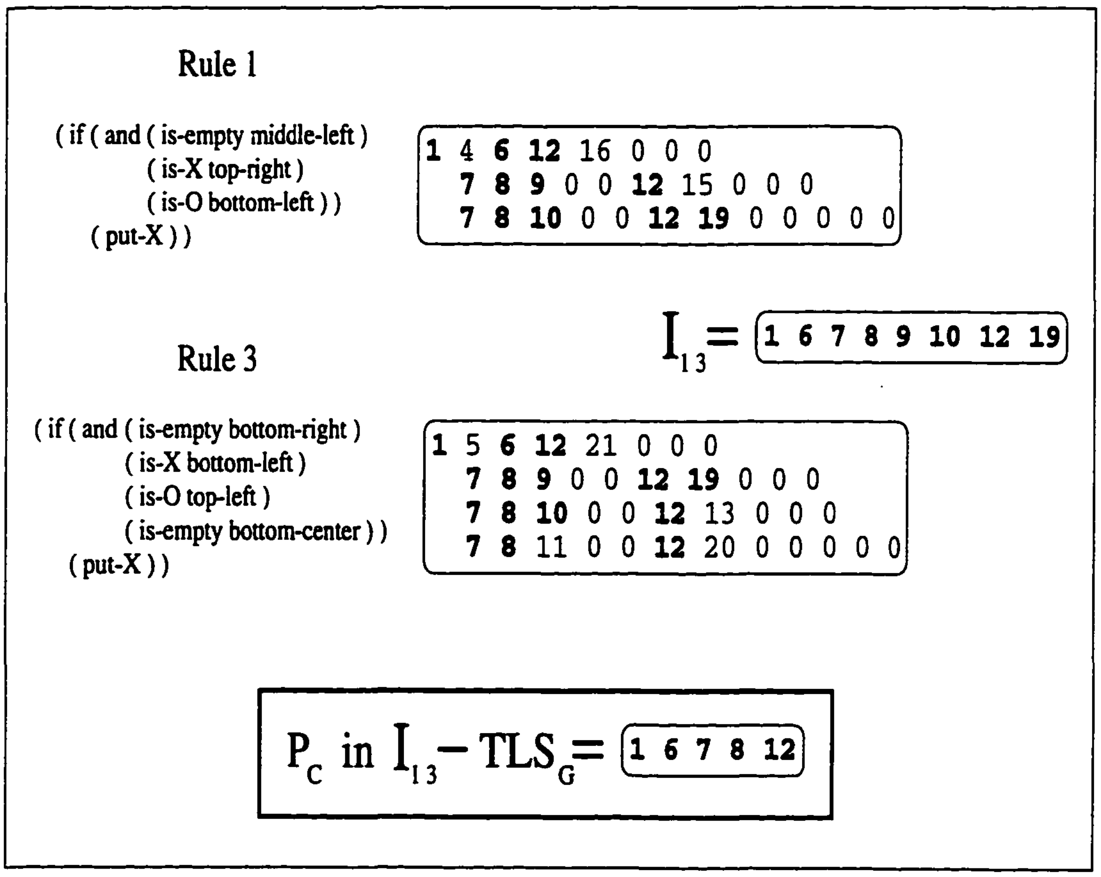


Figure 3.8: The Selection of the Crossover Production Label

The result of this operation is the set of all shared labels to productions whose expansions contain at least one non-terminal symbol. The rationale behind the elimination of TLS_G productions is explained below. One member of $I_{XY} - TLS_G$ is then chosen as the *crossover production label*

Definition 3.4 *Given traces $\mathcal{X}$ and $\mathcal{Y}$ with intersection I_{XY}, the* **crossover production label** *is an arbitrarily selected production label $P_C \in I_{XY} - TLS_G$*

Figure 3.8 shows the method by which the crossover production label is selected in greater detail. Note that for non-trivial grammars P_C is guaranteed to be non-empty, since all traces must contain the first production of the grammar.

In many cases, several instances of the label P_C can be found in a parent trace. For

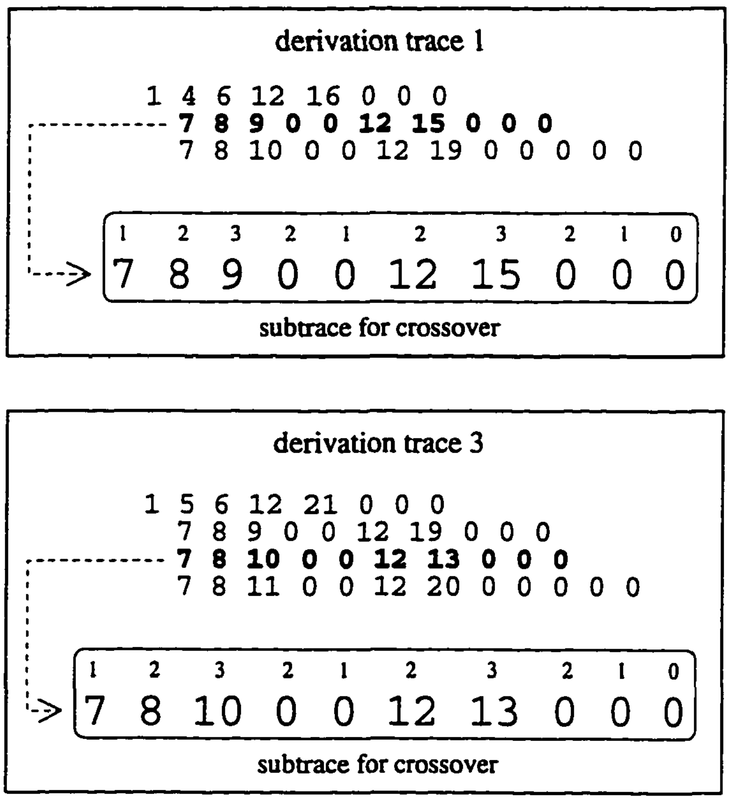


Figure 3.9: Determination of Crossover Subtraces

example, the labels 7, 8 and 12 occur many times in the traces of both Rule 1 and Rule 3. Therefore, it is necessary to determine the *trace loci for crossover.*

Definition 3.5 *Given traces* $\mathcal{X} = x_1x_2..x_n$ *and* $\mathcal{Y} = y_1y_2..y_m$ *with crossover production label* P_C, i *and* j *are* **trace loci for crossover** *iff* $1 \leq i \leq n$ *and* $1 \leq j \leq m$ *and* $x_i = y_j = P_C$.

Unlike traditional crossover, the trace locus is not used to split rules apart in order to exchange "before" and "after" pairs. Such a method would not ensure that syntactically valid traces would be generated. Instead, the derivation traces of both rules are examined in order to find *derivation subtraces.*

Definition 3.6 *A* **derivation subtrace** *is a contiguous subsequence* $S = x_Cx_{C+1}..x_{Index-1}$ *of* $\mathcal{X} = x_1x_2..x_n$, *determined by the following algorithm:*

```
let $Parity = 0$
let $Index$ = locus $C$ in $\mathcal{X} = x_1x_2..x_n$
repeat
    if $x_{Index} > 0$
        let $Parity = Parity + 1$
    else
        let $Parity = Parity - 1$
    let $Index = Index + 1$
until $Parity = 0$
let $S = x_Cx_{C+1}..x_{Index-1}$
```

Beginning with locus C, the derivation trace is searched from left to right until the number of zero and non-zero elements examined is equal. This section of the trace represents a parsed subtree headed by the selected trace locus. Assuming label 7 was chosen as the production locus, Figure 3.9 shows how the derivation subtraces for crossover can be found by parity count.

As depicted in Figure 3.10, crossover subtraces are then exchanged between the parents

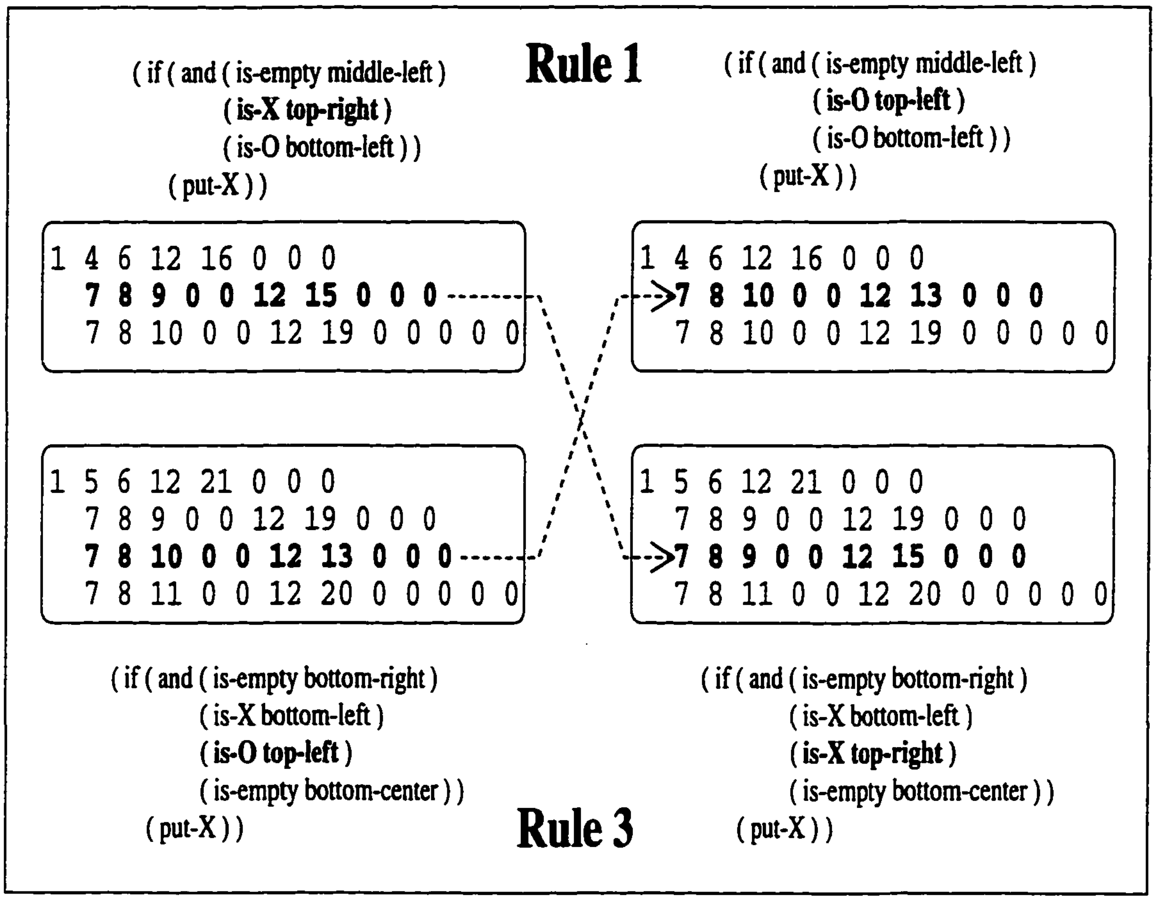


Figure 3.10: Crossover between Grammar-Based Rules

in order to form two new rules. Since this modified crossover process permits an exchange of data only between subtrees with common root elements, syntactically valid rules will always be generated. Although some computation time is required to find elements common to both parents, new rules do not have to be parsed in the grammar. New traces are automatically generated through the crossover operator, and it is then a simple matter to construct the corresponding rule.

Since productions in TLS_G contain only terminal symbols, the subtraces which they head will always be of constant form. For example, every derivation subtrace headed by production label **9** will be of the form **9 0**. Crossover between members of set TLS_G would therefore lead to the non-productive exchange of identical subtrees.

It is for a similar reason that the Tic-Tac-Toe grammar contains such transition structures as

```
 7:       <exp> → ( <predicate> <square> )
 8: <predicate> → <attribute>
 9: <attribute> → is-X
10: <attribute> → is-O
11: <attribute> → is-empty
```

At first glance, it might seem that production **8** above could eliminated, resulting in the production set

```
 7:       <exp> → ( <attribute> <square> )
 8: <attribute> → is-X
 9: <attribute> → is-O
10: <attribute> → is-empty
```

The original production **8** is necessary, however, as it serves as a mechanism for choosing between three different production labels. Without this intermediate step, the crossover operator would be incapable of exchanging different `<attribute>` values. For example, two partial derivation subtraces from the revised grammar might be

7 8 0 <square> 0
7 9 0 <square> 0

In the above example, the subtraces 8 0 and 9 0 could not be exchanged through crossover, since no common production locus exists. Crossover could only occur between entire subtraces, permanently linking all `<attribute>` and `<square>` values in production 7.

Using the original production set, the corresponding partial subtraces would be

7 8 9 0 0 <square> 0
7 8 10 0 0 <square> 0

In this case, production 8 could be selected as the locus for crossover, and the two subtraces 8 9 0 0 and 8 10 0 0 could be independently swapped. **Branch productions** such as 8 are commonplace in many of the grammars presented in this paper.

3.2.3 Rule Mutation

The crossover operator described above is sufficiently able to exchange information between existing rule structures in a grammar-based GA; crossover alone, however, cannot introduce new material into the system. As is the case with traditional GAs, a mutation operator is necessary to accomplish this task. Unlike these systems, though, occasionally miscopying a gene is not enough. Care must be taken to mutate a rule and still produce syntactically valid structures.

Figure 3.11 depicts the algorithm for performing mutation on grammar-derived rules.

Mutation of $\mathcal{X}$ to Produce $\mathcal{X}'$

```
let $\mathcal{X}$ = derivation trace of original rule
let $P_M$ = arbitrarily selected member of $L_\mathcal{X} - TLS_G$
let $i$ = index of arbitrarily selected occurrence of $P_M$ in $\mathcal{X}$
let $S_\mathcal{X}$ = derivation subtrace headed by $P_M$ at index $i$
let $S_M$ = arbitrarily generated derivation subtrace headed by $P_M$
let $\mathcal{X}'$ = mutated $\mathcal{X}$ with $S_\mathcal{X}$ replaced by $S_M$
```

Figure 3.11: Rule Mutation Algorithm

First, the derivation trace is examined in order to find the trace's production label set $L_\mathcal{X}$. As with crossover, set difference is then performed between $L_\mathcal{X}$ and TLS_G. The result is the set of all labels to productions in the rule's derivation whose expansion contains at least one non-terminal symbol. One member of $L_\mathcal{X} - TLS_G$ is then selected as the *mutation production label.*

Definition 3.7 *Given trace $\mathcal{X}$ with production label set $L_\mathcal{X}$, the* **mutation production label** *is an arbitrarily selected production label $P_M \in L_\mathcal{X} - TLS_G$.*

Figure 3.12 shows this calculation in greater detail for Rule 4 in the example population.

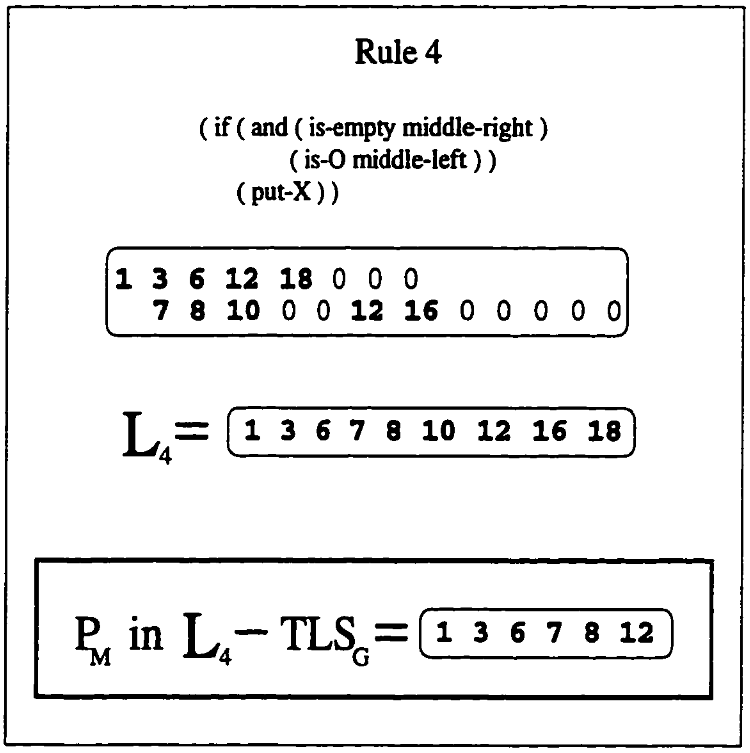

Rule 4
(if (and (is-empty middle-right)
(is-O middle-left))
(put-X))
1 3 6 12 18 0 0 0
7 8 10 0 0 12 16 0 0 0 0 0
L_4 = 1 3 6 7 8 10 12 16 18
P_M in $L_4 - TLS_G$ = 1 3 6 7 8 12

Figure 3.12: Selection of the Production Locus for Mutation

As a final step, the *trace locus for mutation* must be determined for the rule which is to be altered.

Definition 3.8 *Given trace $\mathcal{X} = x_1x_2..x_n$ and mutation production label P_M, i is the* **trace locus for mutation** *iff $1 \leq i \leq n$ and $x_i = P_M$.*

The derivation subtrace for mutation can then be determined using the parity calculation

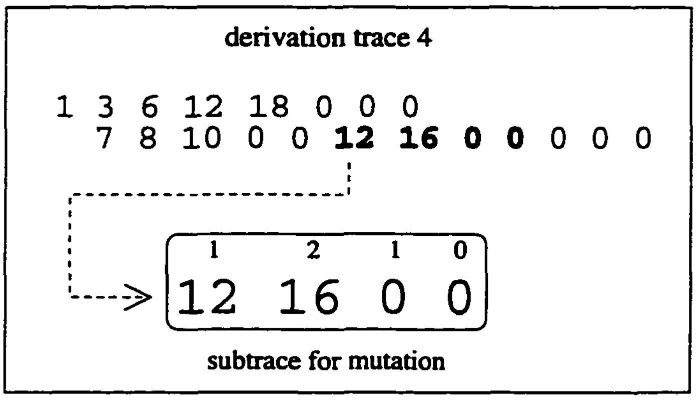


Figure 3.13: Determination of the Mutation Subtrace

algorithm presented in Definition 3.6. One possible mutation subtrace is presented in Figure 3.13. In this example, label 12 serves as production locus.

Next, the mutation subexpression is removed from the trace. It is replaced by a new

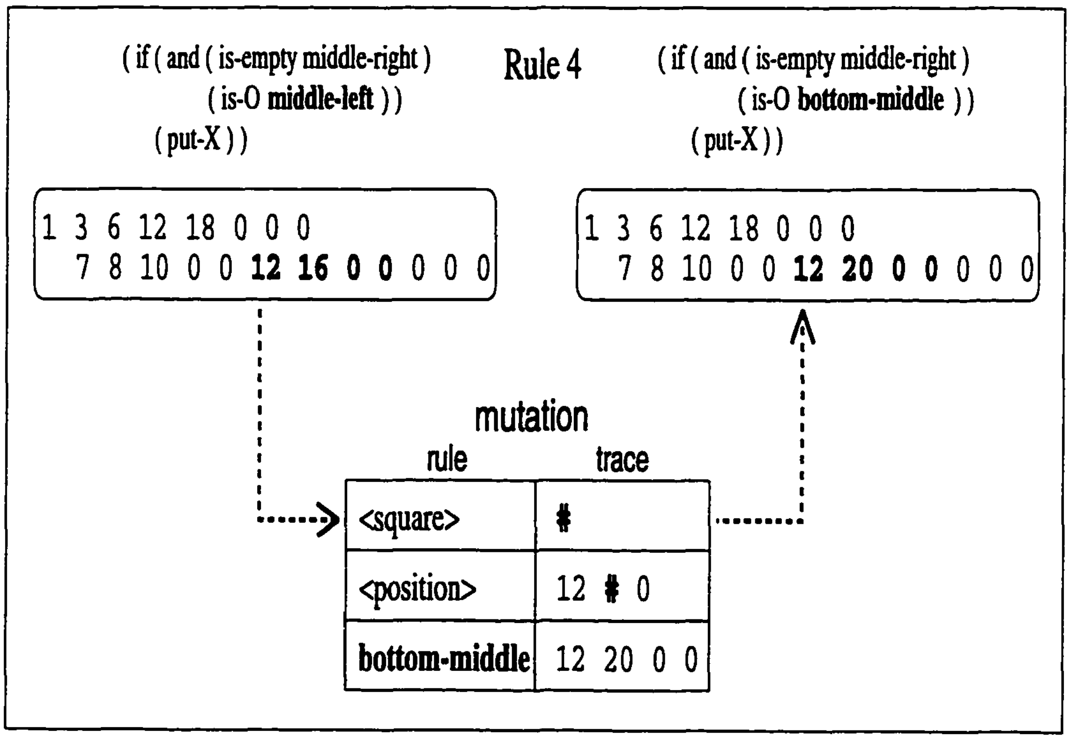


Figure 3.14: Mutation of a Grammar-Based Rule

subtrace arbitrarily created from the grammar using the mutation locus as the starting production. Figure 3.14 shows the mutation of **Rule 4** from the sample population. It is important to note that mutation does not automatically guarantee that an alteration will be made to an individual rule. Because the replacement subtrace is arbitrarily generated, it is possible that all selected productions will be identical to those of the original subtrace. For this reason, the rate at which this operator is applied to population members is generally higher than that of its traditional counterpart. As with crossover, members of TLS_G are excluded from P_M selection because the subtraces they head are always a constant structure. Therefore, a "random" derivation headed by a TLS_G production would *always* be identical to the pre-mutated subtrace.

3.3 GERDS

As part of the research presented in this dissertation, the GERDS (GEnetic Rule Discovery System) package was developed to execute the grammar-based search process described in the previous section. GERDS was implemented in modular form using Common Lisp[240]. A general overview of the GERDS algorithm is presented in Figure 3.15. As can be seen, the design of the system is quite similar to that of the traditional genetic algorithm depicted in Figure 2.1.

The first step undertaken by GERDS is to load a problem-specific EXPERIMENT FILE into the system. This file sets the values for global variables such as the crossover and mutation rates. It also defines the location of the four other external files used by the system. Figure 3.16 lists all GERDS variables and their default values. These values can be redefined in the EXPERIMENT FILE using the Lisp **setq** command. For example, in order to change the population size variable from its default value to 150, the line

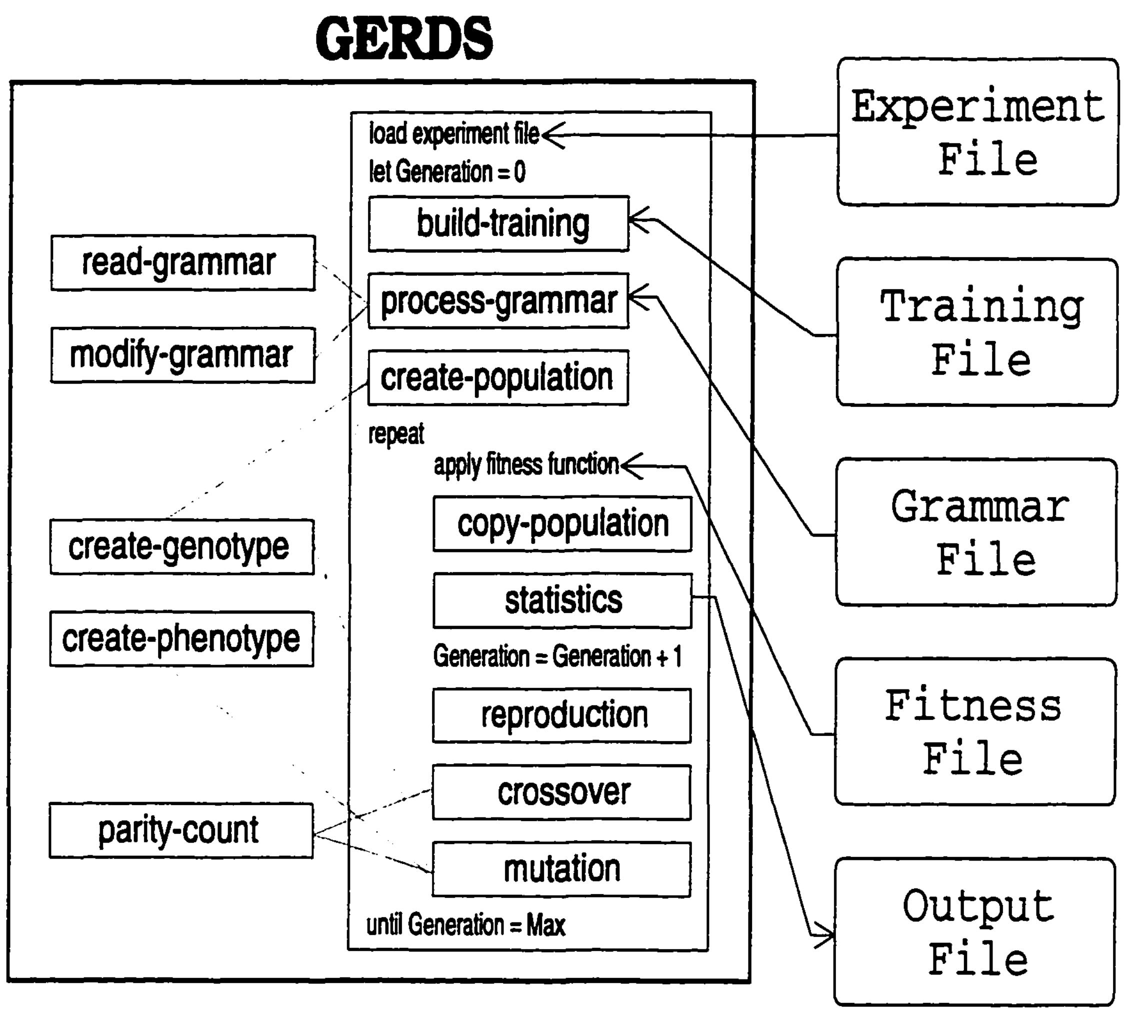


Figure 3.15: Overview of the GERDS algorithm

(setq *population* 150)

should appear somewhere in the file.

Next, the **read-training** module loads a set of example cases from TRAINING FILE. This data is then stored internally in `<train-list>`, which can be accessed by the fitness function to test population members during classification and machine learning experiments. In the default case that `*train-file*` is `nil`, the **read-training** module is not executed.

The **process-grammar** module then constructs an internal grammar for the experiment.

DESCRIPTION	GLOBAL	DEFAULT
the number of "fittest" individuals to display to output file after each generation	***best-display***	5
the percent chance of crossover taking place between two individuals	***crossover-rate***	0.6
the name and path of the file where the function "fitness" can be found	***fitness-file***	"fitness-function.o"
the total number of generations to evaluate during the course of the experiment	***generations***	100
the name and path of the file where the grammar for creating population members can be found	***grammar-file***	"grammar.lsp"
the percent chance of mutation affecting each allele of an individual	***mutation-rate***	0.02
the name and path of the file where statistical information on performance will be sent	***output-file***	"output.lsp"
the number of "fittest" individuals which automatically survive into the next generation	***overlap***	0
the total number of individuals contained in the population for this experiment	***population***	100
the name and path of the file where example data can be found (if required by the fitness function)	***train-file***	nil

Figure 3.16: Global Parameters used by the GERDS algorithm

It calls **read-grammar**, which inputs the set of productions listed in GRAMMAR FILE. This information is then passed to **modify-grammar**, which transforms these rules so that table lookup can be used when referencing grammar symbols. Figure 3.17 shows an example GRAMMAR FILE describing the Tic-Tac-Toe grammar used in earlier examples. Each transition rule in this file is represented as a list whose first element is the left-hand side of the production. For example,

(CONDITION (AND STATEMENT EXPRESSION))

GRAMMAR FILE

```
( CROSSOVER RULE STATEMENT EXPRESSION PREDICATE ATTRIBUTE )
( MUTATE RULE STATEMENT EXPRESSION PREDICATE ATTRIBUTE )

( RULE ( IF CONDITION ( PUT-X ) ) )
( CONDITION STATEMENT )
( CONDITION ( AND STATEMENT EXPRESSION ) )
( CONDITION ( AND STATEMENT EXPRESSION EXPRESSION ) )
( CONDITION ( AND STATEMENT EXPRESSION EXPRESSION EXPRESSION ) )
( STATEMENT ( IS-EMPTY SQUARE ) )
( EXPRESSION ( PREDICATE SQUARE ) )
( PREDICATE ATTRIBUTE )
( ATTRIBUTE IS-X)
( ATTRIBUTE IS-O )
( ATTRIBUTE IS-EMPTY )
( SQUARE POSITION )
( POSITION TOP-LEFT )
( POSITION TOP-CENTER )
( POSITION TOP-RIGHT )
( POSITION MIDDLE-LEFT )
( POSITION MIDDLE-CENTER )
( POSITION MIDDLE-RIGHT )
( POSITION BOTTOM-LEFT )
( POSITION BOTTOM-CENTER )
( POSITION BOTTOM-RIGHT )
```

Figure 3.17: Sample Grammar File Processed by GERDS

corresponds to the production

<condition> → (and <statement> <expression>)

The only exception to this interpretation of GRAMMAR FILE entries occurs if either the word CROSSOVER or MUTATE appears as the first element of a list. In this case, GERDS treats the expression as a **specification list** defining the symbols allowed to serve as loci for the operator in question. In other words, the list

(CROSSOVER RULE STATEMENT EXPRESSION PREDICATE ATTRIBUTE) would restrict crossover to accept as a locus only those productions whose left-hand sides began with one of the five symbols RULE, STATEMENT, EXPRESSION, PREDICATE or ATTRIBUTE. Specification lists are optional, and as will be seen in later chapters, are used only rarely as a method of encapsulating rule subexpressions.

The next step of the algorithm is **create-population**, which uses the problem grammar to generate new population members. It first makes a call to **create-genotype**, which arbitrarily constructs a derivation trace beginning with a selected production locus. In the case of **create-population**, this locus is always 1. The **create-phenotype** module, in turn, constructs the rule structure corresponding to this new trace.

GERDS then enters a relatively straight-forward execution loop which closely follows the traditional GA paradigm. First, a fitness function is applied to the population in order to determine individual rule merit. FITNESS FILE contains the code for the user-defined function named "fitness" for each experiment. A default function is also provided, which simply uses the Lisp **eval** statement in determining a phenotype's value. The **copy-population** module then saves this evaluated population, while **statistics** sends information about each generation to the selected OUTPUT FILE.

Finally, the three basic genetic operators are applied to the population. The **reproduction** module copies structures chosen by roulette wheel selection into the next generation. The recombination operators **crossover** and **mutation** are then applied. Both operators make use of **parity-count**, which returns the subtrace beginning with a selected production locus. The **mutation** module also applies **create-genotype** and **create-phenotype** in order to generate new mutated subrules. After Max generations have been created and evaluated, program execution halts.

A complete listing of the GERDS package is presented in Appendix A to this dissertation.

3.4 Related Work

3.4.1 Cramer

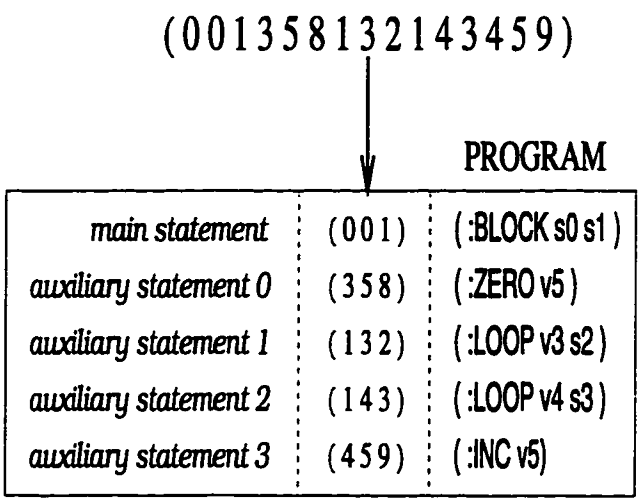
Cramer's[50] work marked an important turning point in the development of rule-based genetic algorithms. Although still using a binary representation scheme, Cramer interpreted these strings to be integers in a simple "number-string" language for generating short sequential computer functions. This technique was used successfully to produce two-input, single-output multiplication functions.

JB, the language Cramer first devised, was a variation of the algorithmic language PL, and consisted of the operator set shown in Figure 3.18. Programs in JB were simple lists of integers, divided into statements of length three. Extra integers at the ends of :ZERO and :INC statements were simply ignored during program execution. The first statement in a JB program was defined as the *main statement*. Subsequent operations were known as *auxiliary statements*. For example, Figure 3.19 depicts the phenotype for the JB function to calculate $v5 = v3 * v4$.

Despite the fact that Cramer devised a way to encode arbitrary computer programs into

INDEX	OPERATOR	FUNCTION
0	(:BLOCK statement statement)	performs first statement before second
1	(:LOOP variable statement)	performs statement variable times
2	(:SET variable variable)	sets first variable to second variable
3	(:ZERO variable)	sets variable to zero
4	(:INC variable)	adds one to variable

Figure 3.18: Cramer's JB Operator Set



main statement	(001)	(:BLOCK s0 s1)
auxiliary statement 0	(358)	(:ZERO v5)
auxiliary statement 1	(132)	(:LOOP v3 s2)
auxiliary statement 2	(143)	(:LOOP v4 s3)
auxiliary statement 3	(459)	(:INC v5)

Figure 3.19: A Sample JB Program for Multiplication

a binary format, there were two major problems with the straight-forward use of the JB language. First of all, since the semantic positioning of each integer was extremely sensitive to change, mutation could easily destroy an entire program. Second, JB programs were strongly epistatic in nature, and therefore not well-suited to crossover. In other words, the standard genetic operators would not work on JB programs.

In order to overcome this obstacle, Cramer created *TB*, a modified version of the JB language, in order to take advantage of the "implicit tree-like nature or JB programs". In TB, auxiliary statements were no longer used. Instead, when a statement was initially generated, all other required statements were immediately created and recursively expanded. The sample multiplication program from above would therefore have the form

(0 (3 5) (1 3 (1 4 (4 5))))

in TB.

Mutation in Cramer's TB-language system was severely limited. Only statements located at the leaves of program trees could be altered and still preserve TB structure. Therefore, mutation was restricted to the :INC, :SET and :ZERO operators. Crossover was similarly limited. Single statements could not be exchanged, as subtrees were treated as indivisible units. Despite these limitations and the narrow applicability of the representation scheme, Cramer's work with number-string programs was an important early attempt at using the GA paradigm on higher-order structures such as computer programs.

3.4.2 Bickel and Bickel

As part of their investigation in using genetic techniques to create expert systems, Bickel and Bickel[24] developed the *GENES* program. Each member of the GENES population was an *expert*, a linear list of simple condition-action rules. The number and length of these rules were randomly determined using several initializing parameters. One possible rule in a GENES expert might be

$$\text{(IF (NOT ((C1 = 2) AND (C4} \leq \text{3)) OR (C2} \geq \text{12)) THEN A1)}$$

Mutation was performed on a single rule within each expert, and required the use of auxiliary tables of information about operators, operands and conditions and actions. Via table lookup, the relational operator OR could be mutated to an AND, the boolean operator $>$ could be changed to $\leq$, or an operand such as 32 could be changed to 212. The mutation operator in GENES also allowed for the removal or addition of a rule from an individual expert.

Bickel and Bickel also used a simple **inversion** operator. It functioned by randomly choosing two points along the length of an expert rule set. The list of rules between these two points were then spliced out of the rule list, reversed, and finally re-inserted. Figure 3.20 shows the effect of the inversion operator on an arbitrary rule set. Since using inversion changed the order of rule evaluation within a given expert, GENES was able to alter the priority associated with each decision rule.

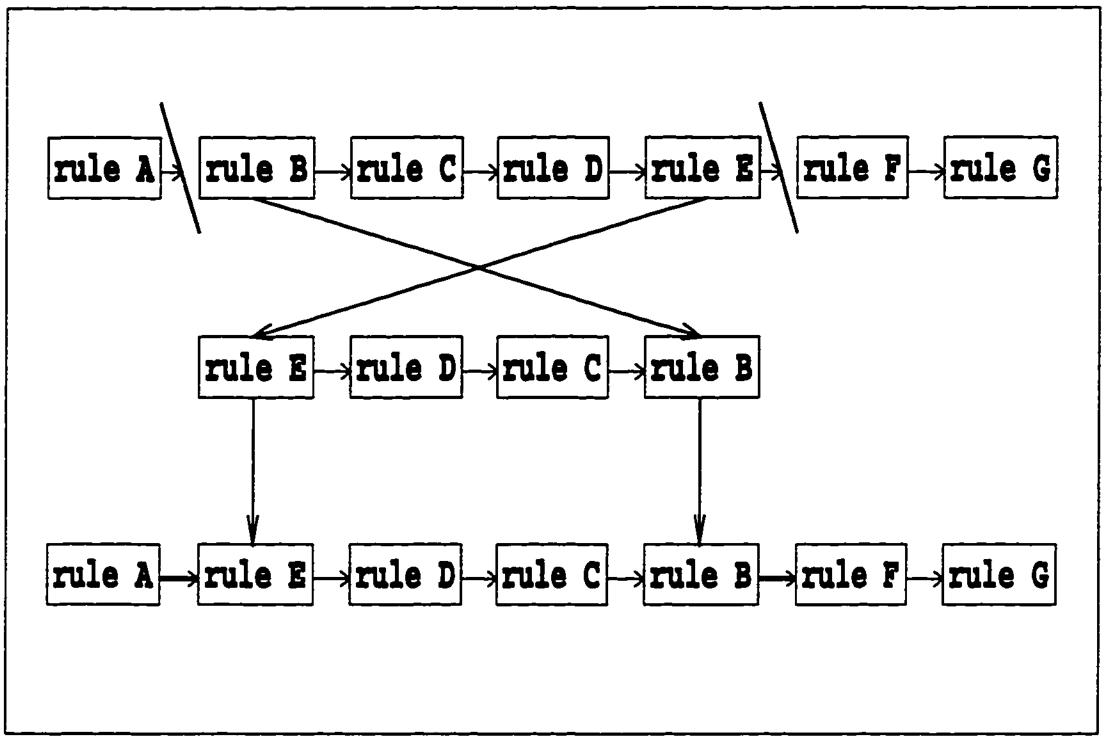


Figure 3.20: The Inversion Operator in GENES

In Bickel and Bickel's system, crossover was allowed only at points between rules. Since the length of the rule sets varied, one locus for crossover was selected by taking the modulus of the shorter expert, and the other by taking the modulus of the longer. If both loci turned out to be less than the length of the shorter expert, then a *double crossover* occurred. As depicted in Figure 3.21, double crossover operated by exchanging the sublist of rules from both experts found between these two loci, thereby allowing both experts to retain their original size. If only one locus was less than the length of the shorter rule list, single crossover exchanged only the tail end of the experts.

The GENES model was tested on a small scale with some success; however, the multiple auxiliary tables required by GENES in order to perform mutation on individual rules severely limited the generality of this approach. It also restricted the size and scope of prob-

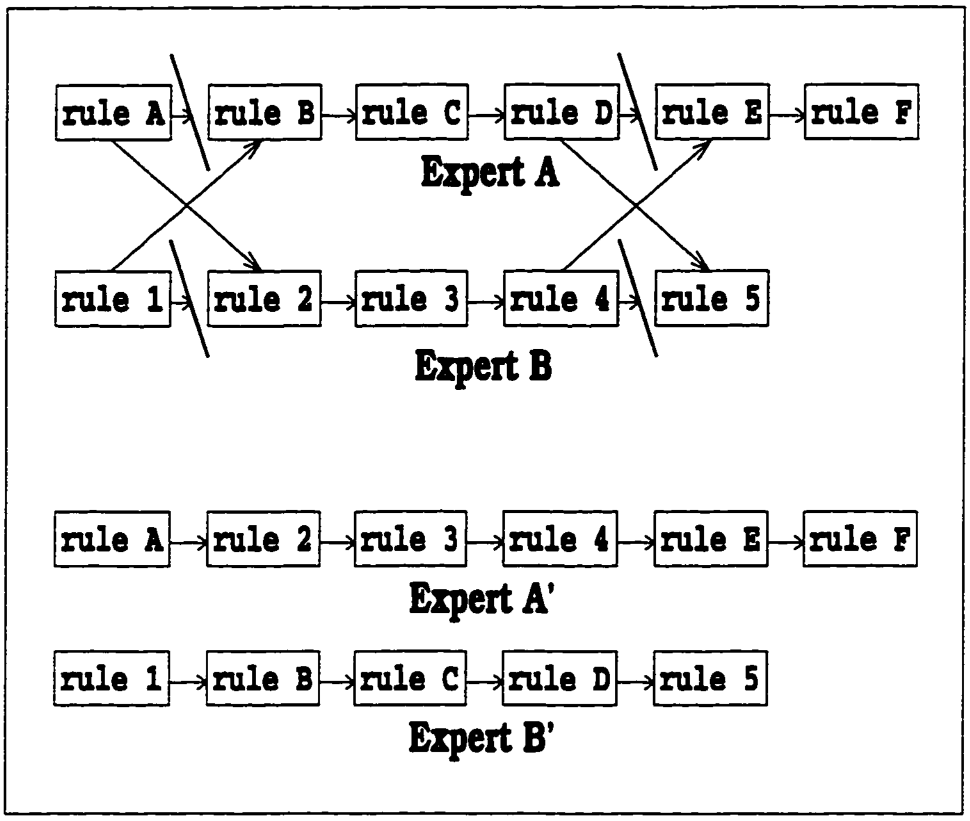


Figure 3.21: The Double Crossover Operator in GENES

lems which could be addressed by the system. In addition, the inability of the crossover operator to exchange information between individual rules resulted in each complex rule structure being treated as a simple gene in an "expert" chromosome.

3.4.3 Fujiki and Dickinson

Fujiki and Dickinson[83] explored the use of genetic algorithms for discovering Lisp source code for solving the Prisoner's Dilemma problem. The programs generated by their study were based on sets of productions for generating Lisp cond expressions like the one shown in Figure 3.22 Using this grammar, Fujiki and Dickinson's system randomly created a population of cond expressions. Productions were separated into two categories: those that

```
          index → (cond (t action))
          index → (cond index-cond-term (t action))
index-cond-term → (logical action)
index-cond-term → (logical action) index-cond-term
         action → 1
         action → 2
        logical → (not logical)
        logical → (l-op logical logical)
        logical → (equal nround 1)
        logical → (equal nround 10)
        logical → (equal op-play action)
        logical → (equal op2-play action)
        logical → if-any
           l-op → and
           l-op → or
         if-any → past-def-op
         if-any → past-coop-op
```

Figure 3.22: Fujiki and Dickinson's Grammar for the Prisoner's Dilemma

had only terminal symbols on their right-hand side, and those that had one or more variable symbols.

The length of the generated **cond** was controlled by a selection algorithm in which the probability of choosing productions from the terminal symbol category was increased as the length of the expression grew. Each condition-action pair in the **cond** was considered to be one individual piece of information to be used by their GA. These were never examined by recombination operators. Production sets were used only for the creation of the initial population.

Fujiki and Dickinson's crossover operator worked by dividing the parent **cond** expressions at two random points between condition-action pairs. New **conds** were then created by combining the first part of one expression with the second part of the other expression. In order to ensure that the final condition-action pair at the end of the **cond** contained the only condition in the expression to always evaluate true, the last member of the list could not be exchanged.

The mutation operator worked by randomly removing one condition-action pair from an expression and replacing it with a new pair created by a separate production set. As with crossover, mutation of the final **cond** element was restricted. An inversion operator was also used by Fujiki and Dickinson which reversed the order of every condition-action pair in the expression excepting the last.

Although Fujiki and Dickinson applied a grammar to the task of genetic search, the system offered very little overall structure to the generated rules. The size and shape of the generated expressions were instead controlled by a problem-specific external selection algorithm. In addition, the grammar was used only to generate complete condition-action pairs. Rules were therefore treated by necessity as single genes. As a result, the applicability of the crossover and mutation operators was severely curtailed.

3.4.4 Grefenstette

Grefenstette[108, 110] and others investigated the use of genetic algorithms for solving *sequential decision tasks*, processes in which a decision-making agent iteratively interacts with a discrete-time dynamic system. Such a system moves from state to state as a result of performing one of a finite number of actions. These actions are in turn selected by applying the agent's decision making rules.

For their research, Grefenstette, Ramsey and Schultz developed a specialized genetic algorithm called *SAMUEL* and applied it to the sequential decision task known as the *Evasive Maneuvers (EM)* problem. The tactical goal of the EM problem was to maneuver a plane in such a way as to avoid its being hit by an approaching missile. The missile was able to track the motion of the plane and steer toward its anticipated course. The speed of the missile, however, decreased for each course correction undertaken. If the missile speed dropped below a certain threshold, the plane escaped destruction.

Unlike traditional genetic algorithms, the SAMUEL system was designed to operate on a high-level representation. Each rule in the system had the form

if (and c_1 c_2 .. c_n) then (and a_1 a_2 .. a_m)

where each c_i was a condition and each a_j an action. Conditions for the EM problem were based upon the states of six sensors which gathered information about the current tactical situation. These sensors and the information they provided are listed in Figure 3.23. The

SENSOR	MEANING	RANGE	UNIT	TYPE
last-turn	current turning rate of plane	-180 to 180	45	linear
time	time since detection of missile	0 to 19	1	linear
range	missile's current distance from plane	0 to 1500	100	linear
bearing	direction from plane to missile	1 to 12	1	cyclic
heading	missile's direction relative to plane	0 to 350	10	cyclic
speed	missile's current speed	0 to 1000	50	linear

Figure 3.23: Sensors used by SAMUEL for the *EM* Problem

range column shows the extreme upper and lower sensor settings, while unit refers to the separation between discrete values within each range. For example, the sensor last-turn could take on any of the nine values in $[-180, -135, -90, -45, 0, 45, 90, 135, 180]$.

The form of each condition depended on the type of sensor it contained. *Linear sensors* took on linearly ordered numeric values such as **time**. Conditions for this sensor type specified legal upper and lower bounds for the sensor value. The condition

(speed 100 230**)**

would therefore match sensor values in which $100 \leq$ **speed** ≤ 230. *Cyclic sensors*, on the other hand, took on cyclicly ordered numeric values such as **bearing**. Since the next "higher" value of **bearing** 12 was **bearing** 1, there were no absolute endpoints. Therefore, conditions for cyclic sensors could take on any legal values. Thus, the condition

(heading 340 30**)**

would match any of the sensors values in $[340, 350, 0, 10, 20, 30]$.

Each action of an EM rule consisted of a single control variable **turn**, which ranged in value from -180 to 180 in increments of 45. Its application led to adjustments in the course of the airplane during the next time step or *episode*. An EM problem was divided into twenty episodes that began when a missile was detected and prematurely ended when either the plane was hit or the missile was evaded. The fitness function used by the EM problem was

$$f_x = \begin{cases} 1000 & \text{if plane escapes} \\ 100t & \text{if plane is hit at time } t \end{cases}$$

The aim of the SAMUEL system was the discovery of a *tactical plan*, a complete set of decision rules for the EM problem. For this reason, SAMUEL adopted a different approach to genetic search, applying recombination operators at the level of the tactical plan rather than the individual rule. Initially, each tactical plan in the population consisted of nine *maximally general rules* in which every sensor condition contained both the extreme upper and lower sensor values. A maximally general rule could therefore be interpreted as *for any sensor settings, turn Y*

where *Y* was one of the nine possible values for the turn control variable.

In order to create plausible new rules from these initial tactical plans, SAMUEL used a genetic operator called *specialize*. It was invoked whenever a maximally general rule was fired leading to a successful evasion, and there was still space in the tactical plan for an additional rule. By applying specialize, a new rule was created in which every condition was modified to cover only half the legal values for the sensor. The starting point to the condition subrange was calculated by finding the midpoint between the sensor state and its nearest extreme sensor value. Figure 3.24 shows the specialize operator in greater detail.

maximally general rule

```
if (and (last-turn -180 180)
        (time 0 19)
        (range 0 1500)
        (bearing 1 12)
        (heading 0 350)
        (speed 0 1000)) then (turn 90)
```

sensors	
last-turn	90
time	4
range	600
bearing	3
heading	60
speed	700

half distance from extreme	interval start	half	start + half
(180-90)/2=45	90+45=135	-180	135-180=-45
(4-0)/2=2	0+2=2	10	2+10=12
(600-0)/2=300	0+300=300	750	300+750=1050
(3-1)/2=1	1+1=2	6	2+6=8
(60-0)/2=30	0+30=30	180	30+180=210
(1000-700)/2=150	1000-150=850	-500	850-500=350

rule after specialization

```
if (and (last-turn -45 135)
        (time 2 12)
        (range 300 1050)
        (bearing 2 8)
        (heading 30 210)
        (speed 350 850)) then (turn 90)
```

Figure 3.24: SAMUEL's specialization operator

The sensor **speed**, for example, might have a value 700. The nearest extreme sensor value is the maximum of 1000. The halfway point between 1000 and 700 would therefore be 850, which would serve as the maximum bound on the new specialized condition. The minimum would be found by taking half of the speed sensor's range (1000/2 = 500) and subtracting it from the halfway point, yielding the new specialized condition

(speed 350 850)

Each specialized rule was assumed to be plausible, since its action was known to be successful in at least one situation: the sensor states that triggered rule specialization.

Crossover in SAMUEL operated by assigning each rule from two parent tactical plans to one of two offspring plans. SAMUEL was able to examine traces of the parents' previous performance when distributing rules to the children. Figure 3.25 shows two traces and the new tactical plans created through crossover. A sequence of rules in a parent trace which led to a successful missile evasion was treated as a unit whenever possible in order to increase the likelihood that productive behavior would be inherited. The crossover operator was restricted so that no plan received duplicate copies of a rule. In other words, if $R_{1,3}$ and $R_{2,9}$ were identical rules, crossover would ensure that they were each distributed to different tactical plans.

SAMUEL's final genetic operator, mutation, introduced a new rule to a tactical plan by making a random change to an existing one. For example, mutation might alter the condition (time 3 7) to (time 3 11) or it might change the action from (turn −90) to (turn 45). The new value produced by mutation was chosen from the set of legal values for each sensor and control variable.

The SAMUEL program was highly adapted to problems involving discrete numeric values of limited range. In addition, the unique crossover and specialization operators were developed with the intent of "plan" discovery. Although it was successfully applied to the

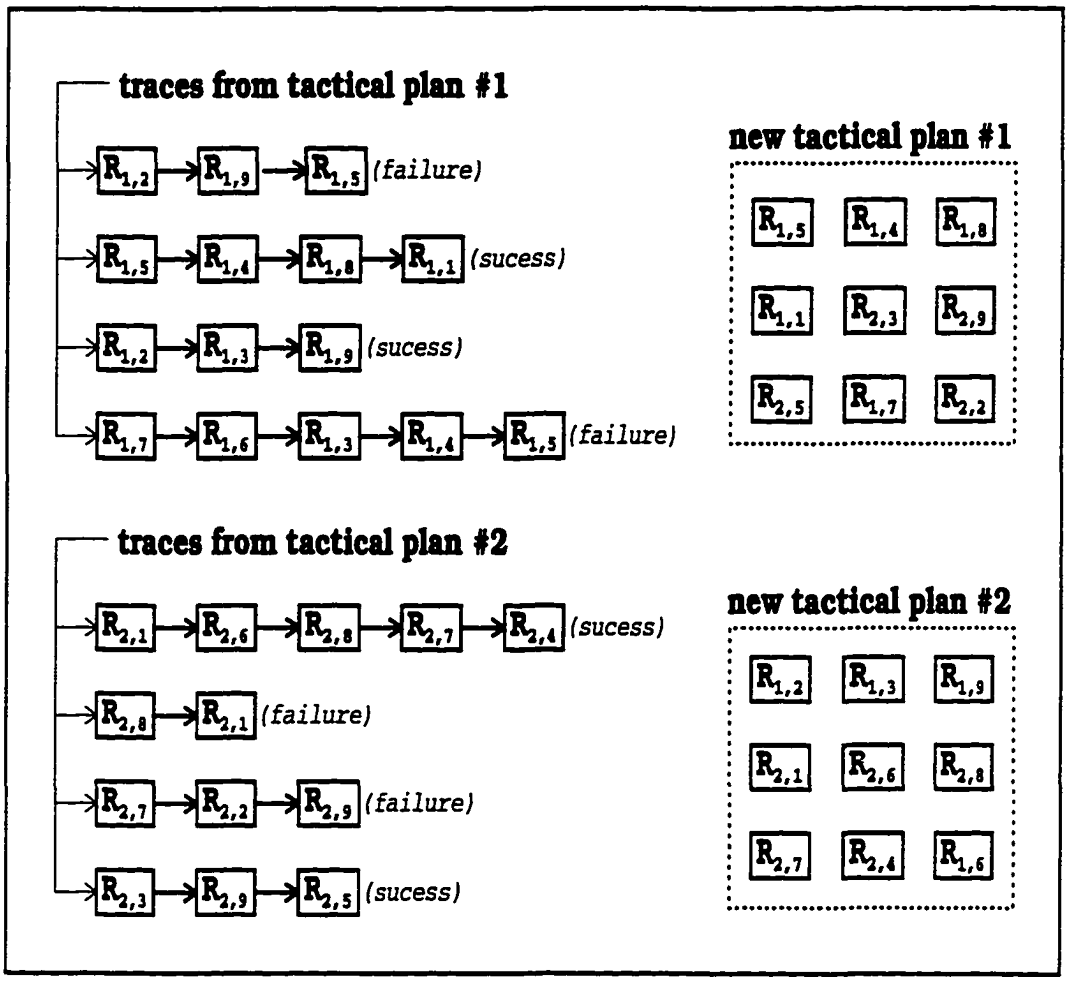

traces from tactical plan #1
$R_{1,2}$ $R_{1,9}$ $R_{1,5}$ (failure)
$R_{1,5}$ $R_{1,4}$ $R_{1,8}$ $R_{1,1}$ (sucess)
$R_{1,2}$ $R_{1,3}$ $R_{1,9}$ (sucess)
$R_{1,7}$ $R_{1,6}$ $R_{1,3}$ $R_{1,4}$ $R_{1,5}$ (failure)
new tactical plan #1
$R_{1,5}$ $R_{1,4}$ $R_{1,8}$
$R_{1,1}$ $R_{2,3}$ $R_{2,9}$
$R_{2,5}$ $R_{1,7}$ $R_{2,2}$
traces from tactical plan #2
$R_{2,1}$ $R_{2,6}$ $R_{2,8}$ $R_{2,7}$ $R_{2,4}$ (sucess)
$R_{2,8}$ $R_{2,1}$ (failure)
$R_{2,7}$ $R_{2,2}$ $R_{2,9}$ (failure)
$R_{2,3}$ $R_{2,9}$ $R_{2,5}$ (sucess)
new tactical plan #2
$R_{1,2}$ $R_{1,3}$ $R_{1,9}$
$R_{2,1}$ $R_{2,6}$ $R_{2,8}$
$R_{2,7}$ $R_{2,4}$ $R_{1,6}$

Figure 3.25: Crossover between tactical plans in SAMUEL

EM problem with 90% accuracy, the domain-specific nature of the overall system severely limits the wide-spread application of the paradigm.

3.4.5 Koza

Koza's[149, 152] work on *genetic programming* marked a significant advance in the application of the genetic paradigm to higher representation schemes. In Koza's work, each member of the population was a "program" which corresponded to a simple Lisp S-expression. This expression was constrained to contain only members of a set F of functions. These functions in turn could operate only over members of a terminal set *T* of constants and variables. The genetic programming methodology required that there be closure between the two sets *F* and *T*; in other words, each function had to accept as an argument any value returned by any other function or terminal. The S-expressions contained in the initial population were randomly generated using members of *F* and *T*. The structure and size of these programs were controlled by the system's many variables and parameters.

Crossover, as well as the other recombination operators developed by Koza, functioned by treating each S-expression program as a tree structure. The first step in performing crossover was the selection of a random point within each of two parent programs. These points were heads of the two subtrees to be exchanged through crossover. For example, the S-expressions

```
            ( OR ( NOT D1 ) ( AND D0 D1) )
( OR ( OR D1 ( NOT D0 ) ) ( AND ( NOT D0 ) ( NOT D1 ) ) )
```

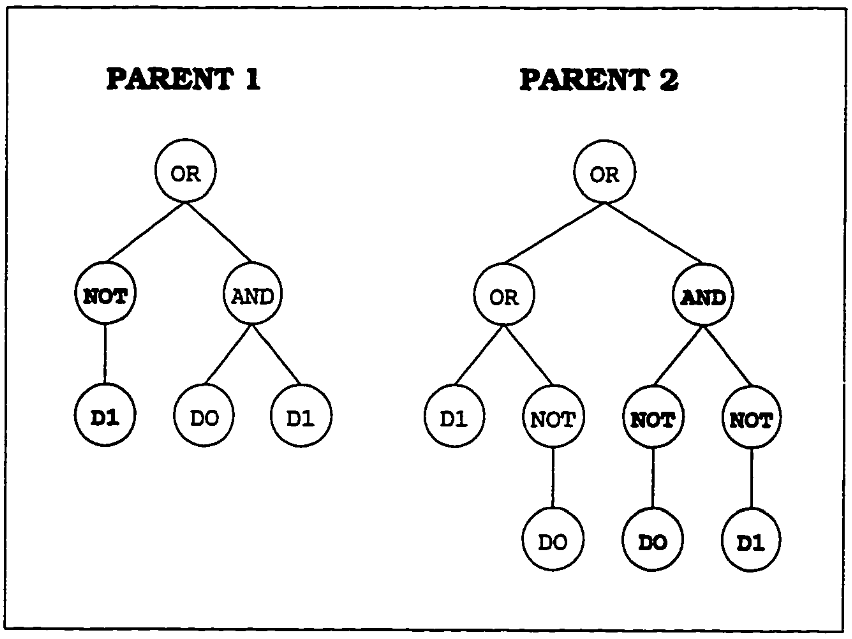


Figure 3.26: Examples of Genetic Programs

could be represented by the two program trees shown in Figure 3.26, where the expressions (NOT D1) and (AND (NOT D0) (NOT D1)) have been selected as the points for

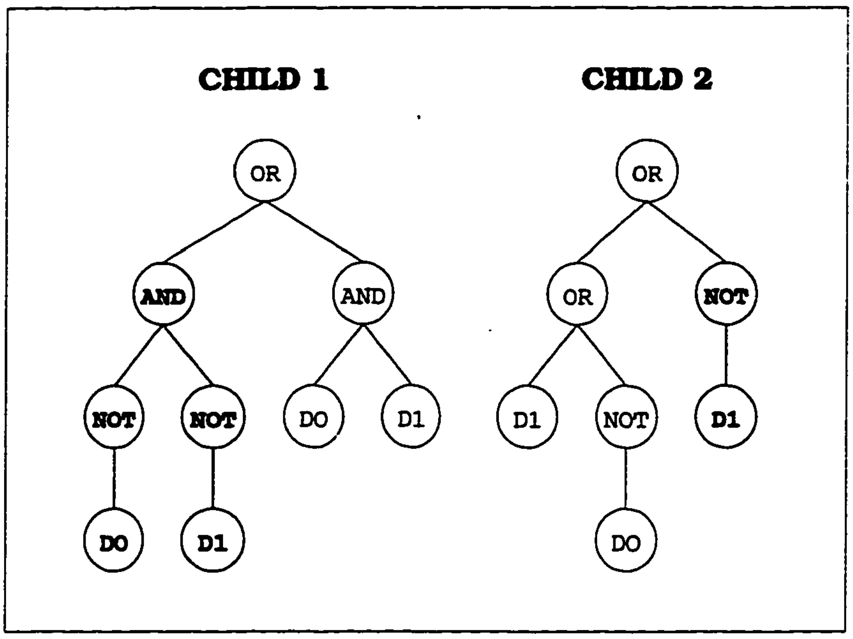


Figure 3.27: Crossover Between Genetic Programs

crossover. As depicted in Figure 3.27, these two subtrees were then exchanged between parent programs in order to produce the two new programs

```
( OR ( AND ( NOT D0 )( NOT D1 ) ) ( AND D0 D1) )
( OR ( OR D1 ( NOT D0 ) ) ( NOT D1 ) )
```

Because all members of an S-expression were restricted to returning values of the same type, crossover between trees always produced two legal genetic programs. As with the standard GA, the crossover operator, in conjunction with reproduction, was the workhorse of the search process.

Koza's mutation operator functioned by arbitrarily selecting a random point in the S-expression tree. The entire subtree beginning at this node was then removed and replaced by a new subtree which was randomly generated using the same control parameters that

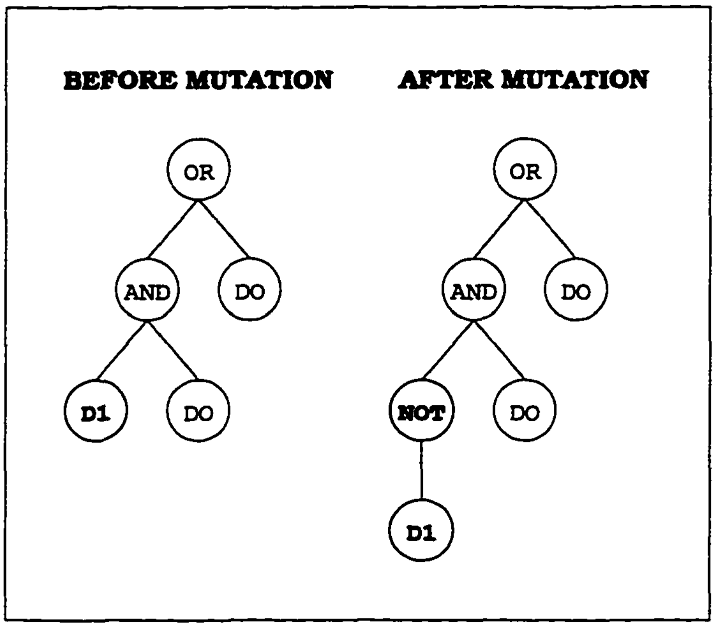


Figure 3.28: The Mutation Operator in Koza's Genetic Programming

guided the development of the initial population. Figure 3.28 depicts the mutation of the program

```
( OR ( AND D1 DO ) DO )
```

where D1 has been selected as the mutation point. This node is removed and then replaced by a new subtree (NOT D1). Although the application of his mutation operator resulted in valid new programs, Koza very rarely included it as part of an experiment. This was because his crossover operator acting alone was capable of altering any node within the "free-form" genetic program tree.

Koza's work also included a *permutation* operator which served to shuffle the positions of all children of a parent. As such, it shared a similar function to the inversion operators discussed earlier. For many genetic programs such as the boolean function example, permutation of a tree would not result in a new program. However, for functions such as division, the importance of parameter ordering becomes apparent. Figure 3.29 shows the

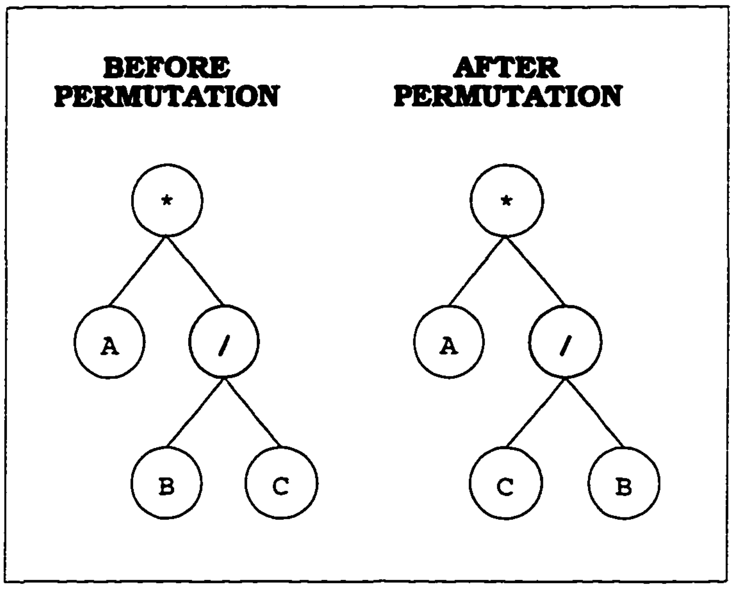


Figure 3.29: The Permutation Operator in Koza's Genetic Programming

permutation of the program tree A * B / C, where the division operator has been chosen as the permutation point. The position of the child nodes B and C are then rearranged, resulting in the new program A * C / B.

Due to the unconstrained shape of genetic program trees, Koza's work required the application of an *editing* operator to simplify S-expressions. To perform editing, a set of rules was recursively applied to a program tree. This rule set contained both domain-independent and domain-specific simplification routines. Regardless of the domain to which the genetic programming paradigm was applied, a function containing only constants as subtrees could always be replaced by its functional evaluation. DeMorgan's laws could be applied to simplify S-expressions in boolean domains, whereas expressions such as $A*1$ could be replaced by A in mathematical applications. The editing operator could be applied at any time during genetic programming evolution.

Koza successfully applied his genetic programming methodology to a wide variety of applications such as sequence induction and multiple symbolic regression. There are many domains, however, in which his paradigm would be unsuitable. The system was designed to operate only upon functional programs in the form of Lisp S-expressions. As demonstrated by the wide success of the traditional bitstring approach to GAs, not all structures well-suited to genetic search are self-evaluating.

Furthermore, the closure property of F and T required that all functions return the same type of values. There are many problems, however, which cannot be adequately described by only one data type. Special care had to be taken even when this was the case. For example, functions such as division and logarithm had to be "protected" over the range of integers so that the expressions $A/0$ and $\log(-3)$ would return some integer value.

The non-structured form of the genetic program was another inherent source of difficulty. Many problem domains have structures which follow rigid guidelines where positioning of elements is important. In addition, genetic programming had an unfortunate tendency to find large "ugly" solutions containing redundant information. Although the application of the editing operator partially remedied this problem, it was extremely time-consuming and had to be tailor-made for each domain.

Perhaps most importantly, crossover between genetic programs worked against population **convergence**. In the traditional GA approach, when crossover is applied to two identical bitstrings, the resulting children are guaranteed to be copies of the parents. In this manner, highly fit individuals propagate over the course of several generations. In genetic programming, on the other hand, when two identical parents mated, two random subtrees were exchanged. For example, if crossover is performed between identical S-expressions

```
( OR ( NOT D1 ) ( AND D0 D1) )
( OR ( NOT D1 ) ( AND D0 D1) )
```

where (NOT D1) and D0 have been selected as crossover points, the two new S-expressions

```
( OR D0 ( AND D0 D1) )
( OR ( NOT D1 ) ( AND ( NOT D1 ) D1) )
```

are created. Thus, unless the same nodes in both parents were selected as the locus, crossover between duplicate parents in genetic programming would result in entirely new S-expressions. The grammar-based methodology presented in this dissertation, on the other hand, imposes an underlying structure to rules in the search space and thereby fosters population convergence.

Chapter 4

Rule and Bitstring Comparison

4.1 Syntactic Representation of Bitstrings

Throughout the previous discussions of Sections 2.1, 2.2 and 3.1, a 6-bit chromosome was used to represent integers in the interval $[0, 63]$. Intuitively, each of these binary strings is a member of the language

$$\mathcal{B} = (1 + 0)^6$$

As previously stated, the syntactic genetic algorithm is capable of generating any set of structures defined by an arbitrary language. As shown in Figure 4.1, it is therefore a

```
1: <bitstring> → <gene> <locus1>
2:    <locus1> → <gene> <locus2>
3:    <locus2> → <gene> <locus3>
4:    <locus3> → <gene> <locus4>
5:    <locus4> → <gene> <locus5>
6:    <locus5> → <gene>
7:      <gene> → <allele>
8:    <allele> → 0
9:    <allele> → 1
```

Figure 4.1: 6-bit Binary String Grammar

relatively simple task to construct a grammar G describing members of $\mathcal{B}$.

As an example of how G creates rules, consider S = **001101** $\in \mathcal{B}$. Under the binary representation, string S is equivalent to a chromosome with the genotype 001101 and the phenotype thirteen. For the syntactic GA, string S corresponds to a rule with 001101 acting as phenotype and the derivation trace

$$\begin{array}{l} \mathcal{X} = 1\ 7\ 8\ 0\ 0\ 2\ 7\ 8\ 0\ 0\ 3\ 7\ 9\ 0\ 0\ 4\ 7\ 9 \\ \phantom{\mathcal{X} = }0\ 0\ 5\ 7\ 8\ 0\ 0\ 6\ 7\ 9\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \end{array}$$

serving as genotype. The remaining $6^2 - 1$ members of $\mathcal{B}$ can be generated from G in a similar manner.

4.1.1 Syntactic Emulation of Binary Crossover

Binary crossover operates by exchanging all genes occurring after an arbitrarily selected crossover site in two parent strings. As stated in Section 2.2, for binary chromosomes of length l there exist $l - 1$ different interchange sites. Figure 4.2 depicts the five possible

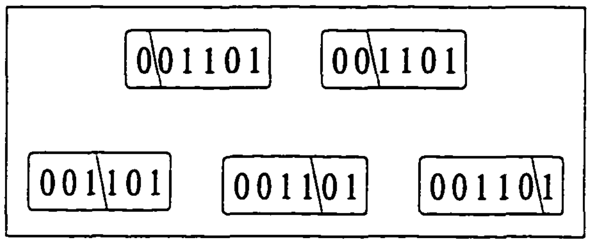


Figure 4.2: Possible Crossover Sites for Bitstring 001101

crossover loci (indicated by the "\" marks) for the 6-bit string 001101.

Rule crossover for the syntactic genetic algorithm, on the other hand, exchanges derivation subtraces headed by a "randomly" determined production common to both parents. Since the derivation trace of every rule derived from G contains exactly 18 productions, only six of which are members of the terminal label set TLS_G of G, there exist twelve possible crossover sites in each rule: 1, 2, 3, 4, 5, 6, and the six instances of 7. Thus, rule crossover allows for a wider variety of genetic interchange than its binary counterpart. It is possible, however, to restrict the application of syntactic crossover by using a specification list.

As detailed in Section 3.3, when a crossover specification list is associated with a gram-

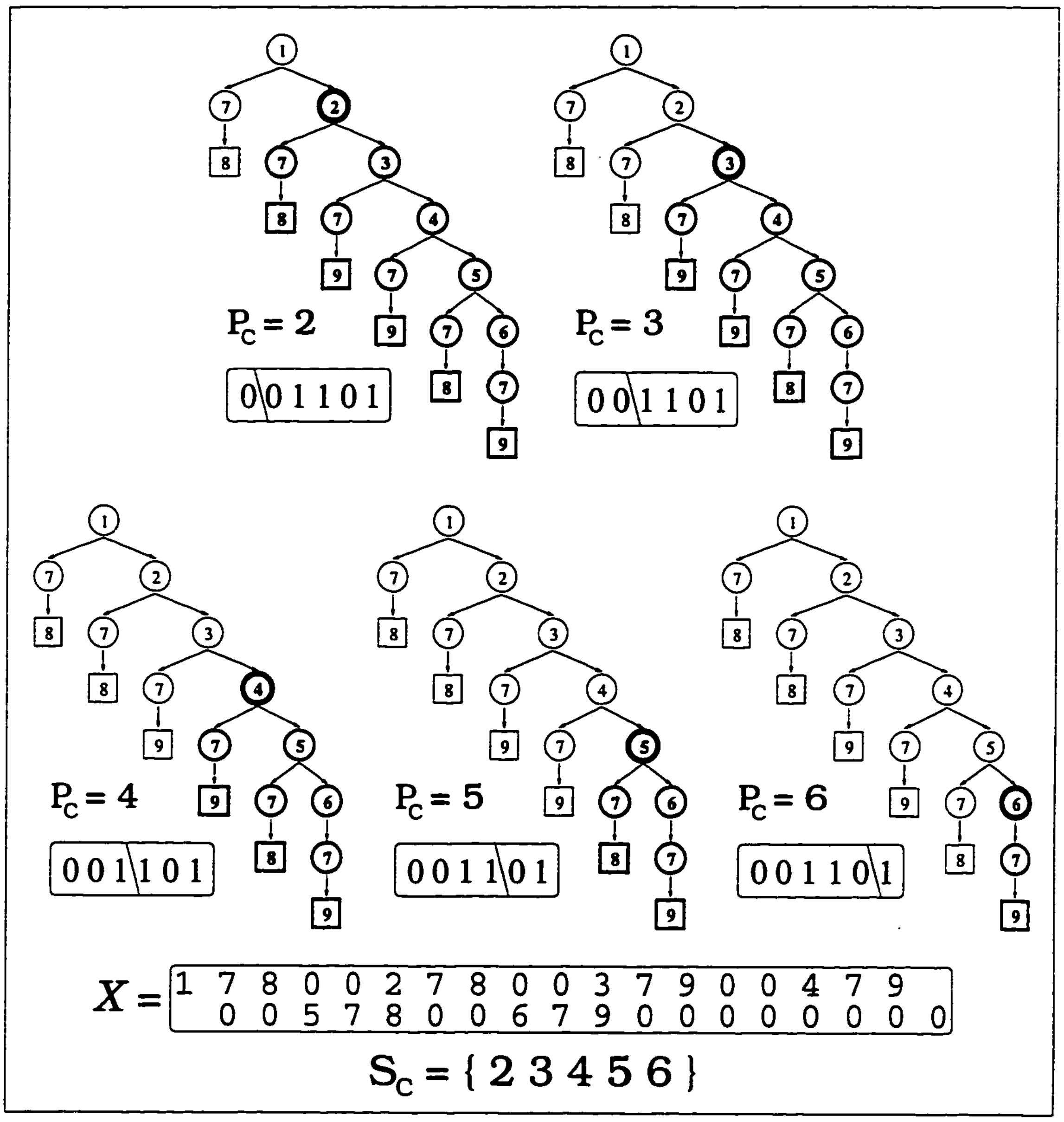

$P_c = 2$
0\01101
$P_c = 3$
00\1101
$P_c = 4$
001\101
$P_c = 5$
0011\01
$P_c = 6$
00110\1
X = 1 7 8 0 0 2 7 8 0 0 3 7 9 0 0 4 7 9 0 0 5 7 8 0 0 6 7 9 0 0 0 0 0 0 0 0
$S_C = \{2\ 3\ 4\ 5\ 6\}$

Figure 4.3: Possible Crossover Sites for Rule 001101

mar, the selection of a crossover production label P_C is limited to productions whose left-hand side appear in that list. Thus, if

$$\mathsf{L_C} = (\text{ CROSSOVER LOCUS1 LOCUS2 LOCUS3 LOCUS4 LOCUS5 })$$

is used as a crossover specification list for G, only members of the set

$$\mathsf{S_C} = \{2\ 3\ 4\ 5\ 6\}$$

can be chosen as the crossover production label. Since all elements of $\mathsf{S_C}$ appear exactly once in each derivation trace, five possible crossover subtraces exist for each rule. As illustrated in Figure 4.3, there is a one-to-one correspondence between these derivation subtraces and the five crossover loci for the binary operator. Thus, the syntactic crossover of rules in G using specification list $\mathsf{L_C}$ is equivalent to binary crossover performed on 6-bit chromosomes.

4.1.2 Syntactic Emulation of Binary Mutation

Binary mutation functions by changing the value of one or more genes along the length of the chromosome. In effect, this amounts to taking the complement of every bit with probability p_m. Figure 4.4 depicts the six possible mutation loci (indicated by the surrounding "□"

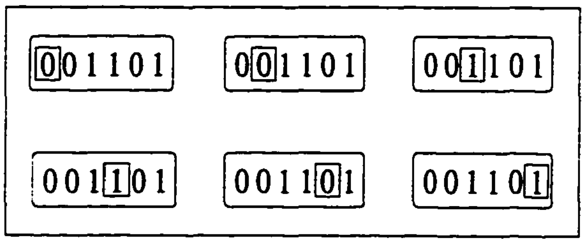


Figure 4.4: Possible Mutation Sites for Bitstring 001101

symbols) for the example chromosome 001101.

In contrast, syntactic mutation operates by first removing the derivation subtrace headed by a mutation production label P_M, and then replacing it with a newly constructed derivation subtrace headed by that same production. Rule mutation is applied with probability

p_m to every production in the derivation trace, with the exception of members of the terminal label set. Since every rule in G has a derivation trace containing 18 productions, of which only six are members of TLS_G, up to twelve different P_M may be mutated. As with syntactic crossover, however, this selection can be limited by using a specification list.

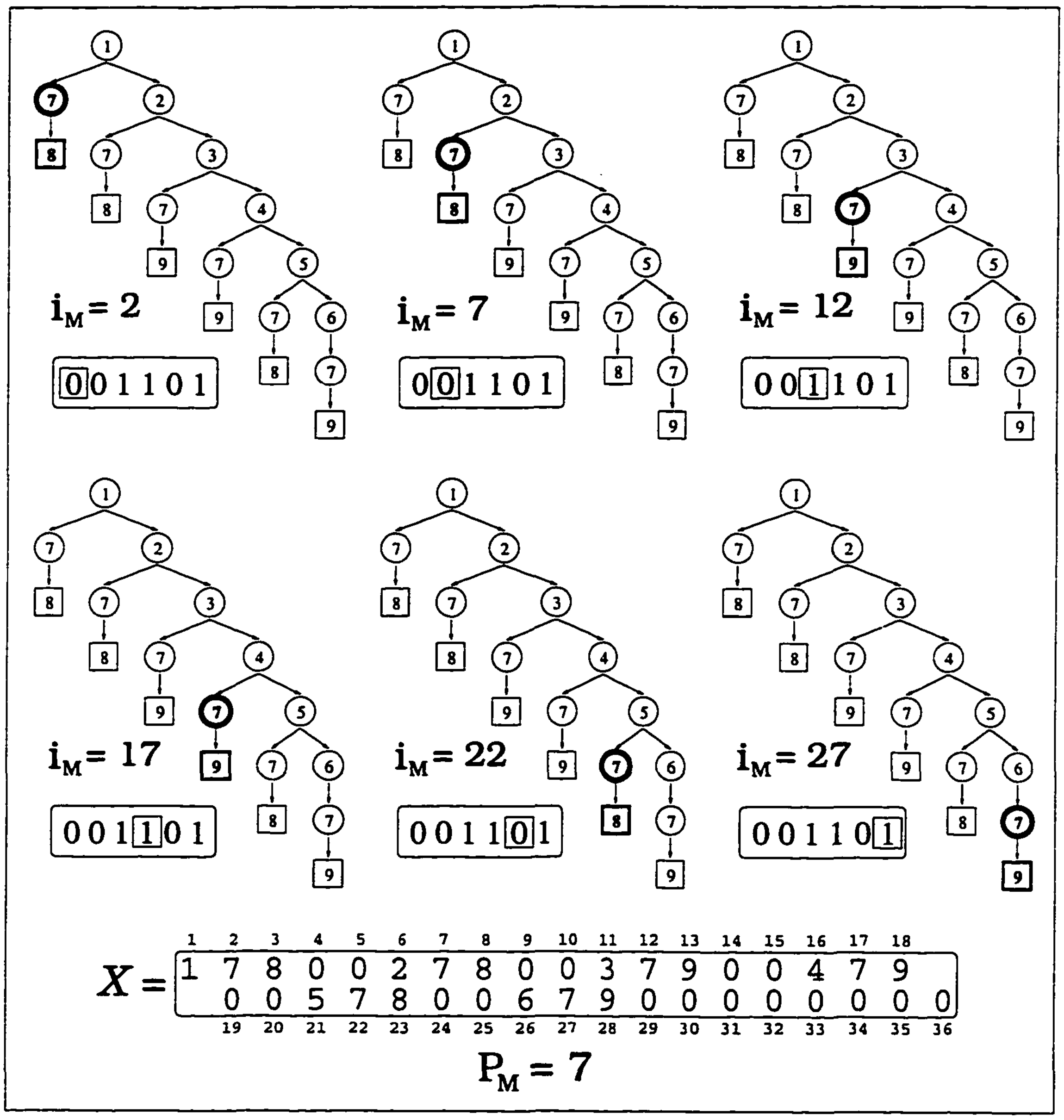


Figure 4.5: Possible Mutation Sites for Rule 001101

Thus, if the mutation specification list

$$\mathrm{L_M} = (\text{ MUTATION GENE })$$

is associated with grammar G, only members of the singleton set

$$\mathrm{S_M} = \{7\}$$

can be selected as a mutation production label. The derivation trace of every rule in G contains six instances of 7, each of which heads one of the two mutation subtraces

```
7 8 0 0
7 9 0 0
```

corresponding to rule subexpressions **0** and **1**, respectively. As depicted in Figure 4.5, there is a direct relationship between these derivation subtraces and the six potential mutation loci for the binary operator. Thus, the syntactic mutation of rules in G using specification list $\mathrm{L_M}$ is equivalent to the binary mutation of 6-bit chromosomes.

4.1.3 Syntactic Emulation of Binary Reproduction

Under roulette wheel selection, a binary chromosome s_j is copied into the subsequent generation with probability

$$p(s_j) = \frac{f(s_j)}{\sum_{i=1}^{N} f(s_i)}$$

where f is some fitness function and N is the population size. The syntactic representation presented in the dissertation also uses roulete wheel selection, and therefore reproduces rule r_j with probability

$$p(r_j) = \frac{f(r_j)}{\sum_{i=1}^{N} f(r_i)}$$

4.1.4 Equivalence of Syntactic and Binary Representation

The ability of the syntactic genetic algorithm to represent a 6-bit binary string can easily be generalized to

Observation 4.1 *Any problem whose parameter set can be encoded by a binary GA into chromosomes of length* L *can also be expressed in a syntactic GA by rules derivable from a grammar* G *of the form*

1:	<bitstring>	→	<gene> <locus$_1$>
2:	<locus$_1$>	→	<gene> <locus$_2$>
		⋮	
L-1:	<locus$_{L-2}$>	→	<gene> <locus$_{L-1}$>
L:	<locus$_{L-1}$>	→	<gene>
L+1:	<gene>	→	<allele>
L+2:	<allele>	→	0
L+3:	<allele>	→	1

with a crossover specification list G_C = (CROSSOVER LOCUS$_1$ LOCUS$_2$ .. LOCUS$_L$) *and a mutation specification list* G_M = (MUTATE GENE).

4.2 Binary Optimization

As a simple example of binary GA optimization, consider the task of finding the greatest

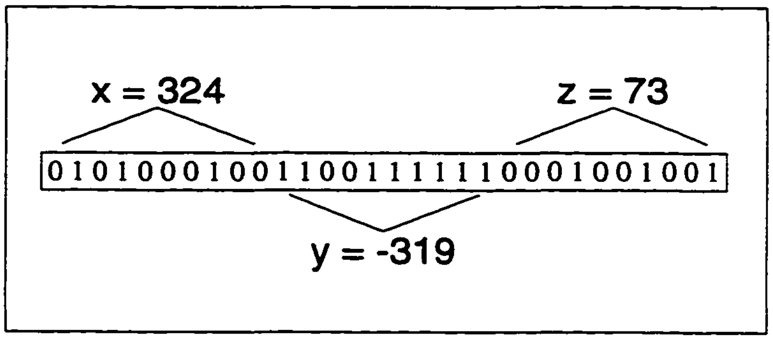


Figure 4.6: Encoding for Binary Optimization of $f_{x,y,z} = x^2 + y^2 + z^2$

functional value of the equation

$$f_{x,y,z} = x^2 + y^2 + z^2$$

in the interval $[-511, 511]$ for the variables x, y and z. One possible binary encoding for this problem is shown in Figure 4.6. Each independent variable in this example is represented in sign-magnitude notation.

The syntactic GA presented in this dissertation can be used to construct corresponding

1:	<bitstring>	→	<gene> <locus1>
2:	<locus1>	→	<gene> <locus2>
3:	<locus2>	→	<gene> <locus3>
4:	<locus3>	→	<gene> <locus4>
5:	<locus4>	→	<gene> <locus5>
6:	<locus5>	→	<gene> <locus6>
7:	<locus6>	→	<gene> <locus7>
8:	<locus7>	→	<gene> <locus8>
9:	<locus8>	→	<gene> <locus9>
10:	<locus9>	→	<gene> <locus10>
11:	<locus10>	→	<gene> <locus11>
12:	<locus11>	→	<gene> <locus12>
13:	<locus12>	→	<gene> <locus13>
14:	<locus13>	→	<gene> <locus14>
15:	<locus14>	→	<gene> <locus15>
16:	<locus15>	→	<gene> <locus16>
17:	<locus16>	→	<gene> <locus17>
18:	<locus17>	→	<gene> <locus18>
19:	<locus18>	→	<gene> <locus19>
20:	<locus19>	→	<gene> <locus20>
21:	<locus20>	→	<gene> <locus21>
22:	<locus21>	→	<gene> <locus22>
23:	<locus22>	→	<gene> <locus23>
24:	<locus23>	→	<gene> <locus24>
25:	<locus24>	→	<gene> <locus25>
26:	<locus25>	→	<gene> <locus26>
27:	<locus26>	→	<gene> <locus27>
28:	<locus27>	→	<gene> <locus28>
29:	<locus28>	→	<gene> <locus29>
30:	<locus29>	→	<gene>
31:	<gene>	→	<allele>
32:	<allele>	→	0
33:	<allele>	→	1

Figure 4.7: 30-bit Binary String Grammar

"bitstring rules" which can utilize the same encoding mechanism. Figure 4.7 shows a grammar for generating binary strings of length $l = 30$. As shown in Section 4.1, the specification lists

$$L_C = (\text{CROSSOVER LOCUS1 LOCUS2 LOCUS3} \ldots \text{LOCUS29})$$
$$L_M = (\text{MUTATE GENE})$$

must also be associated with this grammar in order to mimic binary GA performance.

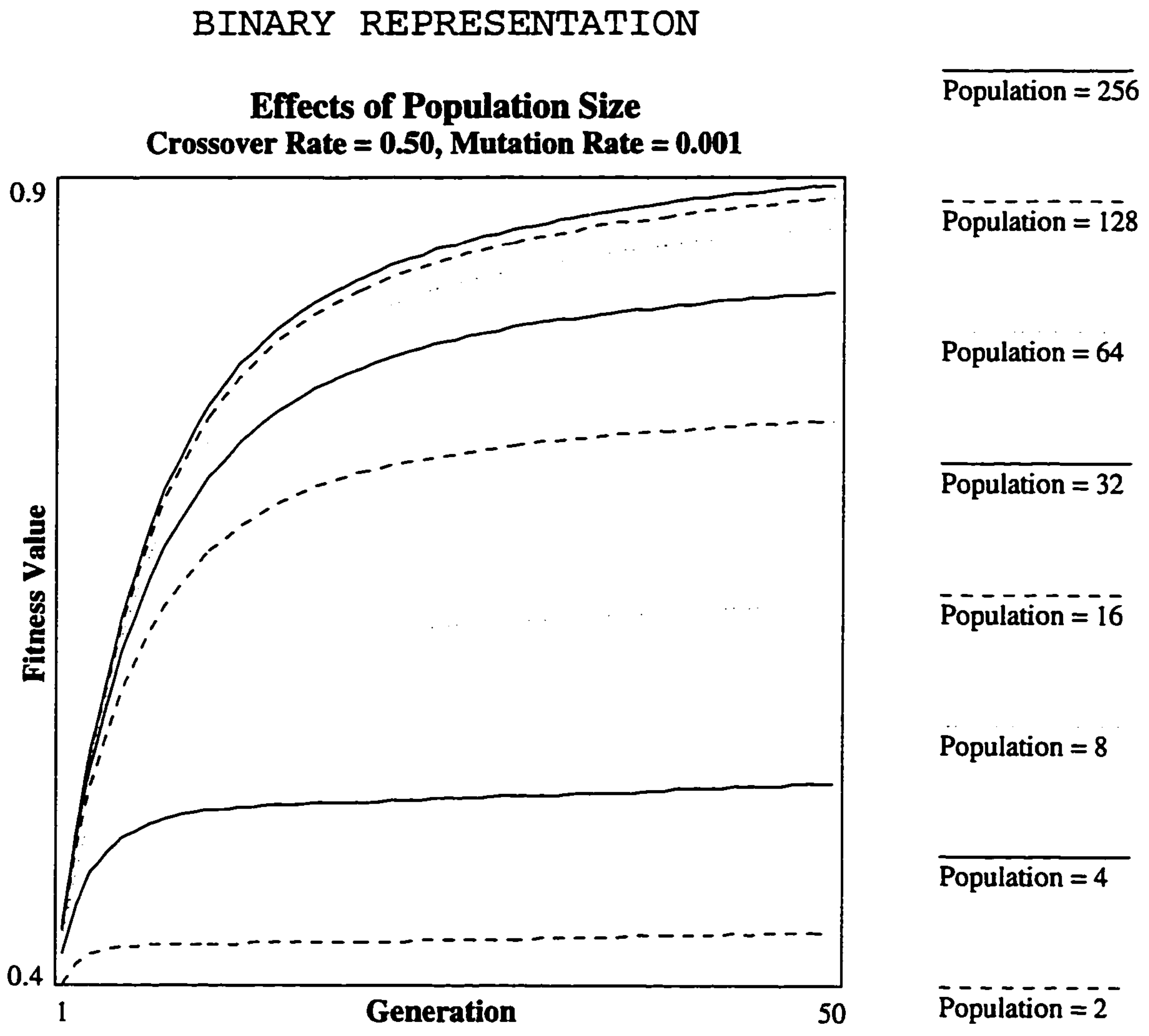


Figure 4.8: Population Size Effect on Binary Population

Since it has been determined that a syntactic GA is capable of effectively representing a 30-bit chromosome encoding three independent binary variables in sign-magnitude notation, the next logical step is a comparative study of on-line binary and syntactic GA performance for the optimization of $f_{x,y,z} = x^2 + y^2 + z^2$. Figure 4.8 shows the effect of population size on the average fitness of the binary genetic algorithm over the course of 1000 trial runs. As expected, larger populations have greater average fitness, as more points within the search

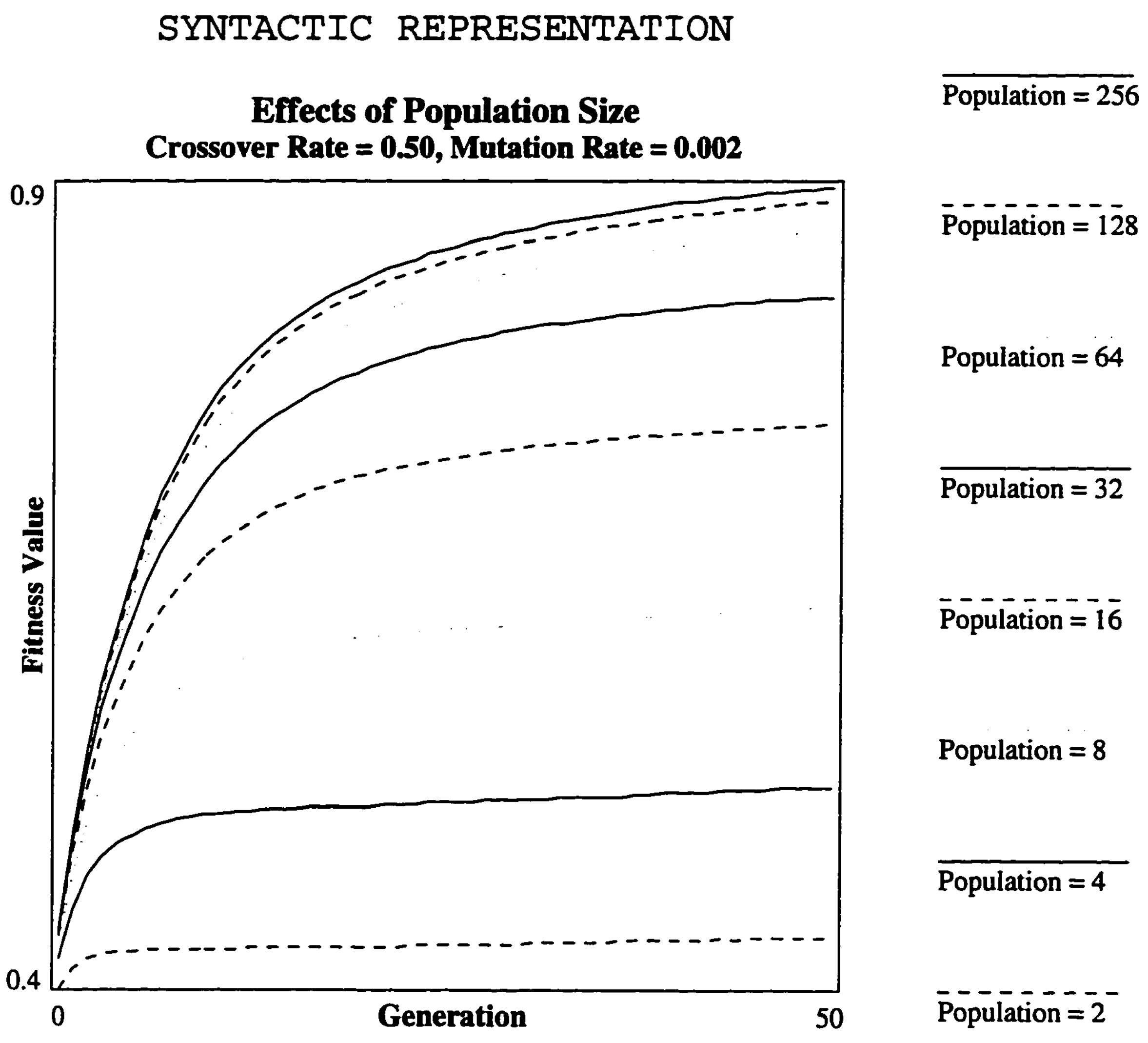


Figure 4.9: Population Size Effect on Syntactic Population

space can be examined during each generation.

Figure 4.9 shows the equivalent effect of population size on the average fitness of a syntactic GA operating on "bitstring rules". The performance of both GAs for this experiment is nearly identical. Note, however, that the mutation rate for the syntactic population is double that of the binary GA. As stated in Section 3.2, the rule mutation operator does not guarantee a new rule will always be produced, since a gene containing allele 1 may

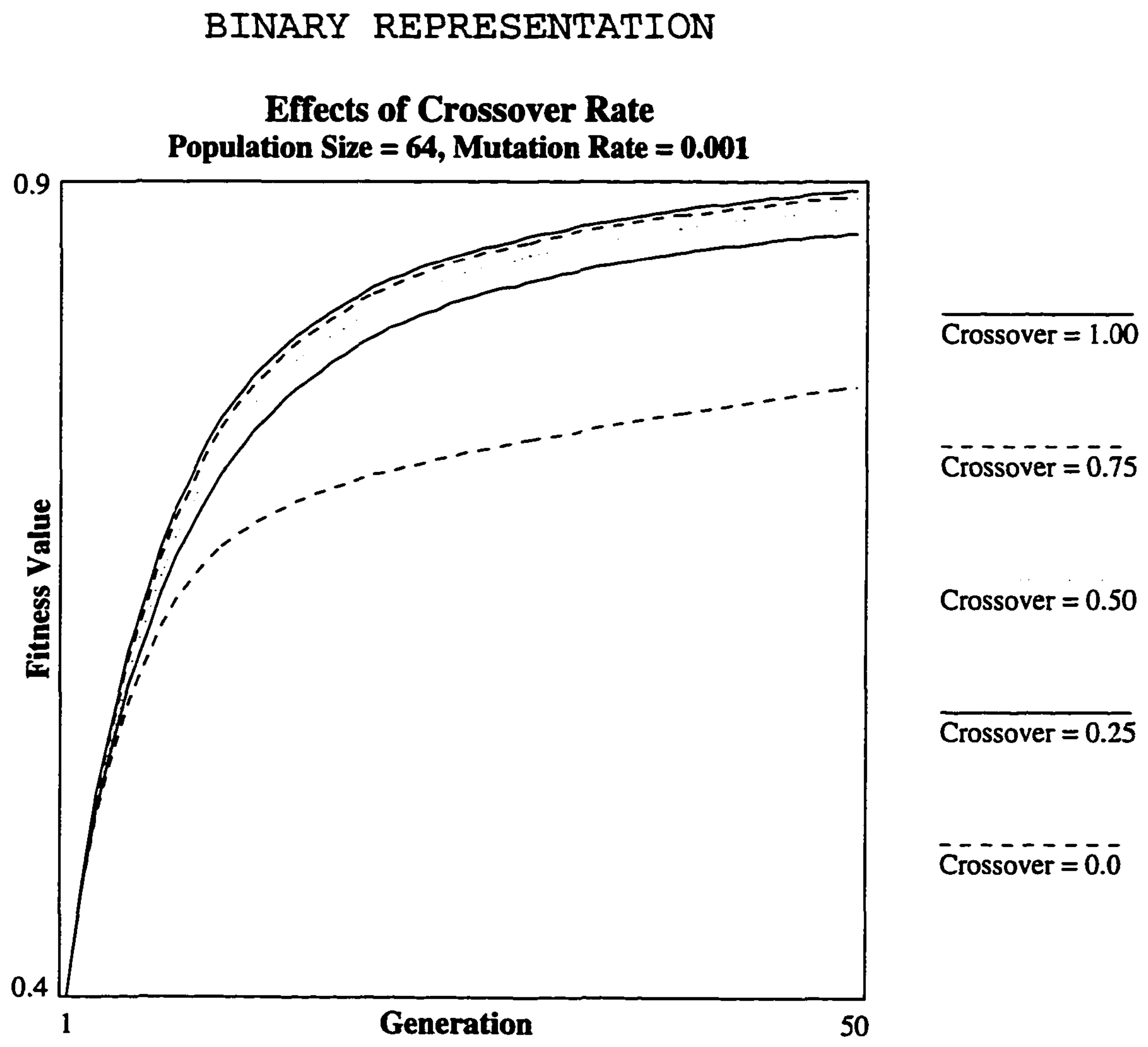


Figure 4.10: Crossover Rate Effect on Binary Population

be mutated to either 1 or 0. By doubling the rate at which the rule mutation operator is applied, the effect of non-productive mutation is eliminated.

Figure 4.10 shows the outcome of applying various crossover rates to the binary GA over the course of 1000 trial runs. The performance curves indicate that greater crossover rates lead to greater average population fitness. In general, however, the optimal crossover rate is highly problem-dependent, and results can vary significantly between individual trial

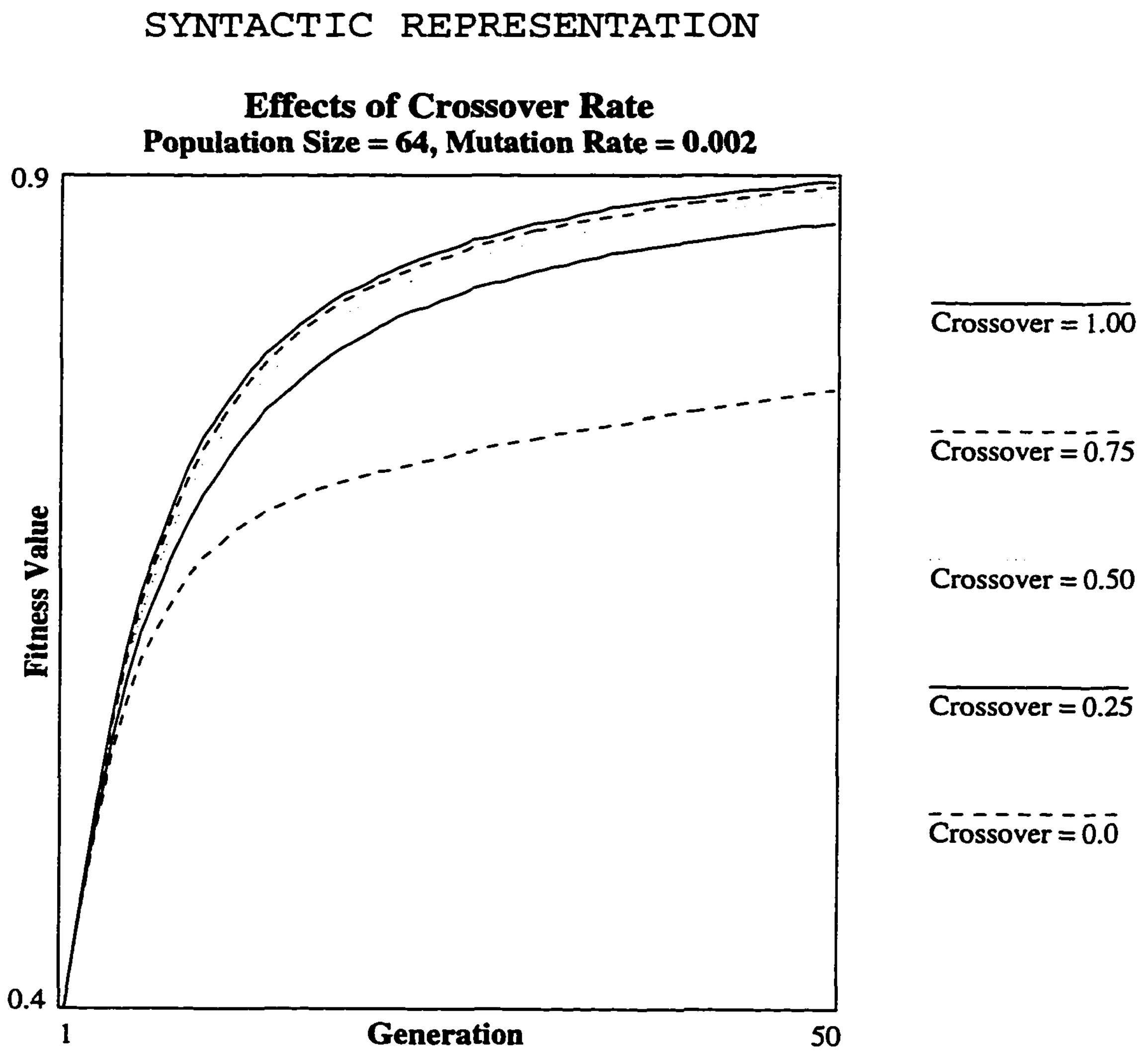


Figure 4.11: Crossover Rate Effect on Rule Population

runs.

Figure 4.11 depicts the effect of applying the same crossover rates to the syntactic GA population. Once again, the performance curves indicate that higher crossover rates generate higher average fitness values. As expected, crossover variance in this experiment produces nearly identical results in both binary and syntactic GAs.

Finally, the effect of varying the mutation rate of the binary GA is presented in Figure

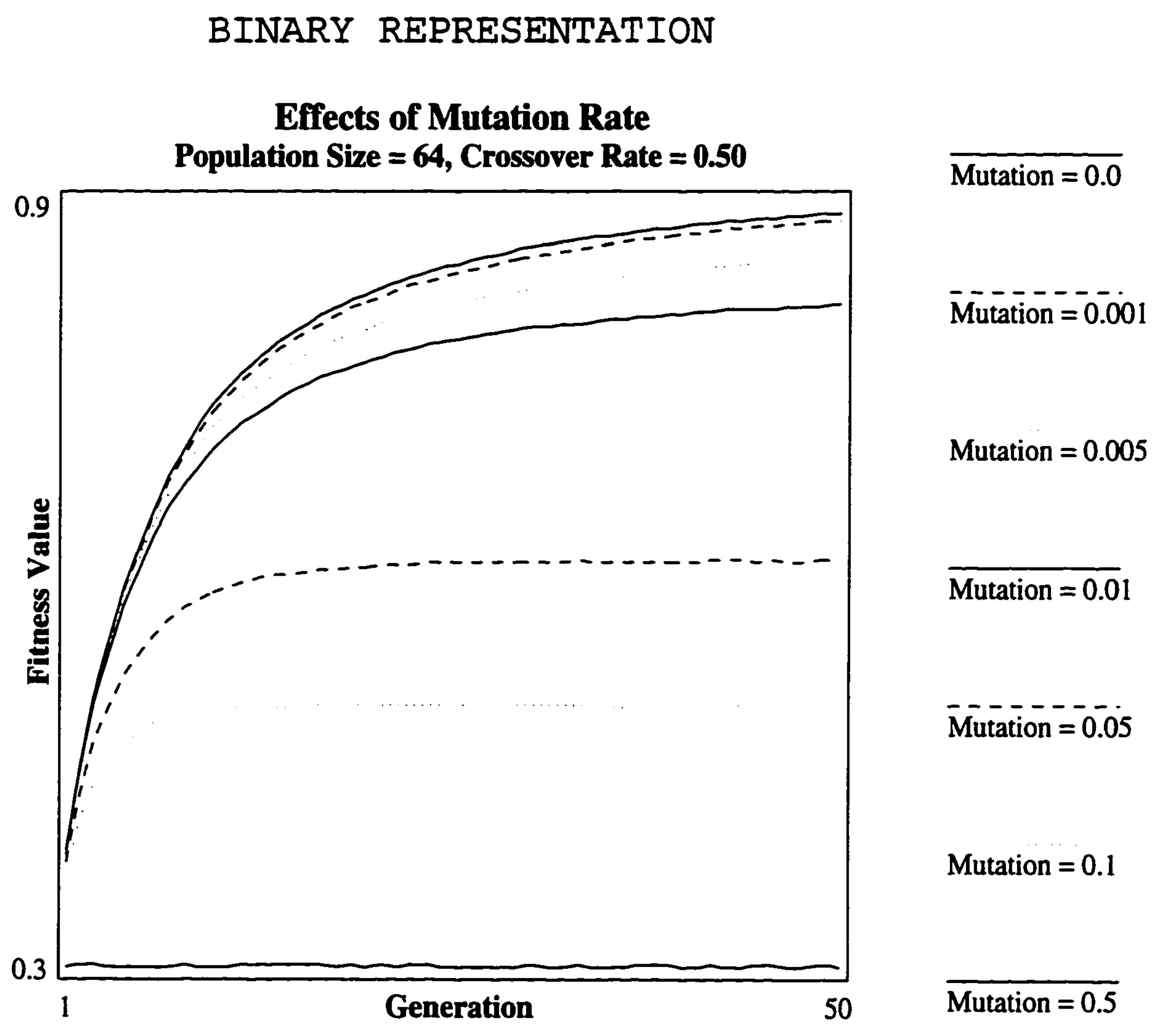


Figure 4.12: Mutation Rate Effect on Binary Population

4.12. The graph illustrates that lower mutation rates lead to greater average population fitness. Mutation is, however, only a secondary operator and is designed to act as a safeguard against premature allele loss. Because of the relatively large number of genes examined in the 1000 trial runs, the visible effect of the mutation operator is diminished. It is interesting to note, however, that a mutation rate of 0.5 leads to a random, non-productive walk through the search space.

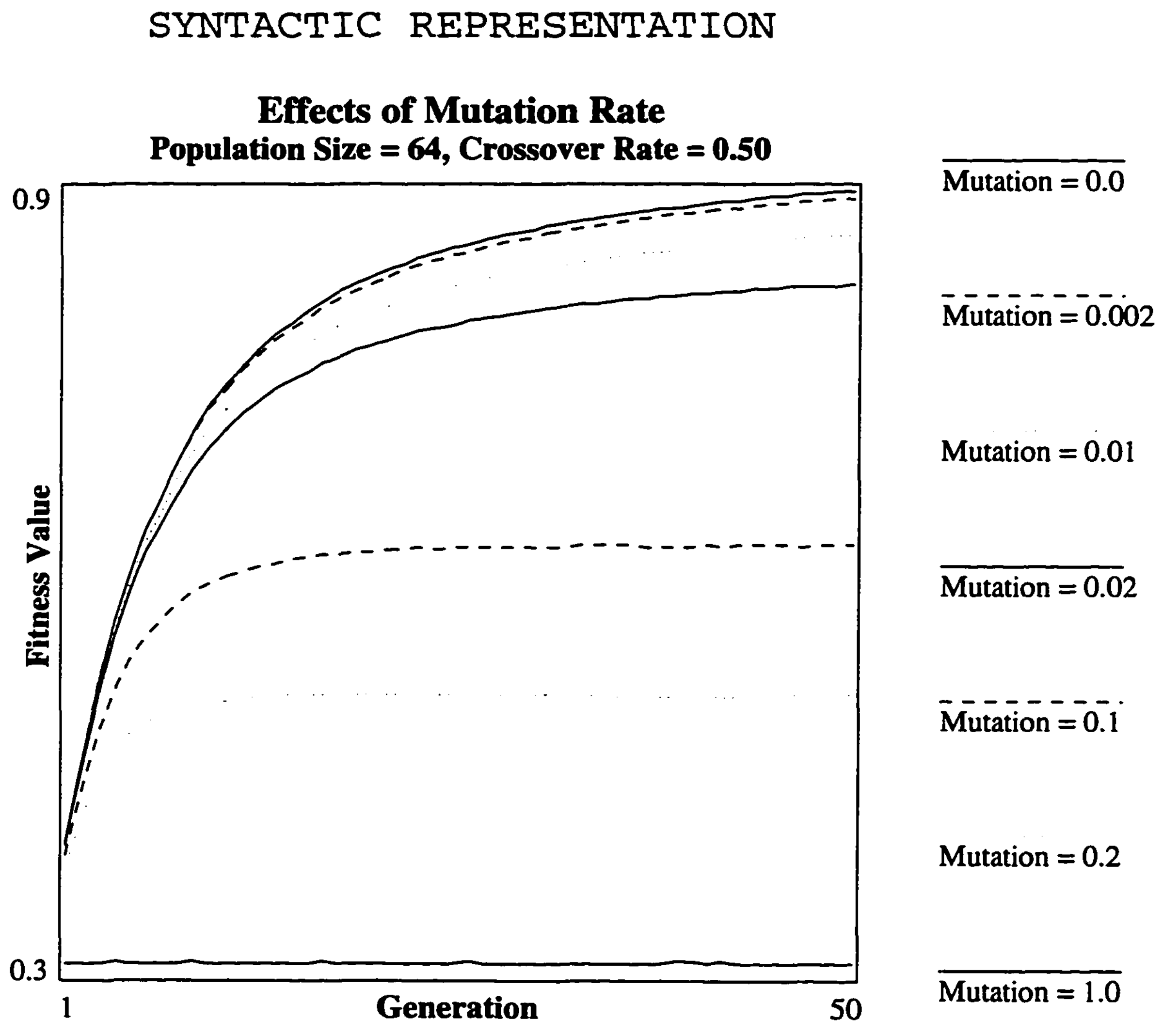


Figure 4.13: Mutation Rate Effect on Rule Population

Figure 4.13 shows the result of applying the same mutation rates to a population of syntactic binary rules. As explained earlier, all mutation rates have been doubled in order for the rule mutation operator to function at the same probability as its binary equivalent. Once again, the outcome of this experiment for both the binary and syntactic GAs is essentially identical.

The empirical evidence of the previous three experiments clearly shows that the syntactic GA is able to emulate a binary GA operating on 30-bit chromosomes, and leads to the general conclusion

Observation 4.2 *A binary GA with a population of* N *chromosomes of length* L, *a crossover rate* P_C *and a mutation rate* P_M *is equivalent to a syntactic GA with crossover rate* P_C *and mutation rate* $2{*}P_M$, *operating on a population of* N *rules derived from a grammar* G *containing* L+3 *productions, crossover specification list* G_C *and mutation specification list* G_M.

4.3 Trace Schemata

As described in Section 2.2, a schema is a simple pattern matching device which serves as the theoretical basis of genetic algorithm research. Under the binary representation, a schema is a string over the ternary alphabet $\{0, 1, *\}$, where the "*" meta-symbol serves as a special "don't care" marker for matching both 0 and 1. An analysis of the method by which schemata are processed by the genetic algorithm produced the result

$$m(\mathcal{H}, t+1) \geq m(\mathcal{H}, t)\frac{f(\mathcal{H}, t)}{\bar{f}(t)}\left[1 - p_c\frac{\delta(\mathcal{H})}{l-1} - p_m o(\mathcal{H})\right]$$

which is known as the Schema Theorem or Fundamental Theorem of Genetic Algorithms. It states that "building block" schemata of above-average fitness, low order and short defining length are sampled with increasing frequency by the genetic algorithm.

Extending the binary schema to a grammar-based approach, a *trace schema* can be defined as follows:

Definition 4.1 *A* **trace schema** $\mathcal{T}$ *is a derivation trace* $\mathcal{D}$ *in which* $n \geq 0$ *derivation subtraces of* $\mathcal{D}$ *have been replaced by the meta-symbol "[ ]".*

Each trace schema can therefore be represented as a list consisting only of production labels, the zero expansion delimiter, and the special meta-symbol "[ ]". As with a derivation trace, the number of production labels and zeroes must be equal.

In keeping with its binary counterpart, the trace schema functions as a similarity template for matching the derivation traces of grammar-defined rules. The "[ ]" meta-symbol serves as a special "don't care" marker indicating that the trace schema *produces* a derivation subtrace.

Definition 4.2 *For an arbitrary grammar* $\mathcal{G}$*, let derivation trace* $\mathcal{D} = d_1 d_2 .. d_n$ *be a string over* $\Sigma = \mathcal{P} \cup \{0\}$*, where* $\mathcal{P} = \{p \mid p \text{ is a production label in } \mathcal{G}\}$*; similarly, let trace schema* $\mathcal{T} = t_1 t_2 .. t_m$ *be a string over* $\Sigma \cup \{[\,]\}$*, where* $m \leq n$*. Then* $\mathcal{T} \Longrightarrow \mathcal{D}$ *(*$\mathcal{T}$ **produces** $\mathcal{D}$*) iff* $\mathcal{D}$ *is obtained from* $\mathcal{T}$ *by replacing each distinct instance of "[ ]" in* $\mathcal{T}$ *with any possible subtrace of* $\mathcal{D}$*.*

A *trace schema match* is then said to occur if the following holds:

Definition 4.3 *Trace schema* $\mathcal{T} = t_1 t_2 .. t_m$ *is a* **trace schema match** *of derivation trace* $\mathcal{D} = d_1 d_2 .. d_n$ *iff* $\mathcal{T} \Longrightarrow \mathcal{D}$*.*

As a simple example of trace schema matching, Figure 4.14 depicts one possible trace schema $\mathcal{T}$ which produces the derivation traces of a subset of rules for the 6-bit binary string grammar presented in Figure 4.1.

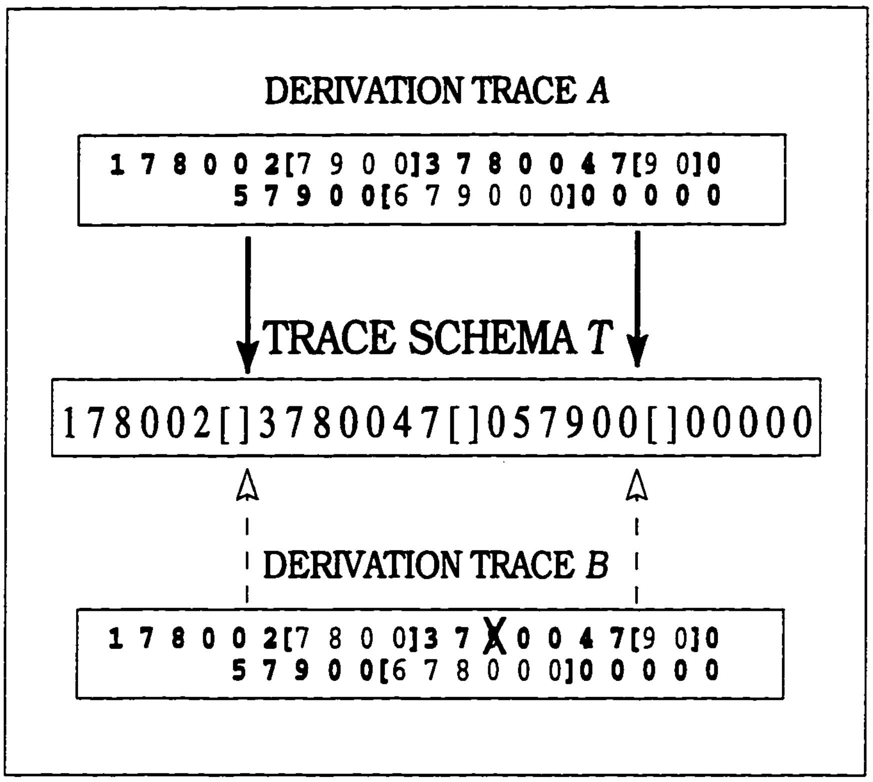


Figure 4.14: Trace Schemata Matching

Derivation trace *A* matches trace schema *T* as

- the first elements 1 7 8 0 0 2 in the trace schema consecutively match derivation trace elements `1 7 8 0 0 2`
- the first "[]" in the trace schema produces the derivation subtrace `7 9 0 0`
- the elements 3 7 8 0 0 4 7 in the trace schema consecutively match derivation trace elements `3 7 8 0 0 4 7`
- the second "[]" in the trace schema produces the derivation subtrace `9 0`
- the elements 0 5 7 9 0 0 in the trace schema consecutively match derivation trace elements `0 5 7 9 0 0`
- the third "[]" in the trace schema produces the derivation subtrace `6 7 8 0 0 0`
- the final elements 0 0 0 0 0 in the trace schema consecutively match derivation trace elements `0 0 0 0 0`

Trace schema T does not match derivation trace B, however, as the production label 9 at locus 13 in the derivation trace does not match the corresponding label 8 in the trace schema.

With an understanding of trace schemata, it is now possible to begin deriving the *Trace Schema Theorem*, a grammar-based equivalent to the Fundamental Theorem of Genetic Algorithms. As will be seen, the development features several similarities to the latter theorem, as well as a number of significant innovations. To begin, the *order of a trace schema* is defined. This will later prove useful for analyzing the effects of the mutation and crossover recombination operators on trace schemata.

The **order of a trace schema** $O(\mathcal{T})$ is the number of production labels contained in trace schema $\mathcal{T}$ which are not members of the grammar-specific terminal label set TLS_G. Figure 4.15 depicts the derivation trees of three representative trace schemata for the 6-bit binary string grammar. Trace schema A contains eight production labels. Of these, two are members of the grammar specific terminal label set TLS_G as indicated by the surrounding "□". The order of trace schema A is therefore $O(A) = 8 - 2 = 6$, the number of internal nodes in a derivation tree of the trace schema. Very specific trace schemata such as trace schema B match only one particular derivation trace, since its derivation tree contains no "[]" meta-symbols. The order of trace schema B is therefore $O(B) = 18 - 6 = 12$. General trace schemata such as trace schema C, on the other hand, contain no production labels. The order of trace schema C is thus $O(C) = 0$.

The **average fitness** $\overline{f(t)}$ of a rule population $\mathcal{R}$ at generation t is defined as

$$\overline{f(t)} = \frac{\sum_{i=1}^{N} f(r_i, t)}{N} \tag{4.1}$$

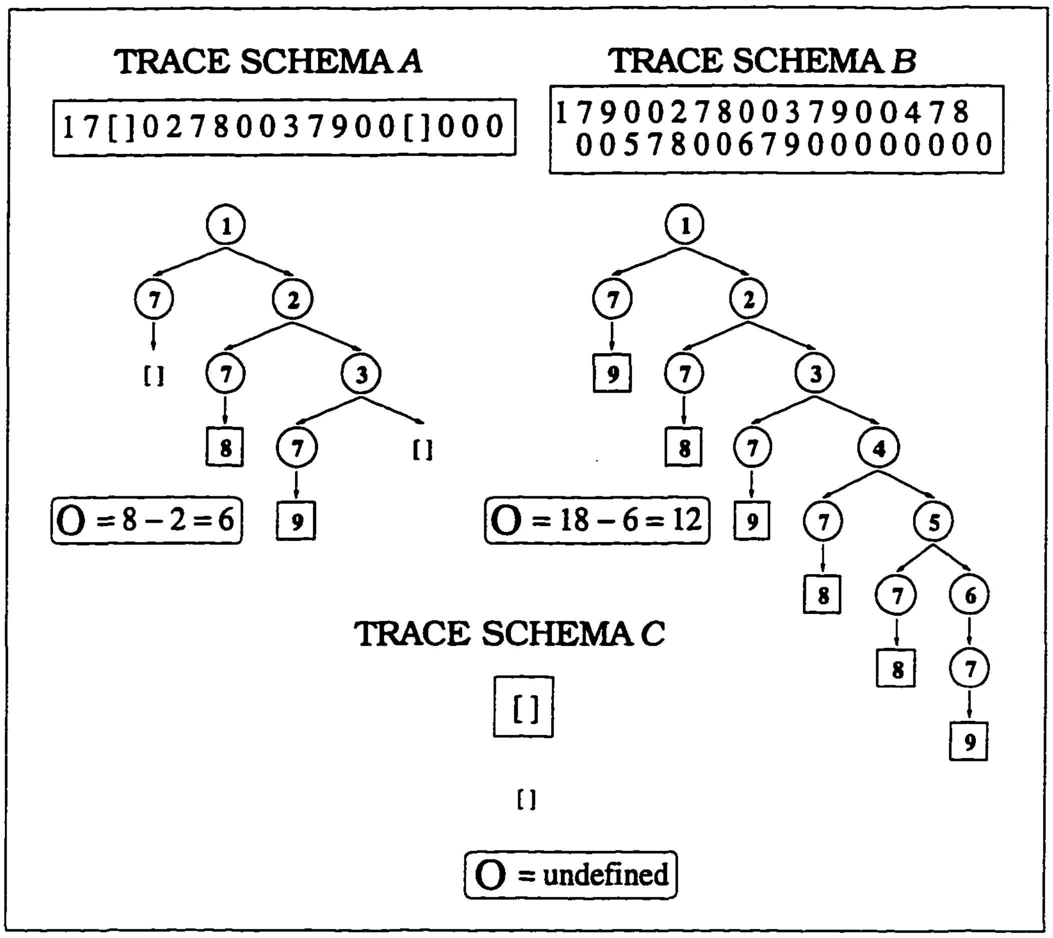


Figure 4.15: O Calculations for Sample Trace Schemata

where f is a fitness function, r_i is a member of $\mathcal{R}$, and N is the population size. The **average fitness of a trace schema** $\mathcal{T}$ is then the average fitness of all rules whose derivation traces match $\mathcal{T}$. If $M(\mathcal{T},t)$ represents the number of trace schema matches of $\mathcal{T}$ in $\mathcal{R}$ at generation t, the average fitness of a trace schema can be expressed as

$$f(\mathcal{T},t) = \frac{\sum_{r_i \in \mathcal{T}} f(r_i,t)}{M(\mathcal{T},t)}$$

The **fitness ratio of a trace schema** is the ratio of the average fitness of a trace schema

$\mathcal{T}$ to the average fitness of $\mathcal{R}$ during generation t, and can be expressed as

$$F(\mathcal{T},t) = \frac{f(\mathcal{T},t)}{\overline{f(t)}} \tag{4.2}$$

As stated in Section 2.1, reproduction with roulette wheel selection copies population members into subsequent generations with probability

$$p(r_j) = \frac{f(r_j)}{\sum_{i=1}^{N} f(r_i)}$$

Each trace schema $\mathcal{T}$ will therefore be copied from generation t with probability

$$p(\mathcal{T},t) = \frac{f(\mathcal{T},t)}{\sum_{i=1}^{N} f(r_i)}$$

After selecting N population members for the new generation $t+1$, the expected number of instances of $\mathcal{T}$ is

$$M(\mathcal{T},t+1) = M(\mathcal{T},t)N\frac{f(\mathcal{T},t)}{\sum_{i=1}^{N} f(r_i,t)}$$

Substituting $\overline{f(t)}$ from Equation 4.1 into the above equation gives

$$M(\mathcal{T},t+1) = M(\mathcal{T},t)\frac{f(\mathcal{T},t)}{\overline{f(t)}} \tag{4.3}$$

which is the trace schema fitness ratio of $\mathcal{T}$.

Equation 4.3 is the trace schema counterpart of the binary schema fitness ratio. It indicates that the growth of trace schemata depends only upon whether rules whose derivation traces match $\mathcal{T}$ have a greater fitness value on average than the population as a whole. Trace schemata with a fitness ratio above the population average fitness will receive increasing trials in future generations, whereas lower-than average trace schemata will be sampled less frequently. At each generation, this process occurs simultaneously for *every* trace schema over $\mathcal{R}$[1]. Thus, many trace schemata are processed in parallel.

The effect of rule crossover on $\mathcal{T}$ depends upon the individual trace schema itself. The crossover operator for grammar-derived rules operates by exchanging two arbitrarily selected derivation subtraces. Unless these subtraces are identical, rule crossover will disrupt some trace schemata over $\mathcal{R}$. As an example, trace schema A in Figure 4.15 will survive crossover if any production label in a derivation subtrace produced by "[]" in the trace schema is selected as the trace locus for crossover. Such an exchange would have no effect on any specified member of trace schema A. The selection of production label 1 would also keep trace schema A intact, as the entire trace schema would be exchanged during crossover. As stated in Chapter 3.2, members of the terminal label set TLS_G are excluded as crossover production labels; therefore, only crossover at the internal nodes of the derivation subtree located below production label 1 would disrupt trace schema A.

Closer observation reveals that the number of disruptive trace loci for crossover is equivalent to $O(\mathcal{T}) - 1$. Every derivation trace contains an equal number of production labels and zero delimiters. Thus, for a derivation trace of length l there are $l/2$ possible crossover sites. The probability p_d of picking a trace locus for crossover which would disrupt $\mathcal{T}$ is therefore

$$p_d(\mathcal{T}) = \frac{O(\mathcal{T}) - 1}{l/2} \tag{4.4}$$

[1]A trace schema over $\mathcal{R}$ is a trace schema matching the derivation trace of a member of $\mathcal{R}$

As shown in earlier figures, each rule derived from the 6-bit binary string grammar has a corresponding derivation trace of length 36. Thus, trace schema A will be disrupted with probability $p_d(A) = (6 - 1)/(36 / 2) = 0.28$.

By extending Equation 4.4, the probability p_s of a trace schema surviving crossover is $1 - p_d$ or

$$p_s(\mathcal{T}) = 1 - \frac{O(\mathcal{T}) - 1}{l/2}$$

Assuming crossover is performed with probability p_c, trace schema survival is therefore bounded by the expression

$$p_s \geq 1 - 2p_c\frac{O(\mathcal{T}) - 1}{l} \tag{4.5}$$

Multiplying Equation 4.3 by Equation 4.5 results in the expression

$$M(\mathcal{T}, t + 1) \geq M(\mathcal{T}, t)\frac{f(\mathcal{T}, t)}{\overline{f(t)}}\left[1 - 2p_c\frac{O(\mathcal{T}) - 1}{l}\right] \tag{4.6}$$

which describes the combined effects of rule reproduction and rule crossover on $\mathcal{T}$. In words, Equation 4.6 states that trace schemata of short order and above-average fitness are increasingly sampled in future generations.

As with rule crossover, the effect of rule mutation on $\mathcal{T}$ depends upon the individual trace schema. The mutation operator for grammar-based rules operates by arbitrary changing a derivation subtrace. Unless the newly created derivation subtrace is equivalent to the original, rule mutation will disrupt some trace schemata over $\mathcal{R}$. For example, trace schema A will survive rule mutation if any production label in a derivation subtrace produced by "[ ]" in the trace schema is selected as the trace locus for mutation. Any alteration occurring in these derivation subtraces would have no effect on trace schema A; however, if any other production label in the derivation trace except TLS_G members were selected as the trace locus for mutation, trace schema A might be destroyed.

The number of disruptive mutation loci in $\mathcal{T}$ is therefore $O(\mathcal{T})$. Assuming p_m is the probability of mutation occurring at each production label in a derivation trace, the chance of a production label remaining unaffected by the mutation operator is $1 - p_m$. As stated earlier, mutation between grammar-derived rules does not necessarily mean that an alteration will be made to schema $\mathcal{T}$; therefore, the mutation survival probability p_s of $\mathcal{T}$ is bounded by the expression

$$p_s(\mathcal{T}) \geq (1 - p_m)^{O(\mathcal{T})} \tag{4.7}$$

For $p_m \ll 1$, Equation 4.7 can be closely approximated by the expression

$$p_s(\mathcal{T}) \geq 1 - O(\mathcal{T})p_m \tag{4.8}$$

The joint effect of all three operators is obtained by combining Equation 4.6 with 4.8, with the result

$$M(\mathcal{T}, t+1) \geq M(\mathcal{T}, t)\frac{f(\mathcal{T}, t)}{\overline{f(t)}}\left[1 - 2p_c\frac{O(\mathcal{T}) - 1}{l}\right][1 - O(\mathcal{T})p_m]$$

Ignoring the small cross-product term

$$2p_cp_mO(\mathcal{T})\frac{(O(\mathcal{T}) - 1)}{l}$$

the above equation can be rewritten as

$$M(\mathcal{T},t+1) \geq M(\mathcal{T},t)\frac{f(\mathcal{T},t)}{\overline{f(t)}}\left[1 - 2p_c\frac{O(\mathcal{T})-1}{l} - p_m O(\mathcal{T})\right] \tag{4.9}$$

Equation 4.9 is thus the trace schema equivalent of the Fundamental Theorem of Genetic Algorithms. It states that above-average trace schemata containing relatively few non-terminal productions are sampled with increasing frequency by syntactic genetic search. As such, it serves as the **Trace Schema Theorem** for rule processing in search and optimization.

In Section 2.2 it was shown that for binary chromosomes of length l, there exists 3^l different schemata. The number of trace schemata obtainable from $\mathcal{G}$, on the other hand, is entirely dependent on the internal structure of the grammar itself. In order to make a comparison between binary and syntactic GA performance, all possible trace schemata can be calculated for the 6-bit binary string grammar presented earlier.

To begin, it is useful to observe that every rule obtained from the binary grammar has a partial rule derivation

1 <gene> 2 <gene> 3 <gene> 4 <gene> 5 <gene> 6 <gene> 0 0 0 0 0 0

where <gene> is expanded by production 7 in the partial grammar

7:	<gene>	$\rightarrow$	<allele>
8:	<allele>	$\rightarrow$	0
9:	<allele>	$\rightarrow$	1

Since the "[]" marker can replace any derivation subtrace, every expansion of production 7 can be a member of a trace schema in any of the ways presented in

U = {[] 7 [] 0 7 8 0 0 7 9 0 0}

In order to determine the number of trace schemata containing L = {1 2 3 4 5 6}, the set U can be substituted for <gene> in the partial derivation as follows:

1 U 2 U 3 U 4 U 5 U 6 U 0 0 0 0 0 0

Since $|\text{U}| = 4$, there must therefore be $4 * 4 * 4 * 4 * 4 * 4 = 4^6$ different trace schemata containing production labels in L.

Next, production label 6 can be removed from L by replacing the subtrace 6 U 0 with "[]" in the partial derivation

1 U 2 U 3 U 4 U 5 U [] 0 0 0 0 0

resulting in $4 * 4 * 4 * 4 * 4 = 4^5$ different trace schemata containing L = {1 2 3 4 5}. This process can be continued by iteratively eliminating productions labels 5 through 1 from $\mathcal{L}$ as seen in

1 U 2 U 3 U 4 U [] 0 0 0 0	$4*4*4*4 = 4^4$
1 U 2 U 3 U [] 0 0 0	$4*4*4 = 4^3$
1 U 2 U [] 0 0	$4*4 = 4^2$
1 U [] 0	$4 = 4^1$
[]	$1 = 4^0$

Using the geometric series substitution

$$\sum_{i=0}^{n} r^i = \frac{r^{n+1} - 1}{r - 1}$$

it can therefore be stated that

$$\frac{4^{l+1} - 1}{3}$$

different trace schemata exist for an arbitrary l-bit binary string grammar.

The number of trace schemata actually represented in population $\mathcal{R}$ can be determined by once again examining the partial derivation of each binary rule. In this case, every expansion of production 7 matches an arbitrary trace schema in any of the ways presented in

$$\text{V} = \begin{cases} \{[] \quad 7\,[]\,0 \quad 7\,8\,0\,0\} & \textit{if } \text{D}' = 7\,8\,0\,0 \\ \{[] \quad 7\,[]\,0 \quad 7\,9\,0\,0\} & \textit{if } \text{D}'' = 7\,9\,0\,0 \end{cases}$$

Since $|\,\mathtt{V}\,| = 3$ for both derivation subtraces $\mathtt{D}' = \mathtt{7\ 8\ 0\ 0}$ and $\mathtt{D}'' = \mathtt{7\ 9\ 0\ 0}$, the set V can be substituted for each occurrence of <gene> in the partial rule derivation. Then, following the previous counting methodology, the total number of trace schemata matched by a given rule in $\mathcal{R}$ can be determined as

1 V 2 V 3 V 4 V 5 V 6 V 0 0 0 0 0 0	$3*3*3*3*3*3 = 3^6$
1 V 2 V 3 V 4 V 5 V [] 0 0 0 0 0	$3*3*3*3*3 = 3^5$
1 V 2 V 3 V 4 V [] 0 0 0 0	$3*3*3*3 = 3^4$
1 V 2 V 3 V [] 0 0 0	$3*3*3 = 3^3$
1 V 2 V [] 0 0	$3*3 = 3^2$
1 V [] 0	$3 = 3^1$
[]	$1 = 3^0$

Thus, each l-bit binary rule in $\mathcal{R}$ matches

$$\frac{3^{l+1}-1}{2}$$

of the $\frac{4^{l+1}-1}{3}$ possible trace schemata. A population $\mathcal{R}$ of size N therefore contains n trace schemata as bound by the expression

$$\frac{3^{l+1}-1}{2} \leq n(\mathcal{R},t) \leq N\frac{3^{l+1}-1}{2}$$

This compares favorably with Equation 2.11

$$2^l \leq n(\mathcal{P},t) \leq N2^l$$

which presents the same calculation for a population $\mathcal{P}$ of binary chromosomes. Syntactic GAs therefore possess the same property of "implicit parallelism" inherent to the traditional genetic algorithm, and are thus able to create populations of rules in such a way as to simultaneously maximize a large number of trace schemata.

As presented in Figure 4.15, each trace schema $\mathcal{T}$ directly corresponds to a unique derivation tree. This property was previously utilized to describe overall features of $\mathcal{T}$ such as trace schema order $O(\mathcal{T})$. As with any tree structure, however, it is also possible to examine an arbitrary subtree of $\mathcal{T}$ defined as a *subtrace schema.*

Definition 4.4 *A* **subtrace schema** $\mathcal{S}$ *is a derivation subtrace* $\mathcal{D}'$ *of* $\mathcal{D}$ *in which* $n \geq 0$ *derivation subtraces of* $\mathcal{D}'$ *have been replaced by the meta-symbol* "[ ]".

Like the trace schema from which it is derived, each subtrace schema is represented as a list consisting of production labels, the zero expansion delimiter, and the "[ ]" meta-symbol. Since by Definition 3.6, each derivation trace $\mathcal{D}$ is also a derivation subtrace $\mathcal{D}'$ of $\mathcal{D}$, the number of subtrace schemata is at least as great as the number of trace schemata for any syntactic GA problem.

A *subtrace schema match* is said to occur if the following holds:

Definition 4.5 *Subtrace schema* $\mathcal{S} = s_1 s_2 .. s_m$ *is a* **subtrace schema match** *of derivation trace* $\mathcal{D} = d_1 d_2 .. d_n$ *iff* $\mathcal{S} \Longrightarrow \mathcal{D}'$, *where* $\mathcal{D}'$ *is any derivation subtrace of* $\mathcal{D}$.

Figure 4.16 depicts several representative subtrace schemata for the 6-bit binary grammar used in earlier examples. Unlike the trace schema, it is quite possible for a subtrace schema to match a derivation trace in more than one way. For example, subtrace schema *Y* could match a binary string rule at any of the 6 possible "bit" positions, while subtrace schema *Z* would match each of the 18 derivation subtraces in every rule.

The significance of the subtrace schema is not readily apparent in cases such as the 6-bit binary grammar, which is rigidly constrained by crossover and mutation specification lists. As will be seen in later experiments, however, subtrace schemata frequently describe the structure of partial rule expressions important in the overall solution to a syntactic GA problem. During rule crossover, these expression are exchanged intact whenever the corresponding subtrace schemata produce at least one of the crossover subtraces. Each

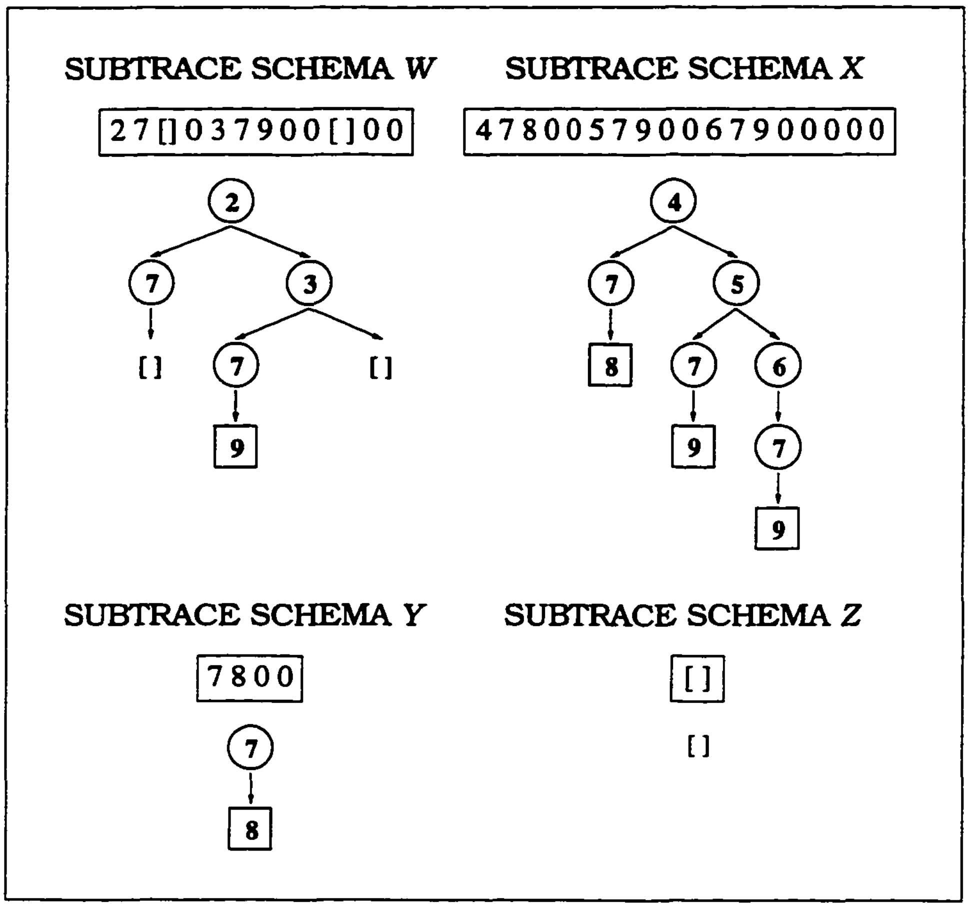


Figure 4.16: Sample Subtrace Schemata

partial solution is then copied into subsequent generations based upon the fitness of each rule whose derivation trace matches the subtrace schema in question. As such, subtrace schemata serve as the "building blocks" in the construction of the complex rule structures manipulated by syntactic GAs.

4.4 Equation Optimization

Genetic algorithms have traditionally been used as an optimization technique for isolating a near-maximal or near-minimal functional value for some set of equations. This method requires the GA to encode all independent variables of the system into a fixed length binary chromosome which can then be manipulated by genetic operators. As was shown in Section 4.1, the syntactic genetic algorithm can be applied successfully to this class of "parameter tuning" problems.

In the following experiment, the concept of optimizing an equation is taken to its next logical progression. Instead of "parameter tuning" the variables within an equation, the

```
 1:     <equation> → ( <operator> <expression1> <expression1> )
 2: <expression1> → <operand1>
 3:     <operand1> → <number>
 4:     <operand1> → ( <operator> <expression2> <expression2> )
 5: <expression2> → <operand2>
 6:     <operand2> → <number>
 7:     <operand2> → ( <operator> <expression3> <expression3> )
 8: <expression3> → <operand3>
 9:     <operand3> → <number>
10:     <operand3> → ( <operator> <expression4> <expression4> )
11: <expression4> → <operand4>
12:     <operand4> → <number>
13:     <operand4> → ( <operator> <number> <number> )
14:     <operator> → <sign>
15:         <sign> → +
16:         <sign> → -
17:         <sign> → *
18:         <sign> → /
19:       <number> → <digit>
20:        <digit> → 0
21:        <digit> → 1
22:        <digit> → 2
23:        <digit> → 3
24:        <digit> → 4
25:        <digit> → 5
26:        <digit> → 6
27:        <digit> → 7
28:        <digit> → 8
29:        <digit> → 9
```

Figure 4.17: Grammar for Optimization of an Equation

equation itself is treated as the parameter to be optimized. Figure 4.17 presents a grammar for producing arithmetic expressions over the operator set $\{+ \; - \; * \; /\}$ and the operand set $\{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\}$. Although this grammar appears relatively simple, it generates over $6.84 * 10^{50}$ distinct rules, and therefore provides the syntactic GA with a relatively large search space to examine.

The equation grammar produces a few obstacles in determining the fitness of individual population members. Each instance of the "/" operator, for example, may have as its divisor either 0 or an expression which evaluates to zero. In addition, many of the equations calculate either negative values or zero; roulette wheel selection, on the other hand, requires that all population members return a positive fitness value. For these reasons, the fitness function

$$f_x = \begin{cases} r_i + 1 & \text{if } r_i > 0 \\ 2^{r_i} & \text{if } r_i \leq 0 \\ \text{LPF} & \text{if } r_i \text{ undefined} \end{cases}$$

is used, where r_i is a rule in population $\mathcal{R}$ and LPF is a constant set to the least positive float. Figure 4.18 presents the best solution with optimal fitness value $f_x = 9^{32} + 1$.

In Section 4.1, it was shown that a binary chromosome of length l can be represented by

```
(*
   (*
      (*(*(*99)(*99))(*(*99)(*99)))
      (*(*(*99)(*99))(*(*99)(*99)))
   )
   (*
      (*(*(*99)(*99))(*(*99)(*99)))
      (*(*(*99)(*99))(*(*99)(*99)))
   )
)
```

Figure 4.18: Best Solution for Equation Optimizer

a syntactic GA as a "bitstring rule" derived from a grammar containing $l + 3$ productions. In order to once again compare binary and syntactic genetic algorithms, the "best solution" obtainable for the equation optimizer can be represented as a binary chromosome. A close observation of the equation grammar reveals that there exist exactly 16 terminal symbols which can be represented using the binary encoding scheme presented in Figure 4.19.

Binary Encoding	Terminal Symbol	Binary Encoding	Terminal Symbol
0000	0	1000	8
0001	1	1001	9
0010	2	1010	(
0011	3	1011	)
0100	4	1100	+
0101	5	1101	-
0110	6	1110	*
0111	7	1111	/

Figure 4.19: Binary Encoding of Equation Optimizer Terminal Symbols

Since the "best solution" contains 125 terminal symbols, it can be represented in a binary string of length $l = 125 * 4 = 500$ bits. This result is easily generalized to

Observation 4.3 *Any rule of length* L *derivable from a grammar* G *containing* T *distinct terminal symbols can be encoded into a binary chromosome of length* $\lceil \log_2 \mathrm{T} \rceil * \mathrm{L}$.

Since the goal of this experiment is the "parameter tuning" of the entire equation to produce this 500-bit optimal result, the entire population of the binary GA must be composed of 500-bit chromosomes. The fitness function f_x for syntactic equations is also sufficient for binary chromosomes, as ill-formed expressions evaluate to LPS.

Figure 4.20 presents the average population fitness of both the syntactic and binary GAs for a population of 100 individuals using a 0.60 crossover rate. As with earlier experiments,

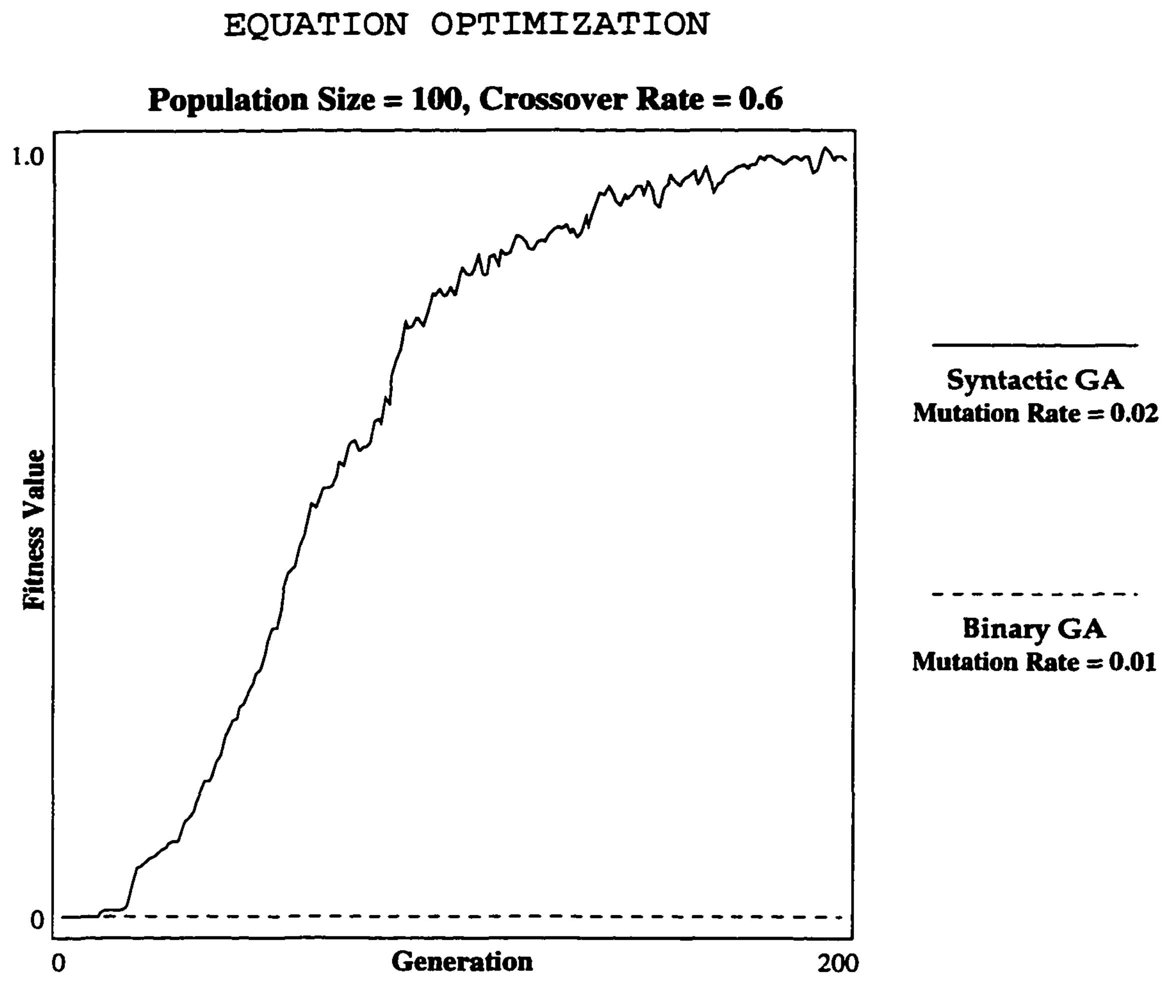


Figure 4.20: Average Fitness of Equation Optimization

the 0.02 mutation rate of the syntactic GA is double that of its binary counterpart. The resulting performance curves are the average of 10 trials.

As can be seen, the syntactic GA population converges upon near optimal solutions. In fact, by generation 75 the "best solution" shown in Figure 4.18 was present in the populations of each of the 10 trials. Since 75 generations of 100 equations represents only 750 of the $6.84 * 10^{50}$ possible equations, the syntactic GA was able to discover the optimal solution after examining only $750/6.84 * 10^{50} = 1.09 * 10^{-48}$ of the points in the search space.

This result is due largely to the bottom-up propagation of important subtrace schemata. For example, the subtrace schemata

19 29 0 0
14 15 0 0

corresponding to **9** and ***** respectively are quickly spread throughout the population. Eventually, the new subtrace schema

13 14 15 0 0 19 29 0 0 19 29 0 0 0

representing (*** 9 9**) is formed and in turn propagated into future generations. This "building block" process continues until the final optimal solution is constructed.

The binary genetic algorithm, on the other hand, has a near-zero average population fitness throughout the course of the experiment. This is not surprising when it is considered that a 500-bit binary chromosome has $2^{500} = 1.27 * 10^{150}$ possible interpretations of which only $4.61 * 10^{50}$ represent well-formed expressions. Thus, there is only the small probability $4.61*10^{50}/1.27*10^{150}*100 = 1.41*10^{-130}$ of discovering an equation during each generation. Even in the unlikely event that one of these valid points in the search space was isolated, it is extremely doubtful that the encoded equation would survive the combined effects of binary crossover and mutation. The inability of the binary GA to operate in this domain of this experiment leads to the result

Observation 4.4 *Due to the inherent difficulties of binary encoding, the effect of gene epistasis and the high order and defining length of typical solution schemata, a binary GA is not equivalent to a syntactic GA for problems whose parameters are structured by a non-trivial grammar* G.

Combining Observation 4.2 with Observation 4.4 leads to the final conclusion

Observation 4.5 *Empirically, the class of problems* B *solvable by a binary GA is a subset of* S, *the class of problems solvable by a syntactic GA.*

As explained in Section 3.2.3, the application of the syntactic mutation operator does not ensure that a change will be made to the original rule. Since the replacement derivation subtrace is arbitrarily constructed, it is possible that the newly selected productions will be identical to those of the pre-mutated derivation subtrace. For grammar G, however, there are only two possible mutation subtraces for $P_M = 7$; therefore, by doubling the rate p_m at which the rule mutation operator is applied, the affect of non-productive mutation can be easily eliminated. Thus, the syntactic mutation of rules in G using specification list $\mathrm{L_M}$ at rate $2p_m$ is equivalent to the binary mutation of 6-bit chromosomes performed at rate p_m.

Chapter 5

Machine Learning and Classification

5.1 Balance Scale

The data for this experiment was obtained from the University of California Irvine Machine Learning Repository[130]. Based upon psychological experiments originally reported by Siegler[228] in 1976, the data has since been used in a variety of forms by several different studies[147, 153, 176, 216]. As depicted in Figure 5.1, the balance scale experiment involves

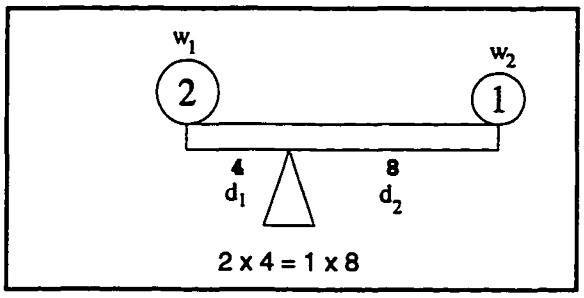


Figure 5.1: Calculation of Balance Condition

the classic Physics problem of determining the proper fulcrum for two objects of weights w_1 and w_2 placed at distances d_1 and d_2 from a center point at opposite ends of a weightless surface. As shown above, a balance condition can be satisfied by ensuring that the quantities $(w_1 * d_1)$ and $(w_2 * d_2)$ have equal values; if not, the "scales" will tilt to either the left or the right, depending on which respective quantity is greater.

The balance scale is a typical classification problem. It serves as an excellent experiment for the syntactic GA because it does not involve a simple boolean discrimination function. Instead, three distinct classes must be distinguished - balanced, right-leaning and left-leaning - based upon an unknown algebraic relationship between four variables. In other words, *three* different equations must be simultaneously optimized. Drawing from the results of Section 4.4, such a task would be exceedingly difficult for the binary GA to accomplish.

Figure 5.2 shows a representative sample of the training data used for this experiment.

```
( 'to-right ( 3 2 3 3 ) )
( 'balanced ( 2 2 1 4 ) )
( 'to-right ( 1 2 3 4 ) )
(  'to-left ( 4 1 1 2 ) )
                 :
(  'to-left ( 5 4 2 3 ) )
( 'to-right ( 2 2 4 3 ) )
( 'balanced ( 1 4 1 4 ) )
( 'balanced ( 3 2 2 3 ) )
```

Figure 5.2: Training Data for Balance Scale Experiment

Each example gives the value of the variables w_1, d_1, w_2 and d_2, as well as the correct classification, indicated by one of the LISP primitives `'to-right`, `'balanced`, or `'to-left`. Every combination of integer values for the variables over the range [1..5] is presented to the

GA; therefore, 625 different items are examined in total. The distribution of the `'to-left` and `'to-right` classes in the training data is equal at 46.08% apiece. The `'balanced` class, however, is represented by only 7.84% of the instances. This uneven distribution of examples poses an interesting challenge for the syntactic GA. Care must be taken to ensure that resources will be apportioned equally to all three target classes.

```
1:          <rule> → ( cond <cond1> <cond2> <cond3> )
2:         <cond1> → ( <equation> 'to-left )
3:         <cond2> → ( <equation> 'balanced )
4:         <cond3> → ( <equation> 'to-right )
5:      <equation> → ( <relational> <expression1> <expression1> )
6:   <expression1> → <operand1>
7:      <operand1> → <term>
8:      <operand1> → ( <operator> <expression2> <expression2> )
9:   <expression2> → <operand2>
10:     <operand2> → <term>
11:     <operand2> → ( <operator> <expression3> <expression3> )
12:  <expression3> → <operand3>
13:     <operand3> → <term>
14:     <operand3> → ( <operator> <expression4> <expression4> )
15:  <expression4> → <operand4>
16:     <operand4> → <term>
17:     <operand4> → ( <operator> <term> <term> )
18:   <relational> → <boolean>
19:      <boolean> → =
20:      <boolean> → <
21:      <boolean> → >
22:      <boolean> → <=
23:      <boolean> → >=
24:      <boolean> → <>
25:     <operator> → <sign>
26:         <sign> → +
27:         <sign> → -
28:         <sign> → *
29:         <sign> → /
30:         <term> → <variable>
31:     <variable> → left-weight
32:     <variable> → right-weight
33:     <variable> → left-distance
34:     <variable> → right-distance
```

Figure 5.3: Grammar for Balance Scale Experiment

The grammar for the balance scale experiment is displayed in Figure 5.3 on the preceding page. Since it is known that a solution to this problem involves discriminating between three separate classes, the grammar is designed to produce LISP **cond** statements containing three distinct elements. Productions 2, 3 and 4 generate conditional statements which return the predicates `'to-left`, `'balanced`, and `'to-right`, respectively. These predicates indicate how each training instance was classified. In the event that none of the elements in the **cond** are triggered, `nil` is returned instead.

The problem statement specifies that each class can be stated as a relationship between two mathematical quantities - the state of both sides of a fulcrum in terms of four variables. The condition of each rule therefore consists of a boolean relational operator separating two mathematical expressions. This design allows the three condition-action pairs to act as independent boolean classifiers, and helps to alleviate any bias against the relatively infrequent occurrences of `'balanced` examples. Each "scale rule" will thus divide its attention equally between all three alternatives.

One of six relational operators - "=", "<", ">", "<=", ">=" or "<>" - is assigned to each element of the **cond** statement by means of productions 18 through 23. The remainder of the grammar is allocated to the creation of mathematical expressions. Two such expressions, in conjunction with a relational operator, can be seen as constituting a single equation. For this reason, the `expression` structure initiated by production 7 in the grammar is borrowed directly from the "Equation Optimizer" of the previous chapter.

Productions 8 through 17 therefore create mathematical expressions of various length. The branching mechanism of the <$\texttt{expression}_i$> non-terminals serves as a biasing mechanism which favors the creation of shorter conditional statements during "random" rule generation and mutation. Each <$\texttt{expression}_i$> is expanded to either a variable or to an infix operation with equal probability. If an operation is chosen, there is again a 50% chance that each term will be expanded by means of <**expression**$_{i+1}$>. This effect is cumulative, so there is only a 25% chance of generating <**expression3**> and a 12.5% chance of creating <**expression4**>. Because of the added number of trace schemata caused by such branching, a term which contains unnecessary calculations such as

```
( - left-weight ( + left-weight left-weight ) )
```

is likely to be simplified to `left-weight` during rule mutation and crossover.

Each operation can contain up to five levels of nested subexpressions. This cutoff point was established by examining the opportunity cost of including additional productions in the grammar. Increasing the size of the grammar impairs the performance of the mutation, crossover and reproduction operators, as more production labels must be examined and copied. Conversely, the likelihood of generating each successive subexpression decreases exponentially with parse tree depth. Through experimentation, it was determined that a maximum depth of five resulted in a favorable tradeoff between processing speed and rule expressiveness.

In keeping with Section 4.4, only the four basic additive and multiplicative functions - "+", "−", "*", and "/" - could be used as mathematical operators. Productions 25 through 29 placed one of these at the beginning of each infix expression. Unlike the "Equation Optimizer", however, these functions did not operate over the integers [0..9]. Since the three balance equations needed to learn a relationship between two weights and two distances, the operand set instead consisted of the four variable names `left-weight`, `right-weight`, `left-distance`, and `right-distance`. Each variable was assigned a value from a corresponding entry in the training data, and were generated by the final five productions of the grammar. Figure 5.4 shows the genotype and phenotype of an arbitrarily derived rule from the experiment grammar.

GENOTYPE

```
(1
   2 5 18 23 0 0 6 7 30 31 0 0 0 0 6 7 30 32 0 0 0 0 0 0
   3 5 18 19 0 0 6 7 30 31 0 0 0 0 6 8 25 28 0 0 9 11 25 26 0 0 12 13 30 32 0 0 0 0 12 13 30 34 0 0 0
      0 0 0 9 11 25 28 0 0 12 14 25 26 0 0 15 16 30 31 0 0 0 0 15 16 30 32 0 0 0 0 0 0 12 14 25 28
      0 0 15 16 30 33 0 0 0 0 15 17 25 27 0 0 30 32 0 0 30 33 0 0 0 0 0 0 0 0 0 0 0 0
   4 5 18 22 0 0 6 8 25 26 0 0 9 10 30 31 0 0 0 0 9 11 25 29 0 0 12 13 30 32 0 0 0 0 12 13 30 31 0 0
      0 0 0 0 0 0 6 8 25 27 0 0 9 10 30 33 0 0 0 0 9 11 25 29 0 0 12 13 30 32 0 0 0 0 12 13 30 33
      0 0 0 0 0 0 0 0 0 0 0 0 )
```

PHENOTYPE

```
( cond
  (
    ( >= left-weight right-weight )
      'to-left
  )
  (
    ( =
      left-weight
      ( *
        ( + right-weight right-distance )
        ( *
          ( + left-weight right-weight )
          ( * left-distance ( - right-weight left-distance ) )
        )
      )
    )
      'balanced
  )
  (
    ( <=
      ( + left-weight ( / right-weight left-weight ) )
      ( - left-distance ( / right-weight left-distance ) )
    )
      'to-right
  )
)
```

Figure 5.4: Rule Generated from Balance Scale Grammar

Each rule $\mathcal{A}$ in the population is evaluated by the fitness function

$$f_{\mathcal{A}} = \begin{cases} C - I & \text{if } C - I > 0 \\ 2^{(C-I)} & \text{otherwise} \end{cases}$$

where C is the number of balance scales correctly classified and I is the number incorrectly identified. The $2^{(C-I)}$ calculation ensures that the poorer the rule performs, the less its fitness value will be. By this means, positive f_A values are always produced, as required by roulette wheel selection.

The syntactic GA performance for the scale balance problem was excellent. Using a population of 200 rules, a 0.7 crossover rate and a 0.02 mutation rate over 50 experimental runs, GERDS discovered an optimal solution within 200 generations in all but two cases. In these exceptions, near optimal solutions in which two of the three classes were correctly identified were present in the population. Figure 5.4 depicts one optimal solution to the

```
( cond
   (
      ( < ( * right-distance right-weight ) ( * left-weight left-distance ) )
         'to-left
   )
   (
      ( = ( * right-distance right-weight ) ( * left-weight left-distance ) )
         'balanced
   )
   (
      ( > ( * right-distance right-weight ) ( * left-weight left-distance ) )
         'to-right
   )
)
```

Figure 5.5: Solution to Balance Scale Experiment

problem, although many variations in the mathematics occurred over the 50 runs.

The basis of the solution was the initial discovery of subtrace schemata such as

```
6 8 25 28 0 0 9 10 31 0 0 0 9 10 33 0 0 0 0 0
6 8 25 28 0 0 9 10 32 0 0 0 9 10 34 0 0 0 0 0
```

corresponding to the expressions

```
( * left-weight left-distance )
( * right-weight right-distance )
```

which is necessary for all three conditional statements of an optimal "scale rule". These subtraces were then quickly propagated throughout the GA population, after which the correct relational operators "<", ">" and "=" were put into place.

Observation 4.5 concluded that the traditional genetic algorithm cannot emulate a syntactic GA for non-trivial grammars. This was empirically evidenced by the vastly different performance of the two paradigms with regard to the "Equation Optimizer". Since the balance scale experiment was structured as a 3-way equation optimization problem and borrows heavily from the grammar of the previous experiment, it is logical to conclude that a binary GA would be unable to effectively solve this problem.

5.2 LED Classification

The data for this experiment was obtained from the University of California Irvine Machine Learning Repository[130], and has been used in several machine learning studies[33, 245].

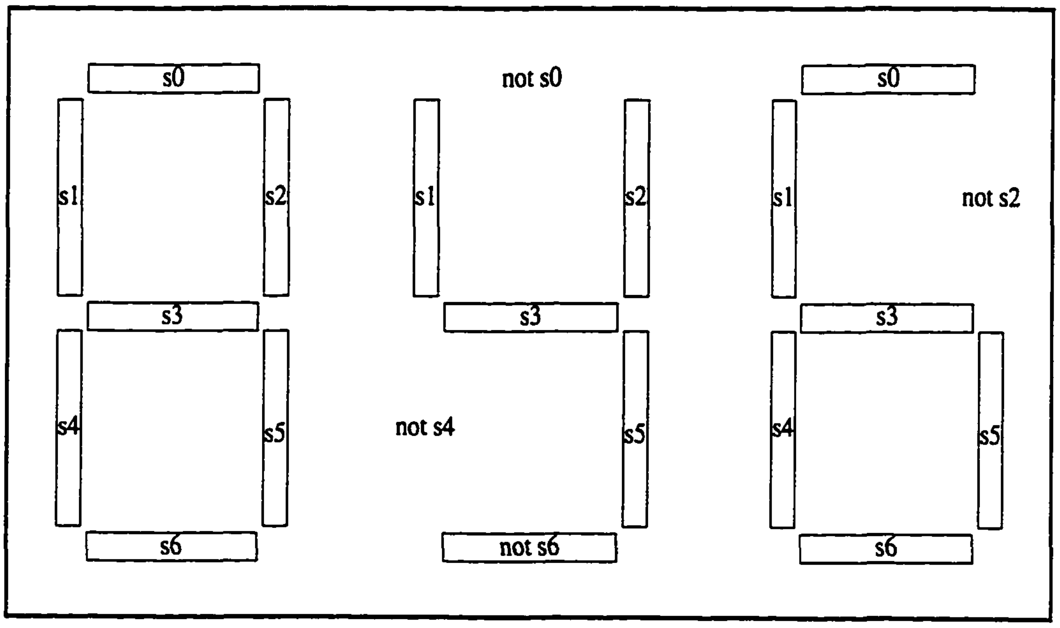


Figure 5.6: Example LED configurations

A standard LED (Light Emitting Diode) display contains seven diodes, which may be lit in certain combinations to produce an alphanumeric character. The purpose of this experiment is to train a learning system to recognize the ten decimal digits. The task is made more difficult by the introduction of noise into the environment. Each of the seven diodes has a 10% chance of misfiring, thereby affecting the output of the overall display. Figure 5.6 presents an example of three LED outputs. The middle display does not correspond to any decimal character, and so obviously contains at least one inversion - it is most probably a 4 with an inverted s6 diode. The left and right displays show the digits 8 and 6 respectively; however, these may also be in error, as the two outputs differ by only one diode.

This experiment is interesting from a GA perspective for several reasons. Since the problem entails the categorization of the digits 0 through 9, ten distinct classes must be learned. The distinction between these classes is blurred by the presence of noise in the data. Because the seven diodes for each display can be represented as boolean values, it is possible for the traditional binary GA to encode this problem. A solution, however, would require the decomposition of the problem into ten separate tasks.

Figure 5.7 presents a small sample of the 2000 training instances used for this experiment. The digits represent the correct classification of each LED display, while the t and nil values signify the states of the seven diodes (read from left to right as s0 through s6). Each training instance was created by using a simple program which randomly selected a correct solution to one of the classes and then altered the status of each diode with a 10% probability. Theoretically, the distribution of each class should be equal at 200; however, variation in the data occurred for each run of the experiment. Figure 5.8 shows information regarding the training instances used in the first run of the system. Note that some digits, especially 8 and 9, are likely to have a high percentage of incorrectly classified instances.

```
( 1 (  nil nil  t   t  nil  t  nil ) )
( 3 (   t  nil nil  t  nil  t   t  ) )
( 4 (  nil  t   t   t  nil  t  nil ) )
( 3 (   t  nil  t   t  nil  t   t  ) )
                :
( 9 (   t   t   t   t  nil nil  t  ) )
( 1 (  nil nil  t  nil nil  t  nil ) )
( 6 (   t   t  nil nil  t   t   t  ) )
( 5 (   t   t   t   t  nil  t   t  ) )
```

Figure 5.7: Training Data for LED Experiment

The grammar used for the LED classifier is presented in Figure 5.9 on the following page. Its design proceeds in a relatively straight-forward manner from the problem statement. Like the balance scale experiment of the previous section, this problem requires the simultaneous learning of multiple concepts with uneven distributions. In the case of the former, a grammar was created to produce LISP cond statements with exactly three

LED DIGIT	NUMBER OF INSTANCES	ALL BARS CORRECT	FALSE POSITIVE
0	172	101	23
1	195	112	14
2	196	92	1
3	214	119	17
4	188	90	4
5	212	119	24
6	214	135	29
7	216	108	15
8	194	132	39
9	199	129	35

Figure 5.8: Distribution of LED Training Data

```
 1:    <rule> → ( cond  <cond0> <cond1> <cond2> <cond3> <cond4> <cond5>
                        <cond6> <cond7> <cond8> <cond9> )
 2:   <cond0> → ( <bars> 0 )
 3:   <cond1> → ( <bars> 1 )
 4:   <cond2> → ( <bars> 2 )
 5:   <cond3> → ( <bars> 3 )
 6:   <cond4> → ( <bars> 4 )
 7:   <cond5> → ( <bars> 5 )
 8:   <cond6> → ( <bars> 6 )
 9:   <cond7> → ( <bars> 7 )
10:   <cond8> → ( <bars> 8 )
11:   <cond9> → ( <bars> 9 )
12:    <bars> → ( and <bar0> <bar1> <bar2> <bar3> <bar4> <bar5> <bar6> )
13:    <bar0> → <choice0>
14: <choice0> → s0
15: <choice0> → ( not s0 )
16:    <bar1> → <choice1>
17: <choice1> → s1
18: <choice1> → ( not s1 )
19:    <bar2> → <choice2>
20: <choice2> → s2
21: <choice2> → ( not s2 )
22:    <bar3> → <choice3>
23: <choice3> → s3
24: <choice3> → ( not s3 )
25:    <bar4> → <choice4>
26: <choice4> → s4
27: <choice4> → ( not s4 )
28:    <bar5> → <choice5>
29: <choice5> → s5
30: <choice5> → ( not s5 )
31:    <bar6> → <choice6>
32: <choice6> → s6
33: <choice6> → ( not s6 )
```

Figure 5.9: Grammar for Balance Function

condition-action pairs. As is evident from production 1, the grammar for this experiment creates a cond containing exactly 10 conditional statements. Productions 2 through 11 are then dedicated to the creation of boolean classifiers for the digits 0 through 9 respectively.

As evidenced by production 12, the condition of all ten statements consists of a LISP and function followed by seven values generated by the <bar$_i$> non-terminals. When expanded, each produces the boolean state of the corresponding diode - either s_i or `( not` s_i `)`. Figure 5.10 shows the genotype and phonotype of one arbitarily created rule.

The crossover and mutation specification lists

(CROSSOVER COND0 COND1 ... COND6 BAR0 BAR1 ... BAR6)
(MUTATE BARS BAR0 BAR1 BAR2 ... BAR6)

are also associated with the grammar in order to limit the scope of the recombination operators. Under these restrictions, mutation may only change the value of one diode at a time. This serves to prevent catastrophic changes to an entire rule. The crossover operator may affect one diode as well, but it is also permitted to exchange an entire conditional statement. By this means, the boolean classifiers within each rule remain isolated from

GENOTYPE

```
( 1
    2 12 13 15 0 0 16 17 0 0 19 21 0 0 22 23 0 0 25 26 0 0 28 29 0 0 31 32 0 0 0 0
    3 12 13 15 0 0 16 17 0 0 19 20 0 0 22 24 0 0 25 26 0 0 28 30 0 0 31 33 0 0 0 0
    4 12 13 14 0 0 16 17 0 0 19 20 0 0 22 23 0 0 25 26 0 0 28 30 0 0 31 32 0 0 0 0
    5 12 13 15 0 0 16 18 0 0 19 21 0 0 22 24 0 0 25 26 0 0 28 29 0 0 31 32 0 0 0 0
    6 12 13 15 0 0 16 18 0 0 19 21 0 0 22 23 0 0 25 27 0 0 28 30 0 0 31 32 0 0 0 0
    7 12 13 14 0 0 16 18 0 0 19 20 0 0 22 23 0 0 25 27 0 0 28 30 0 0 31 33 0 0 0 0
    8 12 13 14 0 0 16 17 0 0 19 20 0 0 22 23 0 0 25 27 0 0 28 29 0 0 31 33 0 0 0 0
    9 12 13 14 0 0 16 17 0 0 19 20 0 0 22 23 0 0 25 27 0 0 28 30 0 0 31 32 0 0 0 0
    10 12 13 15 0 0 16 17 0 0 19 21 0 0 22 23 0 0 25 27 0 0 28 29 0 0 31 33 0 0 0 0
    11 12 13 15 0 0 16 17 0 0 19 20 0 0 22 23 0 0 25 26 0 0 28 30 0 0 31 33 0 0 0 0
0 )
```

PHENOTYPE

```
( cond
  ( ( and ( not s0 ) s1 ( not s2 ) s3 s4 s5 s6 ) 0 )
  ( ( and ( not s0 ) s1 s2 ( not s3 ) s4 ( not s5 ) ( not s6 ) ) 1 )
  ( ( and s0 s1 s2 s3 s4 ( not s5 ) s6 ) 2 )
  ( ( and ( not s0 ) ( not s1 ) ( not s2 ) ( not s3 ) s4 s5 s6 ) 3 )
  ( ( and ( not s0 ) ( not s1 ) ( not s2 ) s3 ( not s4 ) ( not s5 ) s6 ) 4 )
  ( ( and s0 ( not s1 ) s2 s3 ( not s4 ) ( not s5 ) ( not s6 ) ) 5 )
  ( ( and s0 s1 s2 s3 ( not s4 ) s5 ( not s6 ) ) 6 )
  ( ( and s0 s1 s2 s3 ( not s4 ) ( not s5 ) s6 ) 7 )
  ( ( and ( not s0 ) s1 ( not s2 ) s3 ( not s4 ) s5 ( not s6 ) ) 8 )
  ( ( and ( not s0 ) s1 s2 s3 s4 ( not s5 ) ( not s6 ) ) 9 )
)
```

Figure 5.10: Rule Generated from LED Grammar

one another. The syntactic GA therefore processes ten separate subpopulations of boolean classifiers simultaneously.

Each rule $\mathcal{A}$ in the population is evaluated by the fitness function

$$f_{\mathcal{A}} = \begin{cases} C - I & \text{if } C - I > 0 \\ 2^{(C-I)} & \text{otherwise} \end{cases}$$

where C is the number of LED displays correctly classified and I is the number of incorrect instances. Once again, the $2^{(C-I)}$ calculation ensures that poorer performing rules receive lower fitness values while ensuring that a positive result is always produced. This basic mechanism was often found to be useful in the design of fitness functions.

The experiment was conducted using a crossover rate of 0.8, a mutation rate of 0.05, and a population size of 200. For each of the 50 experimental run, GERDS was able to isolate the optimal solution presented in Figure 5.11 within 150 generations. This solution

```
( cond
   ( ( and s0 s1 s2 ( not s3 ) s4 s5 s6 ) 0 )
   ( ( and ( not s0 ) ( not s1 ) s2 ( not s3 ) ( not s4 ) s5 ( not s6 ) ) 1 )
   ( ( and s0 ( not s1 ) s2 s3 s4 ( not s5 ) s6 ) 2 )
   ( ( and so ( not s1 ) s2 s3 ( not s4 ) s5 s6 ) 3 )
   ( ( and ( not s0 ) s1 s2 s3 ( not s4 ) s5 ( not s6 ) ) 4 )
   ( ( and s0 s1 ( not s2 ) s3 ( not s4 ) s5 s6 ) 5 )
   ( ( and s0 s1 ( not s2 ) s3 s4 s5 s6 ) 6 )
   ( ( and s0 ( not s1 ) s2 ( not s3 ) ( not s4 ) s5 ( not s6 ) ) 7 )
   ( ( and s0 s1 s2 s3 s4 s5 s6 ) 8 )
   ( ( and s0 s1 s2 s3 ( not s4 ) s4 s5 ( not s6 ) ) 9 )
)
```

Figure 5.11: Solution to the LED Experiment

was obtained by first discovering a "near hit" to one or more of the ten classes. Since noise in the training data created many different diode states for each digit, a "near hit" was usually not difficult to find. Mutation and crossover would then gradually alter the classifier for that digit until the correct conditions were encountered. These classifiers were then propagated throughout the population until convergence was achieved.

As stated earlier, a binary GA would certainly be able to encode the states of the seven diodes into binary chromosomes; however, solving the problem as presented in this section would still present difficulties. The most obvious solution would entail the decomposition of the experiment into ten separate binary GA problems. This however, would not produce a general classifier in the same sense as the syntactic GA does, and would require a significantly larger overall population. In any event, the expressive advantages of using a high-level language are clear.

5.3 Badge Function

Every person in attendance at the Eleventh International Conference on Machine Learning[1] and the Seventh ACM Conference on Computational Learning Theory[?] received a name badge labeled with a "+" or "-". This labeling was due to some function known only to the person who generated the badges, and depended only upon the position of characters in the attendee's name. The purpose of the experiment is to identify the unknown function using Machine Learning techniques. Since a solution to this problem requires finding a possibly complex interrelationship between characters in a string, a high-level semantic representation is required. As such, the binary GA is not well-suited to this task. The syntactic genetic algorithm offers a more viable solution strategy.

For this experiment, the syntactic GA was presented with 294 names, 210 of which were classified as "+" and 84 of which were "-" instances. Because of the limitations of LISP string processing, each name was transformed into a list of exactly 24 characters, the length of the longest name in the dataset. Figure 5.12 shows examples of the training data. The tilde character represents NULL characters at the end of names whose length is less than

```
( t   ( N A O K I _ A B E ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ) )
( nil ( M Y R I A M _ A B R A M S O N ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ) )
( t   ( D A V I D _ W ! _ A H A ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ) )
( t   ( K A M A L _ M ! _ A L I ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ) )
( nil ( E R I C _ A L L E N D E R ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ) )
( t   ( D A N A _ A N G L U I N ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ) )
                            ⋮
( t   ( T A K E F U M I _ Y A M A Z A K I ~ ~ ~ ~ ~ ~ ~ ~ ~ ) )
( t   ( H O L L Y _ Y A N C O ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ) )
( t   ( J O H N _ M ! _ Z E L L E ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ) )
( nil ( T H O M A S _ Z E U G M A N N ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ) )
( t   ( J E A N - D A N I E L _ Z U C K E R ~ ~ ~ ~ ~ ~ ~ ~ ) )
( t   ( D A R K O _ Z U P A N I C ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ) )
```

Figure 5.12: Training Data for Badge Function Experiment

the maximum. To simplify processing, several character-by-character substitutions were also performed: the space characters between names were replaced by underscores; periods located after initials were changed to exclamation points; and any apostrophes were changed to the "%" character. Because the badge function was boolean, instances of the "+" class were changed to `t` and "-" became `nil`.

In designing a grammar for this experiment, it was necessary to decide how each generated rule would examine the badge names. LISP, of course, provides a variety of functions for handling lists. For reasons of generality, however, rules were limited to the basic `nth` function, which returns the item found at a specified position within a sequence. Simple equality was used to compare the `nth` item with one of the permissible characters in the problem domain.

Figure 5.13 shows the grammar used for the badge function experiment. Each rule is a simple IF-THEN statement whose condition is in disjunctive normal form. The action, contained in production 1, simply returns the `t` LISP primitive to indicate classification as

```
 1:            <rule> → ( if <condition> t )
 2:       <condition> → ( or <term> <term> <term> <term> <term>
                               <term> <term> <term> )
 3:            <term> → <or-choice>
 4:       <or-choice> → <and-expression>
 5:       <or-choice> → ε
 6:  <and-expression> → <expression>
 7:  <and-expression> → ( and <expression> <and-choice2> )
 8:     <and-choice2> → <and2>
 9:            <and2> → <expression>
10:            <and2> → <expression> <and-choice3>
11:     <and-choice3> → <and3>
12:            <and3> → <expression>
13:            <and3> → <expression> <and-choice4>
14:     <and-choice4> → <and4>
15:            <and4> → <expression>
16:            <and4> → <expression> <and-choice5>
17:     <and-choice5> → <and5>
18:            <and5> → <expression>
19:            <and5> → <expression> <and-choice6>
20:     <and-choice6> → <and6>
21:            <and6> → <expression>
22:            <and6> → <expression> <and-choice7>
23:     <and-choice7> → <and7>
24:            <and7> → <expression>
25:            <and7> → <expression> <expression>
26:      <expression> → ( equal <index> <letter> )
27:           <index> → ( nth  <position> badge )
28:          <letter> → <character>
29:       <character> → 'a
30:       <character> → 'b
31:       <character> → 'c
                      ⋮
54:       <character> → 'z
55:       <character> → '-
56:       <character> → '!
57:       <character> → '‾
58:       <character> → '%
59:       <character> → '_
60:        <position> → <number>
61:          <number> → 0
62:          <number> → 1
                      ⋮
84:          <number> → 23
```

Figure 5.13: Grammar for Badge Function Experiment

a "+" instance. The inclusion of

`<or-choice> → ε`

in the branch structure of productions 3 through 5 allows each `<or-choice>` to be expanded by an epsilon transition. Thus, each of the eight instances of `<term>` in production 2 is optional. The condition of each "badge function" therefore contains between 0 and 8 alternatives. The choice of eight as a maximum length was arbitrary. Note that in LISP, an or statement with no arguments evals to `nil`.

If expanded, each term in the or function becomes either a nth test by means of production 6 or an and statement conjoining up to eight tests if production 7 is selected. Once again, the choice of eight as a cutoff point was arbitrary. The various `<and-choice_i>` non-

GENOTYPE

```
( 1 2
   3 5 0 0
   3 4 6 26 27 60 70 0 0 0 28 33 0 0 0 0 0 0
   3 5 0 0
   3 5 0 0
   3 4 7 26 27 60 74 0 0 0 28 58 0 0 0 8 10 26 27 60 63 0 0 0 28 40 0 0 0 11 12 26 27 60 62
      0 0 0 28 35 0 0 0 0 0 0 0 0 0 0
   3 4 6 26 27 60 74 0 0 0 28 33 0 0 0 0 0 0
   3 4 7 26 27 60 83 0 0 0 28 34 0 0 0 8 9 26 27 60 72 0 0 0 28 50 0 0 0 0 0 0 0 0
   3 4 6 26 27 60 66 0 0 0 28 41 0 0 0 0 0 0 0 0 )
```

PHENOTYPE

```
   ( if
      ( or
         ( equal ( nth 9 badge ) 'e )
         ( and
            ( equal ( nth 13 badge ) '% )
            ( equal ( nth 2 badge ) 'l )
            ( equal ( nth 1 badge ) 'g )
         )
         ( equal ( nth 13 badge ) 'e )
         ( and ( equal ( nth 22 badge ) 'f ) ( equal ( nth 11 badge ) 'v ) )
         ( equal ( nth 5 badge ) 'm ) )
      )
         t
   )
```

Figure 5.14: Rule Generated from Badge Function Grammar

terminals are similiar in affect to the branching structure presented in Section 5.1; thus, productions 8 through 21 act to bias the generation of rules in favor of **and** statements of lower complexity.

Production 27 creates the call to **nth**, where **badge** is set to each name in the training data during rule evaluation. Because each name contains 24 characters, the indexes 0 through 23 are obtainable from productions 61 through 84 in the grammar. Production 26 generates the actual comparison of letters, with productions 28 through 59 providing the choice of permissible characters. Figure 5.13 presents a sample rule which has been "randomly" derived from the experiment grammar.

The fitness function for this experiment is identical to those of the previous two problems. Each rule $\mathcal{A}$ in the population is evaluated by the function

$$f_{\mathcal{A}} = \begin{cases} C - I & \text{if } C - I > 0 \\ 2^{(C-I)} & \text{otherwise} \end{cases}$$

where C is the number of badges correctly identified and I is the number incorrectly classified. As seen in Figure 5.15 below, the optimal solution to the problem was surprisingly simple: members of the "+" have a vowel as their second character.

Utilizing a population of 100 rules, a 0.6 crossover rate and a 0.03 mutation rate, the syntactic GA was able to reach this solution within 150 generations in each of the 50 experimental runs. This success can in a large part be attributed to the relatively easy task of isolating the trace schema

50 53 0 0

corresponding to position 1. Once discovered for at least one of the vowels, different letter combinations at this location were gradually explored in subsequent generations until a correct solution was achieved. A traditional GA approach, whose schemata depend upon the position of bits within the chromosome, would be unable to employ this strategy. Although the solution to this experiment is not complex, the variable structure of the rules and the semantic nature of the badge function problem make it unsuitable for traditional GAs.

5.4 Grammatical Induction

Because the syntactic GA depends upon the construction of rules from a problem-specific grammar, one interesting search problem involves the inductive learning of context-free grammars from language examples. The following experiment can be easily generalized to many applications in compiler construction, finite automata, programming languages and computation theory. It also serves as the basis for the "meta-rule" discovery process described in Chapter 6 of this dissertation.

All of the decisions regarding the design, structure and length of the production rules created by this experiment were arbitrarily determined. Figure 5.16 presents a grammar for generating a set of production rules over the non-terminal symbols $\mathcal{N}$ = {S T U V W X Y Z}

```
( if
    ( or
        ( equal ( nth 1 badge ) 'e )
        ( equal ( nth 1 badge ) 'u )
        ( equal ( nth 1 badge ) 'a )
        ( equal ( nth 1 badge ) 'o )
        ( equal ( nth 1 badge ) 'i )
    )
        t
)
```

Figure 5.15: Badge Function Solution

```
 1:            <grammar> → ( <start> <rule> <rule> <rule> <rule> <rule>
                               <rule> <rule> <rule> <rule> )
 2:              <start> → ( S → <symbol2> )
 3:               <rule> → ( <nonterminal-symbol> → <symbol1> )
 4:               <rule> → ε
 5:            <symbol1> → <choice1>
 6:            <choice1> → <terminal-symbol>
 7:            <choice1> → <terminal-symbol><symbol2>
 8:            <symbol2> → <choice2>
 9:            <choice2> → <any-symbol>
10:            <choice2> → <any-symbol><symbol3>
11:            <symbol3> → <choice3>
12:            <choice3> → <any-symbol>
13:            <choice3> → <any-symbol><symbol4>
14:            <symbol4> → <choice4>
15:            <choice4> → <any-symbol>
16:            <choice4> → <any-symbol><symbol5>
17:            <symbol5> → <choice5>
18:            <choice5> → <any-symbol>
19:            <choice5> → <any-symbol><any-symbol>
20:         <any-symbol> → <symbol>
21:             <symbol> → <nonterminal-symbol>
22:             <symbol> → <terminal-symbol>
23: <nonterminal-symbol> → <nonterminal>
24:        <nonterminal> → T
25:        <nonterminal> → U
26:        <nonterminal> → V
27:        <nonterminal> → W
28:        <nonterminal> → X
29:        <nonterminal> → Y
30:        <nonterminal> → Z
31:    <terminal-symbol> → <terminal>
32:           <terminal> → a
33:           <terminal> → b
```

Figure 5.16: Grammar for Grammatical Induction

and the terminal symbols $\mathcal{T} = \{\mathbf{a}\ \mathbf{b}\}$. The non-terminal S is treated as a unique start symbol, and appears only once in each generated grammar as the left-hand side of the first production. The expansion of S can contain symbols in the set $\mathcal{N} \cup \mathcal{T}$. The seven non-terminals T through Z can appear an arbitrary number of times in each grammar. If one of these symbols appears on the left-hand side of a production, its expansion will contain a non-terminal as the first symbol. Subsequent symbols can be members of either $\mathcal{N}$ or $\mathcal{T}$. This grammar structure guarantees that parsing will not lead to infinite recursion, as the expansion of each production must eliminate at least one terminal symbol in the example string.

Production 1 in the grammar generates ten transition rules. The single instance of the start symbol is created by production 2, while the remaining nine rules are derived from the branch structure of productions 3 and 4. As described in Section 5.3, the use of

$$\texttt{<rule>} \rightarrow \epsilon$$

allows the non-terminal symbol `<rule>` to be expanded by an epsilon transition. Thus, each of the nine instances of `<rule>` in production 1 is optional. Since the `<start>` non-terminal is always expanded, each population member will therefore contain between 1 and 10 productions.

GENOTYPE

```
( 1 2 8 10 20 21 23 27 0 0 0 0 11 13 20 21 23 28 0 0 0 0 14 15 20 22 31 32 0 0 0 0 0 0 0 0 0 0
   4 0
   3 23 24 0 0 5 7 31 33 0 0 8 10 20 21 23 24 0 0 0 0 11 13 20 21 23 27 0 0 0 14 16 20 22 31 32
      0 0 0 0 17 19 20 22 31 33 0 0 0 0 20 21 23 27 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   4 0
   4 0
   4 0
   3 23 24 0 0 5 7 31 32 0 0 8 10 20 21 23 26 0 0 0 0 11 12 20 22 31 33 0 0 0 0 0 0 0 0 0 0
   4 0
   3 23 26 0 0 4 6 31 32 0 0 0 0 0
   4 0 0 )
```

PHENOTYPE

```
(
   ( S → WXa )
   ( T → bTWabW )
   ( T → aVb )
   ( V → a )
)
```

Figure 5.17: Rule Generated for Grammatical Induction

The biasing branch structure of the <symbol$_i$> non-terminals in productions 5 through 19 has also been seen before in previous grammars. Each production <rule> has a 50% chance of expanding to one symbol, a 25% chance of expanding to two, a 12.5% chance of expanding to three, a 6.25% chance of expanding to four, and a 3.125% chance of expanding to either five or six symbols. The grammar depicted in Figure 6.8 is therefore capable of generating over $3.72 * 10^{52}$ different "grammar rules" with varying degrees of probability in accordance with the rules stated above. Productions 23 through 30 create members of set $\mathcal{N}$ while 31 through 33 produce members of $\mathcal{T}$. Finally, the branch structure of productions 20 through 22 generate symbols in $\mathcal{N} \cup \mathcal{T}$. Figure 5.17 shows one possible rule derivable from the grammar.

Figure 5.18 presents some of the 200 items used as training data for the inductive

```
( T   ( babaab ) )
( NIL ( a ) )
( T   ( bbba ) )
( NIL ( bbaab ) )
          ⋮
( T   ( abbaba ) )
( NIL ( bbb ) )
( T   ( ab ) )
( NIL ( babbb ) )
```

Figure 5.18: Training Data for Grammatical Induction

learning of a grammar for parsing strings in the

$$\mathcal{L} = ((a + b)(a + b))^+$$

language. The training file was evenly divided into 100 positive and 100 negative instances, as indicated by the T and NIL classifications respectively. Note that since the language $\mathcal{L}$

contains an infinite number of strings, it is impossible to create a training file containing every positive instance.

Each "grammar rule" $\mathcal{A}$ in the population is evaluated by the fitness function

$$f_{\mathcal{A}} = \begin{cases} C - I + (1 - \frac{P}{10}) & \text{if } C - I > 0 \\ 2^{(C - I + \frac{P}{10})} & \text{otherwise} \end{cases}$$

where C is the number of strings in in the training set correctly parsed by C as a member of $\mathcal{L}$, I is the number of strings incorrectly parsed as $\mathcal{L}$ members, and P is the number of productions contained in $\mathcal{A}$. The $C - I$ evaluation in $f_{\mathcal{A}}$ was presented in the previous three experiments. The additional use of the term $\frac{P}{10}$ serves as a secondary fitness measure which rewards grammars containing fewer productions. This acts to eliminate unnecessary or duplicate productions from the "grammar rules" during the course of the experiment.

This experiment was conducted over 25 experimental runs using a population of 300 rules, a 0.75 crossover rate and a 0.02 mutation rate. The maximum $C - I$ value of 100 was reached in all but one case by generation 120. A slow increase in fitness after this point was obtained by minimizing grammar productions by means of the secondary $\frac{L}{10}$ fitness measure. Figure 5.10 presents the optimal grammar for the language $\mathcal{L}$ discovered during

```
(
    ( S → WW )
    ( W → a )
    ( W → b )
    ( W → aWW )
    ( W → bWW )
)
```

Figure 5.19: Solution to $((a + b)(a + b))^+$

generation 46 of the first experimental trial.

In additional experiments, GERDS was able to discover other grammars expressing a variety of formal languages. This success can largely be attributed to the high-level semantic representation adopted by the syntactic GA, which allows the overall structure of the solution to be adapted during grammar induction. The traditional GA, on the other hand, must maintain one fixed-length structure for each problem. Its adaptive and expressive capabilities are therefore limited to problems whose solutions can be easily expressed in a constant format.

5.5 Artificial Life

The syntactic genetic algorithm closely mimics the evolutionary paradigm of natural selection. Operators exist which resemble the reproductive, selective and variative mechanisms found in nature. It is therefore a logical progression to utilize the GA to study and model evolutionary behavior by creating a population of artificial "animals". This type of investigation is frequently referred to as **Artificial Life** in the literature[191, 232, 271]. Although the binary genetic algorithm is capable of modeling artificial populations as well, it can not efficiently handle the more complex structures used in this experiment.

The environment for this problem consists of a topologically toroidal 200 x 200 grid. Each cell in the grid may contain one item: an animal, a rock, or a piece of food. The goal for each animal is to move through the "world" in order to obtain food, which is necessary for its survival. To accomplish this task, they must learn to maneuver around rock hazards and to avoid stronger animals which might prey on them. Figure 5.20 shows a portion of this environment. The "@" symbol represents the position of an animal, the "Ψ" character is the location of a food stuff, and the "Δ" represents the position of a rock. Empty spaces

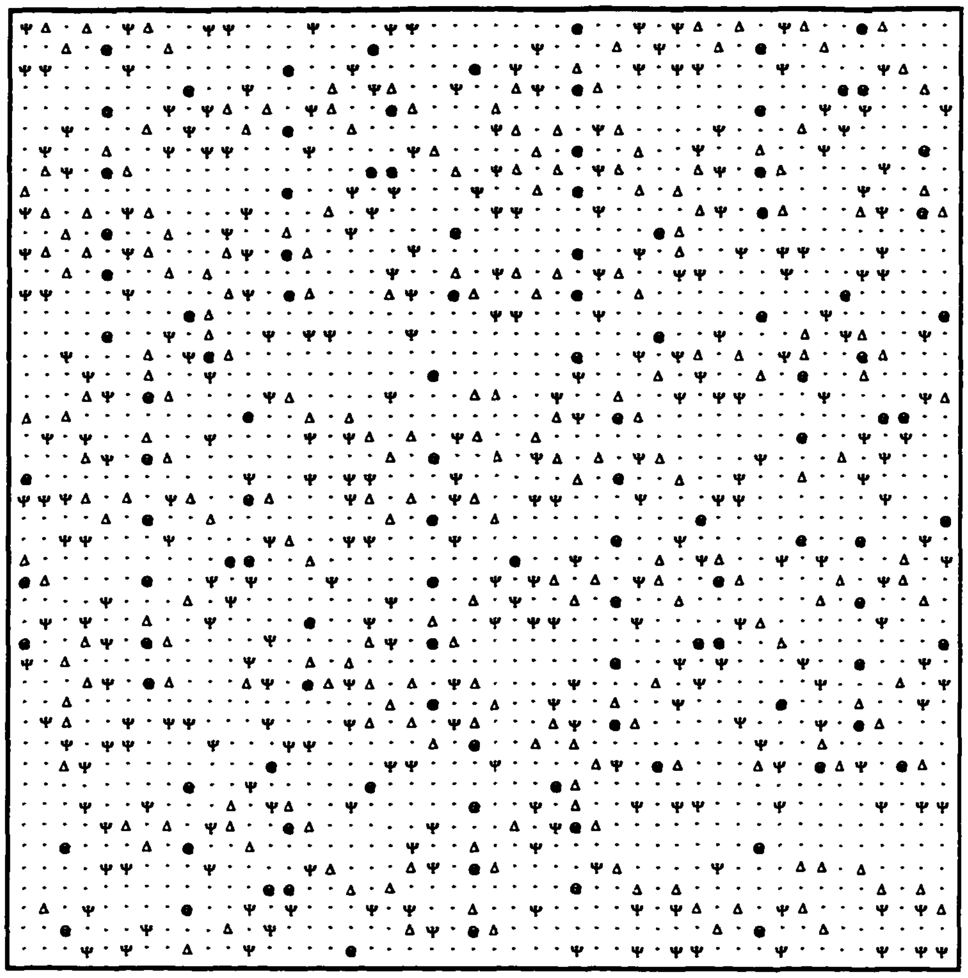


Figure 5.20: Environment for Artificial Life Experiment

in the grid are depicted as a "." character. For each generation, 1000 animals, 2500 rocks and 5000 pieces of food are arbitrarily placed in the world grid.

An animal may attempt to move into any of the four adjoining cells. A space containing a rock may not be entered. If an item of food is encountered at the new location, the animal consumes it and the cell is cleared of food. An animal is also allowed to move onto a space containing a weaker animal. In this event, the stronger animal "survives" and the weaker falls victim to predation. It is removed from the population as it is consumed. Movement onto a space containing an animal of equal or greater strength is prohibited, where strength

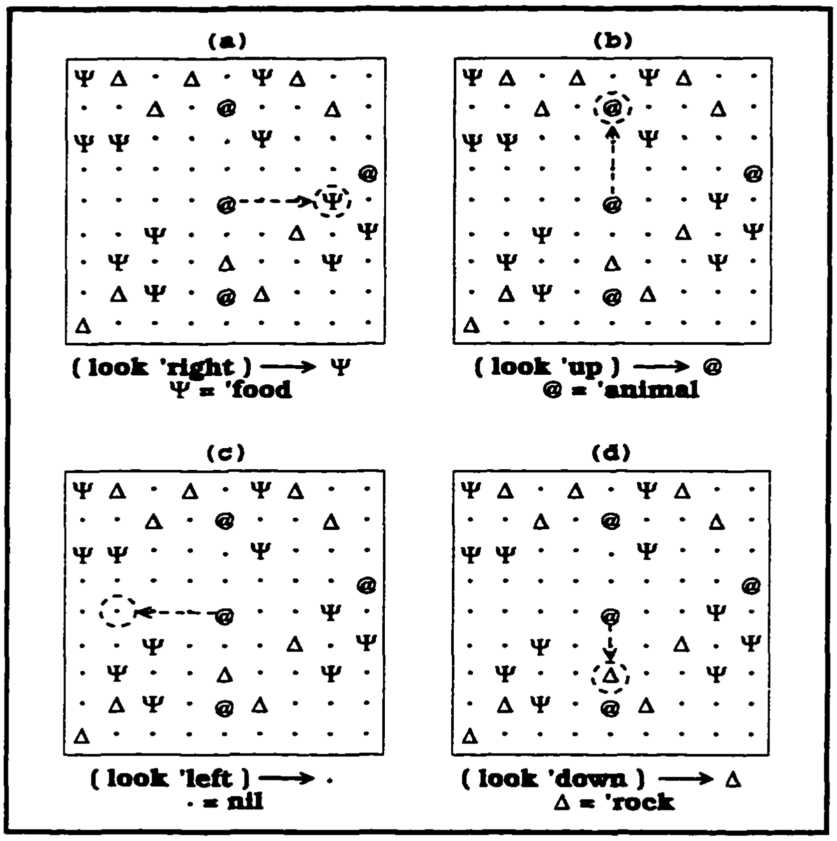


Figure 5.21: The *look* primitive

is expressed as an integer between 1 and 4 inclusive.

The animals obtain information about their surroundings by means of two "senses" - sight and sound. Figure 4.21 graphically illustrates the operation of the vision primitive *look*. Note that this ability is not passive. The animal "mind" must decide to look and specify a direction. Line of sight then extends from an adjacent cell in the stated direction until an object is encountered or the maximum range of vision is exceeded. The primitive then returns the type of object last examined. The vision range is specified by an integer in the interval [1..4]. For the preceding examples, it is assumed that this value is set to 3.

The `listen` primitive operates in a similar fashion. It also requires the animal to actively

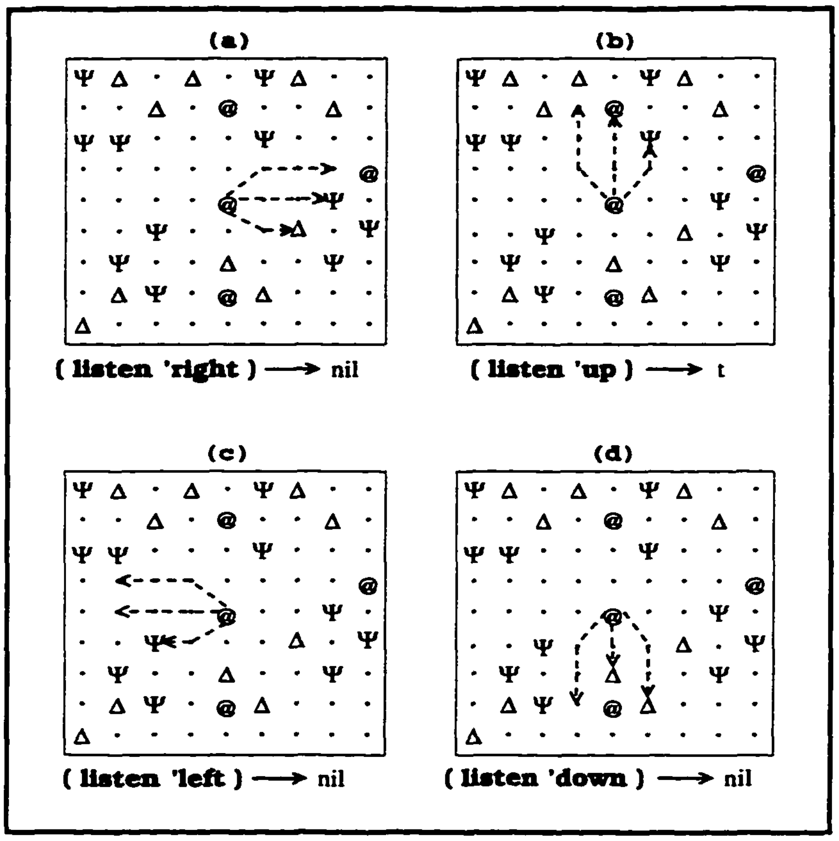


Figure 5.22: The *listen* primitive

specify a direction. Unlike **look**, however, which operates only upon one row or column in the world grid, **listen** examines three adjacent rows or columns simultaneously. As shown in Figure 4.22, each is traversed until either an object is encountered or the maximum range of hearing is exceeded. If an animal was discovered at any of these three positions, **listen** evaluates to true. Hearing range is also an integer between 1 and 4 inclusive, and is assumed to equal 3 in the examples.

During each discrete time step of the experiment, all animals are permitted to examine their environment by means of the **look** and **listen** primitives and, as a result, possibly move to an adjacent location. The timing of these activities depends upon the animal's movement rate. This value is once again an integer in the interval [1..4], where 4 represents the highest speed. All animals of the same movement rate perform their actions in an arbitrary order.

Figure 5.23 presents the grammar for the artificial life experiment. From production 1, it can be seen that each generated rule consists of five parts. The first four of these are integers: the `<see>` non-terminal defines the range of vision; `<hear>` creates the equivalent value for the `listen` primitive; the `<hunt>` symbol generates the strength of each animal used in determining predation; and `<move>` provides the rate of movement. These are created by means of productions 2 through 5 respectively. The actual value of these integers are then determined by the branch structure headed by production 44.

The `<mind>` non-terminal in production 6 is expanded to produce the final part of each rule. It creates a LISP cond statement which acts as a driver program for the individual animal. As suggested by production 7, this statement may contain between 1 and 6 separate condition-action pairs, where 6 was arbitrarily chosen as the limit. Each condition contains expressions such as

```
( equal ( look 'right ) 'rock )
   ( not ( listen 'down ) )
```

designed to test the values returned by the two primitives. Each action, determined by production 34, is simply a request to move in a specific direction. The expansion of the `<exp>` term accounts for the first condition-action from 7. The remaining five may or may not be present, depending on whether the epsilon transition is followed in the branch structure of productions 8, 9 and 10.

Variations of productions 11 through 26 have been seen before in several experiments. Together, they create **and** statements of increasing length with decreasing probability. Each

```
1:            <start> → ( <see> <hear> <hunt> <speed> <mind> )
2:              <see> → ( sees <range> )
3:             <hear> → ( hears <range> )
4:             <hunt> → ( hunts <range> )
5:            <speed> → ( moves <moves> )
6:             <mind> → ( thinks <cond> )
7:             <cond> → ( cond <exp> <term> <term> <term> <term> <term>)
8:             <term> → <expand>
9:           <expand> → <exp>
10:          <expand> → ε
11:             <exp> → ( <and-condition> <action> )
12:   <and-condition> → <and-expression>
13:  <and-expression> → <not-condition>
14:  <and-expression> → ( and <and2> ) )
15:            <and2> → <and2-expand>
16:     <and2-expand> → <not-condition> <not-condition>
17:     <and2-expand> → <and3> <not-condition> <not-condition>
18:            <and3> → <and3-expand>
19:     <and3-expand> → <not-condition>
20:     <and3-expand> → <and4> <not-condition>
21:            <and4> → <and4-expand>
22:     <and4-expand> → <not-condition>
23:     <and4-expand> → <and5> <not-condition>
24:            <and5> → <and5-expand>
25:     <and5-expand> → <not-condition>
26:     <and5-expand> → <not-condition> <not-condition>
27:   <not-condition> → <not-expression>
28:  <not-expression> → ( not <test>)
29:  <not-expression> → <test>
30:            <test> → <sense>
31:           <sense> → ( equal <looks> <item>)
32:           <sense> → ( listen <direction>)
33:           <looks> → ( look <direction>)
34:          <action> → ( move <direction>)
35:            <item> → <object-seen>
36:     <object-seen> → 'animal
37:     <object-seen> → 'rock
38:     <object-seen> → 'food
39:       <direction> → <way>
40:             <way> → 'left
41:             <way> → 'right
42:             <way> → 'up
43:             <way> → 'down
44:           <range> → <value>
45:           <value> → 1
46:           <value> → 2
47:           <value> → 3
48:           <value> → 4
```

Figure 5.23: Grammar for the Artificial Life Experiment

statement can contain up to 6 terms, as six was determined to provide a favorable tradeoff between processing speed and rule expressiveness. Each term is then placed into a not function with a 50% probability through the actions of the branch structure at productions 27, 28 and 29. Productions 30 through 32 are used to decide which of the sensory primitives - either `look` or `listen` - are contained in each term. The remainder of the grammar is used to choose the direction and object specifications required by these senses. Figure 5.24 shows one possible "animal" created by the grammar for this experiment.

In order to bias the system in favor of more efficient structures, a food interval was

GENOTYPE

```
( 1 2 44 47 0 0 0 3 44 45 0 0 0 4 44 45 0 0 0 5 44 46 0 0 0
   6 7 11 12 14 15 16 27 28 30 32 39 42 0 0 0 0 0 27 29 30 31 33 39 41 0 0 0 35 37
      0 0 0 0 0 0 0 0 0 0 34 39 40 0 0 0 0 0 0
   8 10 0 0
   8 10 0 0
   8 11 13 27 29 30 32 39 41 0 0 0 0 0 0 0 34 39 40 0 0 0 0 0
   8 10 0 0
   8 11 13 27 28 30 31 33 39 43 0 0 0 35 36 0 0 0 0 0 0 34 39 42 0 0 0 0 0 0 )
```

PHENOTYPE

```
(
   ( sees 3 ) ( hears 1 ) ( hunts 1 ) ( moves 2 )
   ( thinks
      ( cond
         (
            ( and ( not ( listen 'up ) ) ( equal ( look 'right ) 'rock ) )
               ( move 'left )
         )
         (
            ( listen 'right )
               ( move 'left )
         )
         (
            ( not ( equal ( look 'down ) 'animal ) )
               ( move 'up)
         )
      )
   )
)
```

Figure 5.24: Rule Generated from Artificial Life Grammar

calculated for each animal. This value determined the number of time steps an animal could survive without finding food. If an animal exceeded this individual limit, it was removed from the population and replaced by a piece of food in the world grid. For each animal $\mathcal{A}$, the food interval $I_{\mathcal{A}}$ was calculated as

$$I_{\mathcal{A}} = 40 - (V_{\mathcal{A}} + H_{\mathcal{A}} + S_{\mathcal{A}} + M_{\mathcal{A}} + \lceil \log_2 |G(\mathcal{A})| \rceil)$$

where $V_{\mathcal{A}}$ is the vision range, $H_{\mathcal{A}}$ is the hearing range, $S_{\mathcal{A}}$ is the strength and $M_{\mathcal{A}}$ is the movement rate of $\mathcal{A}$, and $G(\mathcal{A})$ is the genotype of the rule. Grammar 5.23 produces rules whose derivation traces have at least 62 and at most 786 elements; thus, $I_{\mathcal{A}}$ ranged in value from 14 to 30. By using the food interval mechanism, more complex animals had to exhibit

```
( ( sees 4 ) ( hears 1 ) ( hunts 1 ) ( moves 4 )
  ( mind
    ( cond
      (
        ( equal ( look 'left ) 'food )
          ( move 'left )
      )
      (
        ( equal ( look 'right ) 'food )
          ( move 'right )
      )
      (
        ( equal ( look 'up ) 'food )
          ( move 'up )
      )
      (
        ( not ( equal ( look 'left ) 'rock ) )
          ( move 'left )
      )
      (
        ( not ( equal ( look 'up ) 'rock ) )
          ( move 'up )
      )
    )
  )
)
```

Figure 5.25: A *"Gatherer"* Solution to Artificial Life Experiment

more complex behavior in order to survive.

For each generation, the simulation was continued until the last animal exceeded its food interval. Fitness was the calculated as

$$f_{\mathcal{A}} = LS(\mathcal{A})$$

where LS(A) was the life span of each animal. The simulation was initiated at time step 1, so the resulting functional value was guaranteed to be positive even in the event of immediate predation. The experiment was conducted over the course of 5000 generations utilizing a 0.4 crossover rate and a 0.01 mutation rate.

The open-ended nature of this problem precluded the convergence of the population toward one single solution; instead, several "species" of animals thrived at different times during the experiment. By far the most prevalent of these was the "gatherer" solution, an example of which is presented in Figure 5.25 on the preceding page. This strategy entailed a relatively straight-forward greedy approach to foraging. The animal would look in several directions to see if food was located nearby. If not, it would move in some direction not containing a rock obstacle. One common species defect was a mutation towards cyclic searching patterns. After clearing an area of the available food supply, the faulty animals would then meander repeatedly through the same locations in the world grid until eventually starving. The "gatherer" solution relied primarily upon a wide range of vision. A secondary trait was a fast movement rate, which allowed the animal to be the first to reach nearby food. Throughout most of the experiment, at least a few "gatherers" survived in the population.

Several times during the run, another variety of solution appeared. As its name implies, the "hunter" solution, an example of which is presented in Figure 5.26, relied upon a stable population of animals. Its solution strategy was primarily sound oriented - the "hunter" would move in any direction were an animal was overheard, as long as a rock was not in an

```
( ( sees 1 ) ( hears 4 ) ( hunts 4 ) ( moves 1 )
  ( mind
    ( cond
      (
        ( and ( not ( equal ( look 'right ) 'rock ) ) ( listen 'right ) )
          ( move 'right )
      )
      (
        ( and ( listen 'up ) ( not ( equal ( look 'up ) 'rock ) ) )
          ( move 'up )
      )
      (
        ( and ( not ( equal ( look 'down ) 'rock ) ) ( listen 'down ) )
          ( move 'down )
      )
    )
  )
)
```

Figure 5.26: A *"Hunter"* Solution to Artificial Life Experiment

adjacent location blocking its movement. Although the "hunter" never explicitly searched for food, it often obtained some simply by maneuvering around rocks. Surprisingly, a slow movement rate was an asset to the "hunter", since other animals would move into hearing

```
( ( sees 4 ) ( hears 1 ) ( hunts 1 ) ( moves 1 )
  ( mind
    ( cond
      (
        ( equal ( look 'down ) 'rock )
          ( move 'left )
      )
      (
        ( not ( equal ( look 'down ) 'animal ) )
          ( move 'down )
      )
    )
  )
)
```

Figure 5.27: A *"Scavenger"* Solution to Artificial Life Experiment

range before it would have to act. Since two animals of equal strength could not prey upon each other, this solution tended to over-populate itself toward extinction.

Finally, Figure 5.27 shows one of the most interesting strategies discovered by the experiment. The "scavenger" depended upon out-surviving other animals. Its solution was non-complex, and the animal therefore possessed a long food interval. The goal of the "scavenger" was to find an animal that was unable to move, and wait for it to die and change to food. By avoiding both rocks and animals in one direction, the "scavenger" was likely to keep moving and encounter other foods as well. By the end of the experiment, the majority of the population contained solutions similar to this one.

This experiment exemplifies the expressive prowess of the syntactic GA. Since the "mind" of each population member is a complex LISP statement, GERDS is capable of performing automatic programming tasks. As the Artificial Life problem encompasses a large search space and requires continuous adaptation, the syntactic GA has been shown to handle complex problems involving several solution strategies.

Chapter 6

Conclusion

6.1 Conclusions

The genetic algorithm (GA) is a robust search technique which has been theoretically and empirically proven to provide efficient search for a variety of problems. Due largely to the semantic and expressive limitations of adopting a bitstring representation, however, the traditional GA has not found wide acceptance in the Artificial Intelligence community. In addition, binary chromosomes can unevenly weight genetic search, reduce the effectiveness of recombination operators, make it difficult to solve problems whose solution schemata are of high order and defining length, and hinder new schema discovery in cases where chromosome-wide changes are required.

The research presented in this dissertation describes a grammar-based approach to genetic algorithms. Under this new paradigm, all members of the population are strings produced by a problem-specific grammar. Any structure which can be expressed in Backus-Naur Form can thus be manipulated by genetic operators. As such, a grammar-based GA strategy provides a consistent methodology for handling any population structure expressible in terms of a context-free grammar.

In order to lend theoretical support to the development of the syntactic GA, the concept of a trace schema - a similarity template for matching the derivation traces of grammar-defined rules - was introduced. An analysis of the manner in which a grammar-based GA operates yielded a Trace Schema Theorem for rule processing, which states that above-average trace schemata containing relatively few non-terminal productions are sampled with increasing frequency by syntactic genetic search. Schemata thus serve as the "building blocks" in the construction of the complex rule structures manipulated by syntactic GAs.

As part of the research presented in this dissertation, the GERDS implementation of the grammar-based GA was developed. A comparison between the performance of GERDS and the traditional GA showed that the class of problems solvable by a syntactic GA is a superset of the class solvable by its binary counterpart. To strengthen that conclusion, several experiments encompassing diverse domains were performed with favorable results.

6.2 Future Research

6.2.1 Efficiency

Because traditional GAs typically operate on binary-coded chromosomes, they can quickly perform crossover, mutation and other recombinant operations. GERDS, on the other hand, requires additional processing time in order to discover syntactically proper rule subexpressions. Although the list processing capabilities of LISP are well-suited to the manipulation of derivation traces, its selection as the programming language for GERDS was due largely to LISP's pervasiveness in Artificial Intelligence. The language is not designed for speed or efficiency, however, and memory-intensive computation is often quite slow. Redeveloping

GERDS using a language such as C++ with a highly-optimized compiler would undoubtably improve running time. The syntactic GA is also adaptable to a parallelized implementation. An investigation of GERDS use on a different architecture is therefore in order.

6.2.2 Branch Productions

As described in Section 3.2.2, many of the grammars provided in this dissertation include branch productions such as

```
   <gene> → <allele>
 <allele> → 0
 <allele> → 1
```

which allow crossover and mutation to exchange alternative structures. Although conceptually straight-forward, these productions increase the length of derivation traces, and thus decrease the efficiency of GERDS. They can also act to obfuscate the basic structure of the grammar. One proposed remedy to this situation involves the use of attribute grammars to internally label productions of similar purpose.

6.2.3 Meta-Rule Discovery

As evidenced by the Grammar Induction experiment of Section 5.4, GERDS is capable of learning transition rules describing a context-free language. Since the syntactic GA itself operates upon a problem-specific grammar, one very interesting avenue of future research, which calls for far greater computing resources than were available for the present project, is the development of a self-adapting genetic algorithm. It is hoped that such a system would be able to acquire both general-purpose and problem-specific methods for improving genetic search. One means of achieving this goal would be the the explicit creation and modification of **meta-rules**.

Each meta-rule in the proposed system would represent a heuristic operator which the GA could use to create new rules in the population. These heuristics would serve as a supplement to the standard mutation and crossover operators. The integration of such meta-rules into the genetic algorithm can be accomplished by introducing a two-level architecture. An **application-level GA** would contain rules describing potential solutions to the experiment at hand. A **meta-level GA** would also exist, consisting of a population of heuristic operators which can be used to generate application-level rules in future generations.

The operators developed by the meta-level GA would manipulate derivation traces in order to produce new rules. Each heuristic would itself be derivable from a "meta-level" grammar and could be expressed as a simple IF-THEN rule. Since the meta-rules operate on derivation traces, the condition of these rules would require only basic list and comparison functions. The action of each heuristic would be a call to either the mutation or crossover routine. As an example, the phenotype of one heuristic might be

(if (equal (nth trace 4) 16) (mutate 12))

which would have the effect of mutating a rule at production 12 if the fifth locus in the derivation trace was 16.

After reproduction, rules from the application level would be sent to the meta-level for possible modification. Each derivation trace would be matched against each heuristic. The operator with the highest fitness would then be selected and applied. New meta-level rules would be created by the standard reproduction, mutation and crossover operations after many generations of the application-level population.

Each time a new rule is created by a heuristic, a link would be established connecting that rule to the meta-rule operator which created it. When the fitness function is applied to a rule, this link would be traversed, and a fraction of the calculated value α would be awarded to the meta-rule. The fitness of a meta-rule would therefore reflects the overall success of the rules which it has created.

If a rules fails to reproduce into a subsequent generation, the link joining it to a meta-rule would be erased. The meta-rule would remain in the system, however, as long as its overall fitness remained high. Similarly, if a meta-rule is deleted from the GA, the links joining it to the rules it created would also be erased. The rules would then operate on their own without calculating an α fraction.

Using an economic metaphor, each meta-rule in the proposed system can be seen as an investment firm periodically making business ventures by creating new rules. If some of these ventures prove successful, it is likely that the corporation will remain competitive. If most of the enterprises fail, however, the corporation will likely go bankrupt and be removed from the population.

Although the method outlined above is intuitively sound, it is evident from the above discussion that processing demands make the system infeasible to implement at present. With future access to more powerful machines, however, a practical investigation of meta-rules can be initiated.

Appendix A

GERDS Source Code

```
;****************************************************************************;
;*                                                                          *;
;*                             COPY-POPULATION                              *;
;*                                                                          *;
;* ARGUMENTS:                                                               *;
;*     nil                                                                  *;
;*                                                                          *;
;* GLOBALS:                                                                 *;
;*     <new-population>   - newly created and evaluated population array    *;
;*     *population*       - total number of population members in experiment *;
;*                                                                          *;
;* RETURNS:                                                                 *;
;*     nil                                                                  *;
;*                                                                          *;
;* EFFECTS:                                                                 *;
;*     <old-population>   - set to <new-population>, then sorted by fitness *;
;*     <sum-of-fitness>   - total sum of all fitness values of individuals  *;
;*                                                                          *;
;****************************************************************************;

(defun copy-population ()

; Purpose: copies and sorts new to old population, calculates fitness sum
; Returns: nil

   (setq <new-population> (sort <new-population> '> :key 'individual-fitness))
   (setq <sum-of-fitness> 0.0)
   (dotimes (index *population*)
      (setf (individual-phenotype (aref <old-population> index))
            (individual-phenotype (aref <new-population> index))
      )
      (setf (individual-genotype (aref <old-population> index))
            (individual-genotype (aref <new-population> index))
      )
      (setf (individual-fitness (aref <old-population> index))
            (individual-fitness (aref <new-population> index))
      )
      (setq <sum-of-fitness>
         (+ (individual-fitness (aref <old-population> index)) <sum-of-fitness>)
      )
   )
)

;****************************************************************************;
;*                                                                          *;
;*                            CREATE-GENOTYPE                               *;
;*                                                                          *;
;* ARGUMENTS:                                                               *;
;*    production        - production at which to start building a new trace *;
;*                                                                          *;
;* GLOBALS:                                                                 *;
;*    <grammar>         - array of grammar productions for building a rule  *;
;*    <lookup>          - list of all nonterminals and their productions    *;
;*    <symbols>         - list of nonterminal symbols in productions        *;
;*                                                                          *;
;* RETURNS:                                                                 *;
;*    a new (sub)trace beginning at production                              *;
;*                                                                          *;
;*                                                                          *;
;****************************************************************************;

(defun create-genotype (production)

; Purpose: randomly constructs a subtrace beginning with production
; Returns: new subtrace

   (append
      (list production)
      (expand-trace (aref <grammar> production))
      (list 0)
   )
)

;----------------------------------------------------------------------------;

(defun choose-production (symbol)

; Purpose: randomly selects one element from a list of productions
; Returns: the selected production
; Invoked: expand-trace
```

```
   (let ((production-list (cdr (assoc symbol <lookup>))))
      (nth (random (length production-list)) production-list)
   )
)

;--------------------------------------------------------------------------;

(defun expand-trace (expansion)

; Purpose: randomly expands a subtrace beginning with expansion
; Returns: new subtrace
; Invoked: create-genotype, expand-trace

   (cond
      ((member expansion <symbols>)
         (create-genotype (choose-production expansion))
      )
      ((and (listp expansion) (not (null expansion)))
         (append (expand-trace (car expansion)) (expand-trace (cdr expansion)))
      )
   )
)

;**************************************************************************;
;*                                                                        *;
;*                           CREATE-PHENOTYPE                             *;
;*                                                                        *;
;* ARGUMENTS:                                                             *;
;*    trace             - a derivation (sub)trace of the rule to be made  *;
;*                                                                        *;
;* GLOBALS:                                                               *;
;*    <grammar>         - the array of productions used to build a new rule *;
;*    <productions>     - a special stack used for faster rule expansion  *;
;*    <symbols>         - list of nonterminal symbols in all productions  *;
;*                                                                        *;
;* RETURNS:                                                               *;
;*    a new (sub)rule built from the trace                                *;
;*                                                                        *;
;**************************************************************************;

(defun create-phenotype (trace)

; Purpose: constructs an rule (sub)expression by expanding (sub)trace
; Returns: a (sub)expression

   (setq <productions> (remove 0 trace))
   (expand-rule (aref <grammar> (pop <productions>)))
)

;--------------------------------------------------------------------------;
```

```
(defun expand-rule (rule)

; Purpose: transforms a subtrace into a subexpression using a stack
; Returns: a subexpression
; Invoked: create-phenotype, expand-rule

  (cond
      ((null rule)
         nil
      )
      ((member rule <symbols>)
         (expand-rule (aref <grammar> (pop <productions>)))
      )
      ((atom rule)
         (list rule)
      )
      ((and (listp rule) (listp (car rule)) (equal (list (car rule)) rule))
         (list (expand-rule (car rule)))
      )
      (t
         (let ((new-rule))
            (dolist (this-rule rule new-rule)
               (if (and
                         (listp this-rule)
                         (equal (list (car this-rule)) this-rule)
                   )
                   (setq
                      new-rule
                      (append new-rule (list (expand-rule (car this-rule))))
                   )
          (setq new-rule (append new-rule (expand-rule this-rule)))
               )
            )
         )
      )
  )
)

;***************************************************************************************;
;*                                                                                     *;
;*                               CREATE-POPULATION                                     *;
;*                                                                                     *;
;*                                                                                     *;
;* ARGUMENTS:                                                                          *;
;*     nil                                                                             *;
;*                                                                                     *;
;* GLOBALS:                                                                            *;
;*     *population*        - the total size of the experiment population               *;
;*                                                                                     *;
;* CALLS:                                                                              *;
;*     (create-genotype)  - builds a production list used to construct rule    *;
;*     (create-phenotype) - constructs a new rule from its derivation trace    *;
;*                                                                             *;
;* RETURNS:                                                                    *;
;*     nil                                                                     *;
;*                                                                             *;
;* EFFECTS:                                                                    *;
;*     <new-population>   - a new array of randomly created individuals       *;
;*     <old-population>   - an array of *population* with default values      *;
;*                                                                             *;
;*******************************************************************************;

(defun create-population ()

; Purpose: drives creation of initial population
; Returns: nil

   (create-population-arrays)
   (dotimes (index *population*)
      (make-new-individual index)
   )
)

;-----------------------------------------------------------------------------;

(defun create-population-arrays ()

; Purpose: creates two population arrays of type individual
; Returns: nil
; Invoked: create-population

   (setq <old-population> (make-array *population*))
   (setq <new-population> (make-array *population*))
   (dotimes (index *population*)
      (setf (aref <old-population> index) (make-individual))
      (setf (aref <new-population> index) (make-individual))
   )
)

;-----------------------------------------------------------------------------;

(defun make-new-individual (index)

; Purpose: creates a <new-population> member and stores it at index
; Returns: new initialized defstruct
; Invoked: create-population

   (let ((child (aref <new-population> index)) (trace (create-genotype 1)))
      (setf (individual-genotype child) trace)
      (setf (individual-phenotype child) (create-phenotype trace))
      (setf (individual-fitness child) 0)
   )
)

;******************************************************************************;
;*                                                                            *;
;*                              CROSSOVER                                     *;
;*                                                                            *;
;* ARGUMENTS:                                                                 *;
;*      nil                                                                   *;
;*                                                                            *;
;* GLOBALS:                                                                   *;
;*     <crossover>        - list of productions where crossover can occur     *;
;*     *crossover-rate*   - probability of performing crossover on traces     *;
;*     <new-population>   - array of individuals chosen from <old-population> *;
;*     *overlap*          - number of top individuals to keep each generation *;
;*     *population*       - total number of individuals in current population *;
;*                                                                            *;
;* CALLS:                                                                     *;
;*     (parity-count)     - determines subtrace end to splice for crossover   *;
;*                                                                            *;
;* RETURNS:                                                                   *;
;*     nil                                                                    *;
;*                                                                            *;
;* EFFECTS:                                                                   *;
;*    <new-population>    - new population of genotypes after crossover       *;
;*                                                                            *;
;******************************************************************************;

(defun cross-end (trace locus)

; Purpose: finds end of list being spliced
; Returns: the end sublist
; Invoked: swap-genes

   (if (not (equal (length trace) locus))
      (subseq trace locus)
   )
)

;------------------------------------------------------------------------------;

(defun cross-start (trace locus)

; Purpose: finds front of list being spliced
; Returns: the front sublist
; Invoked: swap-genes

   (if (not (zerop locus))
      (subseq trace 0 locus)
   )
)
```

```
;-----------------------------------------------------------------------;

(defun crossover ()

; Purpose: controls crossover of genotypes in <new-population>
; Returns: nil

   (do
      ((index *overlap* (+ index 2)))
      ((>= (1+ index) *population*))
      (if (< (random 1.0) *crossover-rate*)
         (perform-splicing index (1+ index))
      )
   )
)

;-----------------------------------------------------------------------;

(defun locus-select (trace1 trace2)

; Purpose: selects a locus for crossover
; Returns: the production to be crossed
; Invoked: perform-splicing

   (let ((loci (trace-intersect trace1 trace2)))
      (let ((choice (trace-intersect loci <crossover>)))
         (nth (random (length choice)) choice)
      )
   )
)

;-----------------------------------------------------------------------;

(defun perform-splicing (index1 index2)

; Purpose: finds segments of rule traces and then does crossover
; Returns: nil
; Invoked: crossover

   (let ((trace1 (individual-genotype (aref <new-population> index1)))
         (trace2 (individual-genotype (aref <new-population> index2))))
      (let ((production (locus-select trace1 trace2)))
         (let ((locus1 (start-swap trace1 production))
               (locus2 (start-swap trace2 production)))
            (let ((end1 (parity-count trace1 locus1))
                  (end2 (parity-count trace2 locus2)))
               (setf (individual-genotype (aref <new-population> index1))
                     (swap-genes trace1 trace2 locus1 locus2 end1 end2)
               )
               (setf (individual-genotype (aref <new-population> index2))
                  (swap-genes trace2 trace1 locus2 locus1 end2 end1)
               )
            )
         )
      )
   )
)

;-----------------------------------------------------------------------;

(defun start-swap (trace production)

; Purpose: finds one of the indices of production in the trace
; Returns: the index
; Invoked: perform-splicing

  (let ((locus trace))
     (dotimes (i (1+ (random (count production trace))))
        (setq locus (cdr (member production locus)))
     )
     (- (length trace) (length locus) 1)
  )
)

;-----------------------------------------------------------------------;

(defun swap-genes (trace-1 trace-2 locus-1 locus-2 end-1 end-2)

; Purpose: performs crossover on one structure
; Returns: a trace after crossover
; Invoked: perform-splicing

  (append
     (cross-start trace-1 locus-1)
     (subseq trace-2 locus-2 end-2)
     (cross-end trace-1 end-1)
  )
)

;-----------------------------------------------------------------------;

(defun trace-intersect (list1 list2)

; Purpose: performs intersection on two traces
; Returns: the intersection list without duplicates or zero
; Invoked: locus-select

  (let ((intersect nil))
     (dolist (item list1 (remove 0 (remove-duplicates intersect)))
        (if (member item list2)
           (setq intersect (append (list item) intersect))
```

```
          )
        )
      )
    )

;***************************************************************************;
;*                                                                         *;
;*                          FITNESS-FUNCTION                               *;
;*                                                                         *;
;* ARGUMENTS:                                                              *;
;*    nil                                                                  *;
;*                                                                         *;
;* GLOBALS:                                                                *;
;*    <new-population>   - newly created individuals without fitness values *;
;*    *population*       - total size of the population for the experiment *;
;*                                                                         *;
;* RETURNS:                                                                *;
;*    nil                                                                  *;
;*                                                                         *;
;* EFFECTS:                                                                *;
;*    <new-population>   - new population with all fitness values calculated *;
;*                                                                         *;
;***************************************************************************;

(defun fitness-function ()

; Purpose: Default fitness function in case one in not supplied
; Returns: nil

  (dotimes (index *population*)
     (setf
        (individual-fitness (aref <new-population> index))
        (eval (individual-phenotype (aref <new-population> index)))
     )
  )
)

;***************************************************************************;
;*                                                                         *;
;*                                GA                                       *;
;*                                                                         *;
;* ARGUMENTS:                                                              *;
;*   experiment-file   - name and path of file containing the experiment   *;
;*                                                                         *;
;* GLOBALS:                                                                *;
;*   *best-display*    - number of best population members to show in output *;
;*   <crossover>       - list of productions where crossover can take place *;
;*   *crossover-rate*  - probability of performing crssover on trace pairs  *;
;*   *fitness-file*    - name and path of file containing fitness function  *;
;*   *generations*     - total number of successive population generations  *;
;*   <grammar>         - array of grammar productions for building new rules *;
;*    *grammar-file*     - name and path pf the file containing grammar info   *;
;*    <lookup>           - a list of all nonterminals and their productions    *;
;*    <mutation>         - list of productions where mutation can take place   *;
;*    *mutation-rate*    - probability of mutating each production in a trace  *;
;*    <new-population>   - array of individuals to evaluate in this generation *;
;*    <old-population>   - saved array of individuals from previous generation *;
;*    *output-file*      - name and path of the file where results are output  *;
;*    <output-stream>    - stream for sending information to the *output-file* *;
;*    *overlap*          - number of top individuals to keep each generation   *;
;*    *population*       - the number of individuals in the current population *;
;*    <productions>      - a special stack used for faster rule expansion      *;
;*    <sum-of-fitness>   - total of all fitness functions values in population *;
;*    <symbols>          - list of nonterminal symbols in grammar productions  *;
;*    *train-file*       - path and name of training data file (if used by ga) *;
;*    <train-list>       - list where training data is stored (if used by ga)  *;
;*                                                                             *;
;* CALLS:                                                                      *;
;*    copy-population    - copies <new-population> into <old-population>       *;
;*    create-population - handles the creation of the initial population       *;
;*    crossover          - controls crossover of genotypes in <new-population> *;
;*    fitness            - location of function is specified in *fitness-file* *;
;*    mutation           - controls mutation of genotypes in <new-population>  *;
;*    process-grammar    - controls reading and proccessing of problem grammar *;
;*    read-grammar       - controls the processing and storage of the grammar  *;
;*    read-training      - controls the creation of the list of training data  *;
;*    reproduction       - chooses individuals to place into <new-population>  *;
;*    statistics         - displays stats for generation via <output-stream>   *;
;*                                                                             *;
;* EFFECTS:                                                                    *;
;*    *best-display*     - given default value if not found in experiment-file *;
;*    <crossover>        - internal global variable is created and set to nil  *;
;*    *crossover-rate*   - given default value if not found in experiment-file *;
;*    *fitness-file*     - given default value if not found in experiment-file *;
;*    *generations*      - given default value if not found in experiment-file *;
;*    <grammar>          - internal global variable is created and set to nil  *;
;*    *grammar-file*     - given default value if not found in experiment-file *;
;*    <lookup>           - internal global variable is created and set to nil  *;
;*    <mutation>         - internal global variable is created and set to nil  *;
;*    *mutation-rate*    - given default value if not found in experiment-file *;
;*    <new-population>   - internal global variable is created and set to nil  *;
;*    <old-population>   - internal global variable is created and set to nil  *;
;*    *output-file*      - given default value if not found in experiment-file *;
;*    <output-stream>    - internal global variable is created and set to nil  *;
;*    *overlap*          - given default value if not found in experiment-file *;
;*    *population*       - given default value if not found in experiment-file *;
;*    <productions>      - internal global variable is created and set to nil  *;
;*    <sum-of-fitness>   - internal global variable is created and set to nil  *;
;*    <symbols>          - internal global variable is created and set to nil  *;
;*    *train-file*       - given default value if not found in experiment-file *;
;*    <train-list>       - internal global variable is created and set to nil  *;
;*                                                                             *;
;* RETURNS:                                                             *;
;*    nil, output of genetic algorithm is printed to <output-stream>    *;
;*                                                                      *;
;************************************************************************;
;                                                                       ;
;               The Structure Of Each Member Of The Population          ;
;                                                                       ;
;-----------------------------------------------------------------------;

(defstruct individual
   genotype
   phenotype
   fitness
)

;-----------------------------------------------------------------------;
;                                                                       ;
;  Default Values For Global Variables Which Can Be Set In Experiment-file  ;
;                                                                       ;
;-----------------------------------------------------------------------;

(defvar *best-display* 5)
(defvar *crossover-rate* 0.6)
(defvar *fitness-file* "fitness-function.o")
(defvar *generations* 100)
(defvar *grammar-file* "grammar.lsp")
(defvar *mutation-rate* 0.02)
(defvar *output-file* "output.lsp")
(defvar *overlap* 0)
(defvar *population* 100)
(defvar *train-file* nil)

;-----------------------------------------------------------------------;
;                                                                       ;
;            Internal Global Variables Are Initialized to Nil           ;
;                                                                       ;
;-----------------------------------------------------------------------;

(defvar <crossover> nil)
(defvar <grammar> nil)
(defvar <lookup> nil)
(defvar <mutation> nil)
(defvar <new-population> nil)
(defvar <old-population> nil)
(defvar <output-stream> nil)
(defvar <productions> nil)
(defvar <sum-of-fitness> nil)
(defvar <symbols> nil)
(defvar <train-list> nil)

;-----------------------------------------------------------------------;
```

```
;                                                                          ;
;                  Load All Needed Modules For Genetic Algorithm           ;
;                                                                          ;
;--------------------------------------------------------------------------;

(load "copy-population.o")
(load "create-genotype.o")
(load "create-phenotype.o")
(load "create-population.o")
(load "crossover.o")
(load "modify-grammar")
(load "mutation.o")
(load "parity-count.o")
(load "process-grammar.o")
(load "read-grammar.o")
(load "read-training.o")
(load "reproduction.o")
(load "statistics.o")

;--------------------------------------------------------------------------;

(defun ga (experiment-file)

; Purpose: conducts a genetic algorithm experimemt using rules instead of bits
; Returns: nil, output of ga experiment is printed to *output-stream*

   (load experiment-file)
   (load *fitness-file*)
   (process-grammar)
   (if *train-file* (read-training))
   (create-population)
   (let ((<output-stream> (open *output-file* :direction :output)))
      (dotimes (generation *generations*)
         (funcall 'fitness)
         (copy-population)
         (statistics generation)
         (reproduction)
         (crossover)
         (mutation)
      )
      (close <output-stream>)
   )
)

;**************************************************************************;
;*                                                                        *;
;*                              MODIFY-GRAMMAR                            *;
;*                                                                        *;
;* ARGUMENTS:                                                             *;
;*    grammar-list       - a list of the unmodified grammar productions   *;
;*                                                                        *;
;* GLOBALS:                                                                  *;
;*     <crossover>        - list of specified crossover nonterminals or nil  *;
;*     <mutation>         - list of specified mutation nonterminals or nil   *;
;*                                                                           *;
;* RETURNS:                                                                  *;
;*     a modified version of all grammar rules for easier rule construction  *;
;*                                                                           *;
;* EFFECTS:                                                                  *;
;*     <crossover>        - list of all productions where crossover can occur *;
;*     <lookup>           - list of all nonterminals and their productions   *;
;*     <mutation>         - list of all productions where mutation can occur *;
;*     <symbols>          - list of all gensyms representing nonterminals    *;
;*                                                                           *;
;****************************************************************************;

(defun build-nonterminal-associations (grammar-list)

; Purpose: creates a list associating each nonterminal with its productions
; Returns: an association list between nonterminals and productions
; Invoked: modify-grammar

   (let ((nonterm-assoc nil) (nonterms (mapcar 'car grammar-list)))
     (dolist (nonterminal (remove-duplicates nonterms) nonterm-assoc)
         (setq nonterm-assoc
            (append
               nonterm-assoc
               (list (list nonterminal (find-positions nonterminal nonterms)))
            )
         )
      )
   )
)

;---------------------------------------------------------------------------;

(defun find-positions (key nonterminal-list)

; Purpose: finds all grammar rules whose left side is nonterminal key
; Returns: a list of production indices
; Invoked: build-nonterminal-associations

   (let ((position-list nil) (from 0))
      (dotimes (pos (count key nonterminal-list) position-list)
         (setq from (1+ (position key nonterminal-list :start from)))
 (setq position-list (append position-list (list from)))
      )
   )
)

;---------------------------------------------------------------------------;
```

```
(defun flatten (a-list)

; Purpose: places all sublists into main list
; Returns: a flattened list
; Invoked: remove-terminals

   (cond
      ((atom a-list)
         (list a-list)
      )
      (t
         (let ((flat-list))
            (dolist (this-list a-list flat-list)
               (setq flat-list (append flat-list (flatten this-list)))
            )
         )
      )
   )
)

;---------------------------------------------------------------------;

(defun modify-grammar (grammar-list)

; Purpose: modifies list of productions for easier computation
; Returns: grammar list


   (let ((grammar (mapcar 'cdr grammar-list)) (cross nil) (mutate nil))
      (dolist (this-subst (build-nonterminal-associations grammar-list))
         (let ((this-sym (gensym)))
            (setq grammar (subst this-sym (car this-subst) grammar))
            (setq <symbols> (cons this-sym <symbols>))
            (setq <lookup>
               (append <lookup>
     (list (append (list this-sym) (cadr this-subst)))
               )
            )
            (if (or (null <crossover>) (member (car this-subst) <crossover>))
               (setq cross (append cross (cadr this-subst)))
            )
            (if (or (null <mutation>) (member (car this-subst) <mutation>))
               (setq mutate (append mutate (cadr this-subst)))
            )
 )
      )
      (setq <crossover> (remove-terminals cross grammar))
      (setq <mutation> (remove-terminals mutate grammar))
      grammar
   )
)
```

```
;----------------------------------------------------------------------------;

(defun remove-terminals (production-list grammar-list)

; Purpose: remove terminal productions from list
; Returns: list with only nonterminals
; Invoked: modify-grammar

   (dotimes (index (length grammar-list) production-list)
      (if (not (intersection <symbols> (flatten (nth index grammar-list))))
         (setq production-list (remove (1+ index) production-list))
      )
   )
)

;****************************************************************************;
;*                                                                          *;
;*                               MUTATION                                   *;
;*                                                                          *;
;* ARGUMENTS:                                                               *;
;*    nil                                                                   *;
;*                                                                          *;
;* GLOBALS:                                                                 *;
;*    <mutation>         - list of productions where mutation can take place *;
;*    *mutation-rate     - probability of mutating each production in trace *;
;*    <new-population>   - the array of individuals after crossover happens *;
;*    *overlap*          - number of top individuals to keep each generation *;
;*    *population*       - the number of individuals in current population  *;
;*                                                                          *;
;* CALLS:                                                                   *;
;*    (create-phenotype) - builds new phenotype after mutating its genotype *;
;*    (parity-count)     - determines subtrace end for splice in crossover  *;
;*                                                                          *;
;* RETURNS:                                                                 *;
;*    nil                                                                   *;
;*                                                                          *;
;* EFFECTS:                                                                 *;
;*    <new-population>   - array of post-mutated individuals in population  *;
;*                                                                          *;
;****************************************************************************;

(defun mutation ()

; Purpose: controls mutation of genotypes in <new-population>
; Returns: nil

   (do
      ((index *overlap* (1+ index)))
      ((equal index *population*))
      (let ((mutation (perform-mutation
                        (individual-genotype (aref <new-population> index)))))
         (setf (individual-genotype (aref <new-population> index)) mutation)
         (setf (individual-phenotype (aref <new-population> index))
               (make-rule mutation)
         )
      )
   )
)

;----------------------------------------------------------------------------;

(defun perform-mutation (trace)

; Purpose: performs mutation on individual trace or subtrace
; Returns: mutated or original trace
; Invoked: mutation, perform-mutation

   (let ((production (car trace)))
      (cond
         ((null production)
            nil
         )
         ((and (member production <mutation>) (< (random 1.0) *mutation-rate*))
            (append
               (make-trace production)
               (perform-mutation (mutate-end trace (parity-count trace 0)))
            )
         )
         (t
            (cons production (perform-mutation (cdr trace)))
         )
      )
   )
)

;----------------------------------------------------------------------------;

(defun mutate-end (trace locus)

; Purpose: finds end of list being spliced
; Returns: the end sublist
; Invoked: perform-mutation

   (if (not (equal (length trace) locus))
      (subseq trace locus)
   )
)

;****************************************************************************;
;*                                                                          *;
;*                               PARITY-COUNT                               *;
;*                                                                          *;
;* ARGUMENTS:                                                               *;
;*     trace              - a rule trace or subtrace derived from the grammar *;
;*     locus              - the index of the first element in the subtrace    *;
;*                                                                          *;
;* GLOBALS:                                                                 *;
;*     nil                                                                  *;
;*                                                                          *;
;* RETURNS:                                                                 *;
;*     the index of last element of subtrace after parity count from locus   *;
;*                                                                          *;
;****************************************************************************;

(defun parity-count (trace locus)

; Purpose: conducts search for end of subtrace
; Returns: the index of the last element of the sublist

   (search-for-zero 1 (nthcdr (1+ locus) trace) (1+ locus))
)

;--------------------------------------------------------------------------;

(defun search-for-zero (parity subtrace index)

; Purpose: finds end of subtrace by doing parity count
; Returns: the index of the last element of the sublist
; Invoked: parity-count, search-for-zero

   (cond
      ((zerop parity)
         index
      )
      ((zerop (car subtrace))
         (search-for-zero (1- parity) (cdr subtrace) (1+ index))
      )
      (t
         (search-for-zero (1+ parity) (cdr subtrace) (1+ index))
      )
   )
)

;****************************************************************************;
;*                                                                          *;
;*                              PROCESS-GRAMMAR                             *;
;*                                                                          *;
;* ARGUMENTS:                                                               *;
;*     nil                                                                  *;
;*                                                                          *;
;* GLOBALS:                                                                 *;
;*     *grammar-file*     - name and path of the file containing grammar info *;
;*                                                                             *;
;* CALLS:                                                                      *;
;*     (modify-grammar)   - alters productions for more efficient computation *;
;*     (read-grammar)     - reads original grammar from *grammar-file*        *;
;*                                                                             *;
;* RETURNS:                                                                    *;
;*     nil                                                                     *;
;*                                                                             *;
;* EFFECTS:                                                                    *;
;*     <grammar>          - array of grammar productions for building rules   *;
;*                                                                             *;
;*******************************************************************************;

(defun process-grammar ()

; Purpose: stores productions in array and creates helpful lookup variables
; Returns: nil

   (let ((grammar-list (read-grammar)))
      (let ((modified-list (modify-grammar grammar-list)))
         (setq <grammar> (make-array (1+ (length modified-list))))
         (dotimes (pos (length modified-list))
               (setf (aref <grammar> (1+ pos)) (nth pos modified-list))
            )
         )
      )
   )
)

;*******************************************************************************;
;*                                                                             *;
;*                                READ-GRAMMAR                                 *;
;*                                                                             *;
;* ARGUMENTS:                                                                  *;
;*      nil                                                                    *;
;*                                                                             *;
;* GLOBALS:                                                                    *;
;*     *grammar-file*    - name and path of file containing the grammar       *;
;*                                                                             *;
;* RETURNS:                                                                    *;
;*      a list of all productions read from *grammar-file* in order           *;
;*                                                                             *;
;* EFFECTS:                                                                    *;
;*     <crossover>      - a list of nonterminals (if supplied) for crossover  *;
;*     <mutation>       - a list of nonterminals (if supplied) for mutation   *;
;*                                                                             *;
;*******************************************************************************;

(defun get-crossover-and-mutation (instream)

; Purpose: reads top lines of *grammar-file* for possible operator information
; Returns: the first grammar production, updates <crossover> and <mutation>
; Invoked: read-grammar

   (do
      ((rule (read instream nil nil) (read instream nil nil)))
      ((not (or (equal (car rule) 'crossover) (equal (car rule) 'mutation)))
          rule
      )
      (if (equal (car rule) 'crossover)
         (setq <crossover> (append <crossover> (cdr rule)))
         (setq <mutation> (append <mutation> (cdr rule)))
      )
   )
)

;-----------------------------------------------------------------------;

(defun read-grammar ()

; Purpose: reads in each line of *grammar-file* as an individual production
; Returns: a list of productions as they appear in *grammar-file*

   (let ((instream (open *grammar-file*)) (grammar-list nil))
      (do
         ((rule (get-crossover-and-mutation instream) (read instream nil nil)))
         ((null rule))
         (setq grammar-list (append grammar-list (list rule)))
      )
      (close instream)
      grammar-list
   )
)

;***********************************************************************;
;*                                                                     *;
;*                          READ-TRAINING                              *;
;*                                                                     *;
;* ARGUMENTS:                                                          *;
;*     nil                                                             *;
;*                                                                     *;
;* GLOBALS:                                                            *;
;*    *train-file*      - path and file name where training data is found  *;
;*    <train-list>      - list where training data is kept to be processed *;
;*                                                                     *;
;* RETURNS:                                                            *;
;*     nil                                                             *;
;*                                                                     *;
;* EFFECTS:                                                            *;
;*    <train-list>    - all training data in *train-file* stored as a list  *;
;*                                                                     *;
;***********************************************************************;
```

```
(defun read-training ()

; Purpose: reads in each line of *train-file* as training example
; Returns: nil

   (let ((instream (open *train-file*)))
      (do
         ((example (read instream nil nil) (read instream nil nil)))
         ((null example))
         (setq <train-list> (append <train-list> (list example)))
      )
      (close instream)
   )
)

;*******************************************************************************;
;*                                                                             *;
;*                                REPRODUCTION                                 *;
;*                                                                             *;
;* ARGUMENTS:                                                                  *;
;*    nil                                                                      *;
;*                                                                             *;
;* GLOBALS:                                                                    *;
;*    <old-population>   - sorted array of individuals from last generation  *;
;*    *overlap*          - number of top individuals to keep each generation *;
;*    *population*       - total number of individuals to reproduce in array *;
;*    <sum-of-fitness>   - total population fitness of <old-population>      *;
;*                                                                             *;
;* RETURNS:                                                                    *;
;*    nil                                                                      *;
;*                                                                             *;
;* EFFECTS:                                                                    *;
;*    <new-population>   - new array selected from <old-population> members  *;
;*                                                                             *;
;*******************************************************************************;

(defun reproduction ()

; Purpose: chooses new individuals for <new-population>
; Returns: nil

   (do
      ((index *overlap* (1+ index)))
      ((equal index *population*))
      (let ((replace (aref <new-population> index)) (selected (selection)))
         (setf (individual-genotype replace) (individual-genotype selected))
         (setf (individual-phenotype replace) (individual-phenotype selected))
         (setf (individual-fitness replace) (individual-fitness selected))
      )
   )
```

```
)

;-----------------------------------------------------------------------------;

(defun selection ()

; Purpose: does random roulette wheel selection from <old-population>
; Returns: selected population member
; Invoked: reproduction

   (let ((choice (random <sum-of-fitness>)) (partial 0.0))
      (do
         ((index 0 (1+ index)))
         ((or (>= partial choice) (= index *population*))
            (aref <old-population> (1- index)))
         (setq
            partial
            (+ partial (individual-fitness (aref <old-population> index)))
         )
      )
   )
)

;*****************************************************************************;
;*                                                                           *;
;*                               STATISTICS                                  *;
;*                                                                           *;
;* ARGUMENTS:                                                                *;
;*     generation          - number of successive generations already produced *;
;*                                                                           *;
;* GLOBALS:                                                                  *;
;*     *best-display*      - number of best population members to display    *;
;*     <old-population>    - an array of type individual sorted by fitness   *;
;*     <output-stream>     - where to direct statistical information output  *;
;*     *population*        - number of individuals to be found in population *;
;*     <sum-of-fitness>    - total of fitness function values in population  *;
;*                                                                           *;
;* RETURNS:                                                                  *;
;*     nil, displays all information for generation to *standard-output*    *;
;*                                                                           *;
;*****************************************************************************;

(defun print-individual (index)

; Purpose: prints fitness value and phenotype of population member
; Returns: nil, outputs to stream
; Invoked: print-population

  (format <output-stream>
          "fitness value:  ~S~%"
          (individual-fitness (aref <old-population> index))
  )
  (format <output-stream>
          "rule phenotype: ~S~%"
          (individual-phenotype (aref <old-population> index))
  )
)

;------------------------------------------------------------------------;

(defun print-population ()

; Purpose: prints out the *best-display* population members in generation
; Returns: nil, outputs to stream
; Invoked: print-report

   (format <output-stream> "Population Report:~%")
   (dotimes (index *best-display*)
      (format <output-stream>
              "~S~%"
              (individual-phenotype (aref <old-population> index))
      )
   )
)

;------------------------------------------------------------------------;

(defun print-report (generation)

; Purpose: prints out statistical information about population
; Returns: nil, prints population infomation to <output-stream>
; Invoked: statistics

   (format <output-stream>
           "------------------------------------------------------------~%"
   )
   (format <output-stream> "Report for generation ~S:~%~%" generation)
   (format <output-stream>
           "Maximum fitness value:      ~S~%"
           (* 1.0 (individual-fitness (aref <old-population> 0)))
   )
   (format <output-stream>
           "Average population fitness: ~S~%~%"
           (/ <sum-of-fitness> *population*)
   )
)

;------------------------------------------------------------------------;

(defun statistics (generation)

; Purpose: displays statistics for generation via <output-stream>
; Returns: nil

   (print-report generation)
   (if (> *best-display* 0) (print-population))
)

;*****************************************************************************;
```

Bibliography

[1]

[2]

[3] D.H. Ackley. A connectionist algorithm for genetic search. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 121–135. Lawrence Erlbaum Associates, 1985.

[4] D.H. Ackley. An empirical study of bit vector function optimization. In L. Davis, editor, *Genetic algorithms and simulated annealing*, chapter 13, pages 170–204. Morgan Kaufmann, 1987.

[5] David W. Aha. Relating relational learning algorithms. In Stephen Muggleton, editor, *Inductive Logic Programming*, chapter 11, pages 233–254. Turing Institute Press, 1992.

[6] Akiko N. Aizawa and Benjamin Wah. Dynamic control of genetic algorithms in a noisy environment. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 48–55. Morgan Kaufmann, 1993.

[7] H.J. Antonisse. A new interpretation of schema notation that overturns the binary encoding constraint. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 86–91. Morgan Kaufmann, 1989.

[8] H.J. Antonisse and K.S. Keller. Genetic operators for high-level knowledge representations. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 69–76. Lawrence Erlbaum Associates, 1987.

[9] R. Axelrod. The evolution of strategies in the iterated prisoner's dilemma problem. In L. Davis, editor, *Genetic algorithms and simulated annealing*, chapter 3, pages 32–41. Morgan Kaufmann, 1987.

[10] Thomas Bäck and Frank Hoffmeister. Extended selection mechanisms in genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 92–99. Morgan Kaufmann, 1991.

[11] Thomas Bäck, Frank Hoffmeister, and Hans-Paul Schwefel. A survey of evolution strategies. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 2–9. Morgan Kaufmann, 1991.

[12] Sugato. Bagchi, Serdar Uckun, Yutaka Miyabe, and Kazuhiko Kawamura. Exploring problem-specific recombination operators for job shop scheduling. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 10–17. Morgan Kaufmann, 1991.

[13] J.D. Bagley. *The behavior of adaptive systems which employ genetic and correlation algorithms.* PhD thesis, University of Michigan, 1967.

[14] James Edward Baker. Reducing bias and inefficiency in the deletion algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 14–21. Lawrence Erlbaum Associates, 1987.

[15] J.E. Baker. Adaptive selection methods for genetic algorithms. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 101–111. Lawrence Erlbaum Associates, 1985.

[16] N.A. Barricelli. Numerical testing of evolution theories. *ACTA Biotheoretica*, 16:69–126, 1962.

[17] David Beasley, David R. Bull, and Ralph R. Martin. Reducing epistasis in combinatorial problems by expansive coding. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 400–407. Morgan Kaufmann, 1993.

[18] R.K. Belew. When both individuals and populations search: adding simple learning to the genetic algorithm. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 34–41. Morgan Kaufmann, 1989.

[19] R.K. Belew and M. Gherrity. Back propagation for the classifier system. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 275–281. Morgan Kaufmann, 1989.

[20] Kristen Bennett, Michael C. Ferris, and Yannis E. Ioannidis. A genetic algorithm for database query optimization. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 400–407. Morgan Kaufmann, 1991.

[21] Hugues Bersini and Francisco J. Varela. The immune recruitment mechanism: a selective evolutionary strategy. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 520–526. Morgan Kaufmann, 1991.

[22] Bir Bhanu, Sungkee Lee, and John Ming. Self-optimizing image segmentation system using a genetic algorithm. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 362–370. Morgan Kaufmann, 1991.

[23] Jay N. Bhuyan, Vijay V. Raghavan, and Venkatesh K. Elayavalli. Genetic algorithm for clustering with an ordered representation. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 408–415. Morgan Kaufmann, 1991.

[24] Arthur S. Bickel and Rita Wenig Bickel. Tree structured eules in genetic algorithms. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 77–81. Lawrence Erlbaum Associates, 1987.

[25] Pierre Bonelli and Alexandre Parodi. An efficient classifier system and its experimental comparison with two representative learning methods on three medical domains. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 288–295. Morgan Kaufmann, 1991.

[26] L.B. Booker. *Intelligent behavior as an adaptation to the task environment.* PhD thesis, University of Michigan, 1982.

[27] L.B. Booker. Improving the performance of genetic algorithms in classifier systems. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 80–92. Lawrence Erlbaum Associates, 1985.

[28] L.B. Booker. Improving search in genetic algorithms. In L. Davis, editor, *Genetic algorithms and simulated annealing*, chapter 5, pages 61–73. Morgan Kaufmann, 1987.

[29] L.B. Booker. Classifier systems that learn internal world models. *Machine Learning*, 3:161–192, 1988.

[30] L.B. Booker. Triggered rule discovery in classifier systems. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 265–274. Morgan Kaufmann, 1989.

[31] Mark F. Bramlette. Initialization, mutation and selection methods in genetic algorithms for function optimization. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 100–107. Morgan Kaufmann, 1991.

[32] M.F. Bramlette and R. Cusic. A comparative evaluation of search methods applied to parametric design of aircraft. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 213–218. Morgan Kaufmann, 1989.

[33] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees.* Wadsworth International, 1984.

[34] Clayton L. Bridges and David E. Goldberg. An analysis of reproduction and crossover in a binary-coded genetic algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 9–13. Lawrence Erlbaum Associates, 1987.

[35] D.E. Brown, C.L. Huntley, and A.R. Spillane. A parallel genetic heuristic for the quadratic assignment problem. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 406–415. Morgan Kaufmann, 1989.

[36] Craig Caldwell and Victor S. Johnston. Tracking a criminal suspect through "face-space" with a genetic algorithm. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 416–421. Morgan Kaufmann, 1991.

[37] David L. Calloway. Using a genetic algorithm to design binary phase-only filters for pattern recognition. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 422–429. Morgan Kaufmann, 1991.

[38] Hugh M. Cartwright and Gregory F. Mott. Looking around: using clues from the data space to guide genetic algorithm searches. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 108–114. Morgan Kaufmann, 1991.

[39] T.P. Caudell and C.P. Dolan. Parametric connectivity: training of constrained networks using genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 370–374. Morgan Kaufmann, 1989.

[40] D.J. Cavicchio. *Adaptive search using simulated evolution.* PhD thesis, University of Michigan, 1970.

[41] Rong-Jaye Chen, Robert R. Meyer, and Jonathan Yeckel. A genetic algorithm for diversity minimization and its parallel implementation. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 163–170. Morgan Kaufmann, 1993.

[42] P. Chi. Genetic search with proportion estimations. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 92–97. Morgan Kaufmann, 1989.

[43] C.H. Chu. A genetic algorithm approach to the configuration of stack filters. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 219–224. Morgan Kaufmann, 1989.

[44] G.A. Cleveland and S.F. Smith. Using genetic algorithms to schedule flow shop releases. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 160–169. Morgan Kaufmann, 1989.

[45] J.P. Cohoon, S.U. Hegde, W.N. Martin, and D. Richards. Punctuated equilibria: a parallel genetic algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 148–154. Lawrence Erlbaum Associates, 1987.

[46] J.P. Cohoon, W.N. Martin, and D.S. Richards. A multi-population genetic algorithm for solving the k-partition problem on hyper-cubes. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 244–248. Morgan Kaufmann, 1991.

[47] Robert J. Collins and David R. Jefferson. Selection in massively parallel genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 249–256. Morgan Kaufmann, 1991.

[48] M. Compiani, D. Montanari, R. Serra, and P. Simonini. Asymptotic dynamics of classifier systems. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 298–303. Morgan Kaufmann, 1989.

[49] Susan Coombs and Lawrence Davis. Genetic algorithms and communication link speed design: constraints and operators. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 257–260. Lawrence Erlbaum Associates, 1987.

[50] N.L. Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 183–187. Lawrence Erlbaum Associates, 1985.

[51] Charles Darwin. *On the Origin of Species by Means of Natural Selection.* John Murray, 1859.

[52] Rajarshi Das and Darrell Whitley. The only challenging problems are deceptive: global search by solving order-1 hyperplanes. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 166–173. Morgan Kaufmann, 1991.

[53] Y. Davidor. Analogous crossover. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 98–103. Morgan Kaufmann, 1989.

[54] Yuval Davidor. A naturally occuring niche & species phenomenon: the model and first results. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 257–263. Morgan Kaufmann, 1991.

[55] L. Davis. Job shop scheduling with genetic algorithms. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 136–140. Lawrence Erlbaum Associates, 1985.

[56] L. Davis. Adapting operator probabilities in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 61–69. Morgan Kaufmann, 1989.

[57] L. Davis. Mapping neural networks into classifier systems. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 375–378. Morgan Kaufmann, 1989.

[58] L. Davis and M. Steenstrup. Genetic algorithms and simulated annealing: an overview. In L. Davis, editor, *Genetic algorithms and simulated annealing*, chapter 1, pages 1–11. Morgan Kaufmann, 1987.

[59] Lawrence Davis. Bit-climbing, representational bias and test suite design. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 18–23. Morgan Kaufmann, 1991.

[60] Lawrence Davis and Susan Coombs. Genetic algorithms and communication link speed design: theoretical considerations. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 252–256. Lawrence Erlbaum Associates, 1987.

[61] Thomas E. Davis and Jose C. Principe. A simulated annealing like convergence theory for the simple genetic algorithm. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 174–181. Morgan Kaufmann, 1991.

[62] K.A. De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, University of Michigan, 1975.

[63] K.A. De Jong. Genetic algorithms: a 10 year perspective. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 169–177. Lawrence Erlbaum Associates, 1985.

[64] K.A. De Jong and W.M. Spears. Using genetic algorithms to solve np-complete problems. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 124–132. Morgan Kaufmann, 1989.

[65] Kenneth De Jong. On using genetic algorithms to search program spaces. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 210–216. Lawrence Erlbaum Associates, 1987.

[66] M. de la Maza. A seagul visits the race track. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 208–212. Morgan Kaufmann, 1989.

[67] Pedro S. de Souza, , and Sarosh N. Talukdar. Genetic algorithms in asynchronous teams. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 392–397. Morgan Kaufmann, 1991.

[68] K. Deb and D.E. Goldberg. An investigation of niche and species formation in genetic function optimization. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 42–50. Morgan Kaufmann, 1989.

[69] Charles P. Dolan and Michael G. Dyer. Towards the evolution of symbols. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 123–131. Lawrence Erlbaum Associates, 1987.

[70] Marco Dorigo and Enrico Sirtori. Alecsys: A parallel laboratory for learning classifier systems. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 296–302. Morgan Kaufmann, 1991.

[71] A.C. Englander. Machine learning of visual recognition using genetic algorithms. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 197–201. Lawrence Erlbaum Associates, 1985.

[72] Larry J. Eshelman and J. David Schaffer. crossover's niche. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 9–14. Morgan Kaufmann, 1993.

[73] Larry J. Eshelman and J.David Schaffer. Preventing premature convergence in genetic algorithms by preventing incest. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 115–122. Morgan Kaufmann, 1991.

[74] L.J. Eshelman, R.A. Caruana, and J.D. Schaffer. Bias in the crossover landscape. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 10–19. Morgan Kaufmann, 1989.

[75] M.J. Fitzpatrick and J.J. Grefenstette. Genetic algorithms in noisy environments. *Machine Learning*, 3:101–120, 1988.

[76] T.C. Fogarty. Varying the probabilities of mutation in the genetic algorithm. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 104–109. Morgan Kaufmann, 1989.

[77] Carlos M. Fonseca and Peter I. Fleming. Genetic algorithms for multiobjective optimization: formulation, discussion and generalization. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 416–423. Morgan Kaufmann, 1993.

[78] S. Forrest. Implementing semantic network structures using the classifier system. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 24–44. Lawrence Erlbaum Associates, 1985.

[79] Stephanie Forrest and Melanie Mitchell. The performance of genetic algorithms on walsh polynomials: some anomalous results and their explanations. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 182–189. Morgan Kaufmann, 1991.

[80] M.P. Fourman. Compaction of symbolic layout using genetic algorithms. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 141–153. Lawrence Erlbaum Associates, 1985.

[81] D.R. Frantz. *Non-linearities in genetic adaptive search.* PhD thesis, University of Michigan, 1972.

[82] A.S. Fraser. Simulation of genetic systems. *Journal of Theoretical Biology*, 2:329–346, 1962.

[83] Corey Fujiki and John Dickinson. Using the genetic algorithm to generate lisp source code to solve the prisoner's dilemma. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 236–240. Lawrence Erlbaum Associates, 1987.

[84] Paula S. Gabbert, Donald E. Brown, Christopher L. Huntley, Bernard P. Markowicz, and David E. Sappington. A system for learning routes and schedules with genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 430–436. Morgan Kaufmann, 1991.

[85] D.E. Glover. Solving a complex keyboard configuration problem through generalized adaptive search. In L. Davis, editor, *Genetic algorithms and simulated annealing*, chapter 2, pages 12–31. Morgan Kaufmann, 1987.

[86] David E. Goldberg, Kalyanmoy Deb, Hillol Kargupta, and Georges Harik. Rapid, accurate optimization of difficult problems using fast messy genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 56–64. Morgan Kaufmann, 1993.

[87] David E. Goldberg, Kalyanmoy Deb, and Bradley Korb. Don't worry, be messy. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 24–30. Morgan Kaufmann, 1991.

[88] David E.. Goldberg and Jon Richardson. Genetic algorithms with sharing for multimodal function optimization. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 41–49. Lawrence Erlbaum Associates, 1987.

[89] David E. Goldberg and Philip Segrest. Finite markov chain analysis of genetic algorithms. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 1–8. Lawrence Erlbaum Associates, 1987.

[90] David E. Goldberg and Robert E. Smith. Nonstationary function optimization using genetic algorithms with dominance and diploidy. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 59–68. Lawrence Erlbaum Associates, 1987.

[91] D.E. Goldberg. *Computer-aided gas pipeline operation using genetic algorithms and machine learning.* PhD thesis, University of Michigan, 1983.

[92] D.E. Goldberg. Genetic algorithms and rule learning in dynamic system control. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 8–15. Lawrence Erlbaum Associates, 1985.

[93] D.E. Goldberg. Simple genetic algorithms and the minimal, deceptive problem. In L. Davis, editor, *Genetic algorithms and simulated annealing*, chapter 6, pages 74–88. Morgan Kaufmann, 1987.

[94] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.

[95] D.E. Goldberg. Sizing populations for serial and parallel genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 70–79. Morgan Kaufmann, 1989.

[96] D.E. Goldberg. Zen and the art of genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 80–85. Morgan Kaufmann, 1989.

[97] D.E. Goldberg. Probability matching, the magnitude of reinforcement, and classifier system bidding. *Machine Learning*, 5(4):407–425, 1990.

[98] V. Scott Gordon and Darrell Whitley. Serial and parallel genetic algorithms as function optimizers. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 177–183. Morgan Kaufmann, 1993.

[99] M. Gorges-Schleuter. Asparagos an asynchronous parallel genetic optimization strategy. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 422–427. Morgan Kaufmann, 1989.

[100] David Perry Greene and Stephen F. Smith. A genetic system for learning models of consumer choice. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 217–223. Lawrence Erlbaum Associates, 1987.

[101] J.J. Grefenstette. Incorporating problem specific information into genetic algorithms. In L. Davis, editor, *Genetic algorithms and simulated annealing*, chapter 4, pages 42–60. Morgan Kaufmann, 1987.

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

[103] J.J. Grefenstette. A system for learning control strategies with genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 183–190. Morgan Kaufmann, 1989.

[104] J.J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-16(1), January-February 1986.

[105] J.J. Grefenstette and J.E. Baker. How genetic algorithms work: a critical look at implicit parallelism. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 20–27. Morgan Kaufmann, 1989.

[106] J.J. Grefenstette and J.M. Fitzpatrick. Genetic search with approximate function evaluations. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 112–120. Lawrence Erlbaum Associates, 1985.

[107] J.J. Grefenstette, R. Gopal, B. Rosmaita, and D. Van Gucht. Genetic algorithms for the traveling salesman problem. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 160–168. Lawrence Erlbaum Associates, 1985.

[108] J.J. Grefenstette, C.L. Ramsey, and A.C. Schultz. Learning sequential decision rules using simulation models and competition. *Machine Learning*, 5(4):355–381, 1990.

[109] John J. Grefenstette. Multilevel credit assignment in a genetic learning system. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 202–209. Lawrence Erlbaum Associates, 1987.

[110] John J. Grefenstette. Lamarckian learning in multi-agent environments. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 303–310. Morgan Kaufmann, 1991.

[111] Frédéric Gruau. Genetic synthesis of modular neural networks. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 318–325. Morgan Kaufmann, 1993.

[112] Simon Handley. Automated learning of a detector for α-helices in protein sequences via genetic programming. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 271–278. Morgan Kaufmann, 1993.

[113] S.A. Harp, T. Samad, and A. Guha. Towards the genetic synthesis of neural networks. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 360–369. Morgan Kaufmann, 1989.

[114] William E. Hart and Richard K. Belew. Optimizing an arbitrary function is hard for the genetic algorithm. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 190–195. Morgan Kaufmann, 1991.

[115] J. Hesser, R. Maenner, and O. Stucky. Optimization of steiner trees using genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 231–236. Morgan Kaufmann, 1989.

[116] M.R. Hilliard, G.E. Liepins, M. Palmer, M. Morrow, and J. Richardson. A classifier-based system for discovering scheduling heuristics. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 231–235. Lawrence Erlbaum Associates, 1987.

[117] K.J. Hintz. Procedure learning using a variable-dimension solution space. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 237–242. Morgan Kaufmann, 1989.

[118] J.H. Holland. Outline for a logical theory of adaptive systems. *Journal of the Association for Computing Machinery*, 3:297–314, 1962.

[119] J.H. Holland. Processing and processors of schemata. In E.L. Jacks, editor, *Associate information processing*, pages 127–146. American Elsevier, 1971.

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

[121] J.H. Holland. Properties of the bucket brigade. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 1–7. Lawrence Erlbaum Associates, 1985.

[122] J.H. Holland. Escaping brittleness: the possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Genetic algorithms and simulated annealing*, volume 2. Morgan Kaufmann, 1986.

[123] J.H. Holland, K.J. Holyoak, R.E. Nisbett, and P.R. Thagard. *Induction.* The MIT Press, 1986.

[124] J.H. Holland, K.J. Holyoak, R.E. Nisbett, and P.R. Thagard. Classifier systems, q-morphisms, and induction. In L. Davis, editor, *Genetic algorithms and simulated annealing*, chapter 9, pages 116–128. Morgan Kaufmann, 1987.

[125] J.H. Holland and J.S. Reitman. Cognitive systems based on adaptive algorithms. In D.A. Waterman and F. Hayes-Roth, editors, *Pattern directed inference systems*, pages 313–329. Academic Press, 1978.

[126] John H. Holland. Genetic algorithms and classifier systems: foundations and future directions. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 82–89. Lawrence Erlbaum Associates, 1987.

[127] R.B. Hollstein. *Artificial genetic adaptation in computer control systems.* PhD thesis, University of Michigan, 1971.

[128] Hbdollah Homaifar, Xiaoyun Qi, and John Fost. Analysis and design of a general ga deceptive problem. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 196–203. Morgan Kaufmann, 1991.

[129] Andrew Horner and David E. Goldberg. Genetic algorithms and computer-assisted music composition. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 437–441. Morgan Kaufmann, 1991.

[130] htpp://www.ics.uci.edu/mlearn/MLRepository.html. University of california irvine machine learning repository, 1996.

[131] D. Huang. The context-array bucket brigade algorithm: an enhanced approach to credit-apportionment in classifier systems. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 311–316. Morgan Kaufmann, 1989.

[132] Martin Hulin. Analysis of schema distributions. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 204–209. Morgan Kaufmann, 1991.

[133] Philip Husbands and Frank Mill. Simulated co-evolution as the mechanism for emergent planning and scheduling. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 264–270. Morgan Kaufmann, 1991.

[134] M.A. Huynen and P. Hogeweg. Genetic algorithms and information accumulation during the evolution of gene regulation. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 225–230. Morgan Kaufmann, 1989.

[135] Hitoshi Iba, Takio Kurita, Hugo de Garis, and Taisuke Sato. System identification using structured genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 279–286. Morgan Kaufmann, 1993.

[136] Cezary Z. Janikow and Zbigniew Michalewicz. An experimental comparison of binary and floating point representations in genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 31–36. Morgan Kaufmann, 1991.

[137] P. Jog, J.Y. Suh, and D. Van Gucht. The effects of population size, heuristic crossover and local improvement on a genetic algorithm for the traveling salesman problem. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 110–115. Morgan Kaufmann, 1989.

[138] Prasanna Jog and Dirkj Van Gucht. Parallelisation of probabilistic sequential search algorithms. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 170–176. Lawrence Erlbaum Associates, 1987.

[139] Clayton M. Johnson and Stefan Feyock. A genetics-based approach to the automated acquisition of expert system rule bases. In *Proceedings of the IEEE/ACM International Conference on Developing and Managing Expert System Programs*. IEEE Computer Society Press, 1991.

[140] Donald R. Jones and Mark A. Beltramo. Solving partitioning problems with genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 442–449. Morgan Kaufmann, 1991.

[141] Kate Juliff. A multi-chromosome genetic algorithm for pallet loading. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 467–473. Morgan Kaufmann, 1993.

[142] Hillol Kargupta and R.E. Smith. System identification with evolving polynomial networks. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 370–376. Morgan Kaufmann, 1991.

[143] Charles L. Karr. Design of an adaptive fuzzy logic controller using a genetic algorithm. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 450–457. Morgan Kaufmann, 1991.

[144] James D. Kelly, Jr. and Lawrence Davis. Hybridizing the genetic algorithm and the k nearest neighbors classification algorithm. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 377–383. Morgan Kaufmann, 1991.

[145] Kenneth E. Kinnear, Jr. Generality and difficulty in genetic programming: evolving a sort. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 287–294. Morgan Kaufmann, 1993.

[146] Hiroaki Kitano, Stephen F. Smith, and Tetsuya Higuchi. Ga-1: a parallel associative memory processor for rule learning with genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 311–317. Morgan Kaufmann, 1991.

[147] D. Klahr and R.S. Siegler. The representation of children's knowledge. In H.W. Reese and L.P. Lipsitt, editors, *Advances in child development and behavior*, pages 61–116. Academic Press, 1978.

[148] Corey Kosak, Joe Marks, and Stuart Shieber. A parallel genetic algorithm for network-diagram layout. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 458–465. Morgan Kaufmann, 1991.

[149] John R. Koza. Evolving a computer program to generate random numbers using the genetic programming paradigm. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 37–44. Morgan Kaufmann, 1991.

[150] John R. Koza. Simultaneous discovery of reusable detectors and subroutines using genetic programming. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 295–302. Morgan Kaufmann, 1993.

[151] J.R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 768–774. Morgan Kaufmann, 1989.

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

[153] P. Langley. A general theory of discrimination learning. In D. Klahr, P. Langley, and R. Neches, editors, *Production system models for learning and development*, pages 99–161. MIT Press, 1987.

[154] In Lee, Riyaz Sikora, and Michael J. Shaw. Joint lot sizing and sequencing with genetic algorithms for scheduling: evolving the chromosome structure. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 383–390. Morgan Kaufmann, 1993.

[155] Michael A. Lee and Hideyuki Takagi. Dymanic control of genetic algorithms using fuzzy logic techniques. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 76–83. Morgan Kaufmann, 1993.

[156] James R. Levenick. Inserting introns improves genetic algorithm success rate: taking a cue from biology. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 123–127. Morgan Kaufmann, 1991.

[157] David M. Levine. A genetic algorithm for the set partitioning problem. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 481–487. Morgan Kaufmann, 1993.

[158] G.E. Liepins, M.R. Hilliard, Mark Palmer, and Michael Morrow. Greedy genetics. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 90–99. Lawrence Erlbaum Associates, 1987.

[159] Gunar E. Liepins and Lori A. Wang. Classifier system learning of boolean concepts. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 318–323. Morgan Kaufmann, 1991.

[160] Marc Lipsitch. Adaptation on rugged landscapes generated by iterated local interactions on neighboring genes. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 128–135. Morgan Kaufmann, 1991.

[161] M.L. Littman and D.H. Ackley. Adaptation in constant utility non-stationary environments. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 136–142. Morgan Kaufmann, 1991.

[162] Sushil J. Louis and Gregory J.E. Rawlins. Designer genetic algorithms: genetic algorithms in structure design. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 53–60. Morgan Kaufmann, 1991.

[163] C.B. Lucasius and G. Kateman. Application of genetic algorithms in chemometrics. In *Proceedings of the Third International Conference on Genetic Algorithms.* pages 170–176. Morgan Kaufmann, 1989.

[164] B. Manderick and P. Spiessens. Fine-grained parallel genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 428–433. Morgan Kaufmann, 1989.

[165] Bernard Manderick, Mark de Weger, and Piet Spiessens. The genetic algorithm and the structure of the fitness landscape. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 143–150. Morgan Kaufmann, 1991.

[166] Nashat Mansour and Geoffrey C. Fox. A hybrid genetic algorithm for task allocation in multicomputers. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 466–473. Morgan Kaufmann, 1991.

[167] R.E. Marks. Breeding hybrid strategies: optimal behavior for oligopolists. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 198–207. Morgan Kaufmann, 1989.

[168] Andrew J. Mason. Partition coefficients, static deception and deceptive problems for non-binary alphabets. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 210–214. Morgan Kaufmann, 1991.

[169] Zbigniew Michalewicz and Cezary Z. Janikow. Handling constraints in genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 151–157. Morgan Kaufmann, 1991.

[170] G.E. Miller, P.M. Todd, and S.U. Hegde. Designing neural networks using genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 379–384. Morgan Kaufmann, 1989.

[171] J.H. Miller and S. Forrest. The dynamic behavior of classifier systems. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 304–310. Morgan Kaufmann, 1989.

[172] D.J. Montana. Empirical learning using rule threshold optimization for detection of events in synthetic images. *Machine Learning*, 5(4):427–450, 1990.

[173] H. Mühlenbein. Parallel genetic algorithms, population genetics and combinatorial optimization. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 416–421. Morgan Kaufmann, 1989.

[174] H. Mühlenbein, M. Schomisch, and J. Born. The parallel genetic algorithm as function optimizer. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 271–278. Morgan Kaufmann, 1991.

[175] Ryohei Nakano and Takeshi Yamada. Conventional genetic algorithms for job shop problems. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 474–479. Morgan Kaufmann, 1991.

[176] A. Newell. *Unified Theories of Cognition.* Harvard University Press, 1990.

[177] Jean-Pierre Nordvik and Jean-Michel Renders. Genetic algorithms and their potential for use in process control: a case study. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 480–486. Morgan Kaufmann, 1991.

[178] M.O. Odetayo and D.R. McGregor. Genetic algorithm for inducing control rules for a dynamic system. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 177–182. Morgan Kaufmann, 1989.

[179] I.M. Oliver, D.J. Smith, and J.R.C. Holland. A study of permutation crossover operators on the traveling salesman problem. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 224–230. Lawrence Erlbaum Associates, 1987.

[180] Jim Oliver. Discovering individual decision rules: an application of genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 216–222. Morgan Kaufmann, 1993.

[181] Deon G. Oosthuizen. Supergran: a connectionist approach to learning, integrating genetic algorithms and graph induction. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 132–139. Lawrence Erlbaum Associates, 1987.

[182] D.G. Oosthuizen. Machine learning: a mathematical framework for neural network, symbolic and genetics-based learning. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 385–390. Morgan Kaufmann, 1989.

[183] C.B. Pettey and M.R. Leuze. A theoretical investigation of a parallel genetic algorithm. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 398–405. Morgan Kaufmann, 1989.

[184] Chrisila .B. Pettey, Michael R. Leuze, and John J. Grefenstette. A parallel genetic algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 155–161. Lawrence Erlbaum Associates, 1987.

[185] D.J. Powell, S.S. Tong, and M.M. Skolnick. Engeneous domain independent, machine learning for design optimization. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 151–159. Morgan Kaufmann, 1989.

[186] Xiaofeng Qi and Francesco Palmieri. The diversification role of crossover in the genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 132–137. Morgan Kaufmann, 1993.

[187] Yuri Rabinovich and Avi Wigderson. An analysis of a simple genetic algorithm. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 215–221. Morgan Kaufmann, 1991.

[188] Nicholas J. Radcliffe. Forma analysis and random respectful recombination. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 222–229. Morgan Kaufmann, 1991.

[189] Vijay V. Raghavan and Brijesh Agarwal. Optimal determination. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 241–246. Lawrence Erlbaum Associates, 1987.

[190] Connie Loggia Ramsey and John J. Grefenstette. Case-based initialization of genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 84–91. Morgan Kaufmann, 1993.

[191] Thomas S. Ray. Is it alive or is it ga? In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 527–534. Morgan Kaufmann, 1991.

[192] Colin R. Reeves. Using genetic algorithms with small populations. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 92–99. Morgan Kaufmann, 1993.

[193] L. Rendell. Genetic plans and the probabilistic learning system: synthesis and results. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 60–73. Lawrence Erlbaum Associates, 1985.

[194] J.T. Richardson, M.R. Palmer, G.E. Liepins, and M. Hilliard. Some guidelines for genetic algorithms with penalty functions. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 191–197. Morgan Kaufmann, 1989.

[195] Rick L. Riolo. Bucket brigade performance: I. long sequences of classifiers. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 184–195. lawrence erlbaum associates, 1987.

[196] Rick L. Riolo. Bucket brigade performance: Ii. default hierarchies. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 196–201. lawrence erlbaum associates, 1987.

[197] Rick L. Riolo. Modeling simple human category learning with a classifier system. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 324–333. Morgan Kaufmann, 1991.

[198] R.L. Riolo. The emergence of coupled sequences of classifiers. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 256–264. Morgan Kaufmann, 1989.

[199] R.L. Riolo. The emergence of default hierarchies in learning classifier systems. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 322–327. Morgan Kaufmann, 1989.

[200] G. Roberts. A rational reconstruction of wilson's animat and holland's cs-1. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 317–321. Morgan Kaufmann, 1989.

[201] George G. Robertson. Parallel implementation of genetic algorithms in a classifier system. In L. Davis, editor, *Genetic algorithms and simulated annealing*, chapter 10, pages 129–140. Morgan Kaufmann, 1987.

[202] George G. Robertson. Parallel implementation of genetic algorithms in a classifier system. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 140–147. Lawrence Erlbaum Associates, 1987.

[203] David Rogers. G/splines: a hybrid of friedman's multivariate adaptive regression splines (mars) algorithm with holland's genetic algorithm. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 384–391. Morgan Kaufmann, 1991.

[204] Steve G. Romaniuk. Evolutionary growth perceptrons. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 334–341. Morgan Kaufmann, 1993.

[205] H. Ros. Some results on boolean concept learning by genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 28–33. Morgan Kaufmann, 1989.

[206] R.S. Rosenberg. *Simulation of genetic populations with biochemical properties*. PhD thesis, University of Michigan, 1967.

[207] Gerhard Roth and Martin D. Levine. A genetic algorithm for primitive extraction. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 487–494. Morgan Kaufmann, 1991.

[208] A.L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3(3):210–229, 1959.

[209] Adrian V. Sannier II and Erik D. Goodman. Genetic learning procedures in distributed environments. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 162–169. Lawrence Erlbaum Associates, 1987.

[210] J. David Schaffer and Larry J. Eshelman. On crossover as an evolutionarily viable strategy. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 61–68. Morgan Kaufmann, 1991.

[211] J.D. Schaffer. Learning multiple pattern discrimination. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 74–79. Lawrence Erlbaum Associates, 1985.

[212] J.D. Schaffer. Multiple objective optimization with vector evaluated genetic algorithms. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 93–100. Lawrence Erlbaum Associates, 1985.

[213] J.D. Schaffer. Some effects of selection procedures on hyperplane sampling by genetic algorithms. In L. Davis, editor, *Genetic algorithms and simulated annealing*, chapter 7, pages 89–103. Morgan Kaufmann, 1987.

[214] J.David Schaffer, R.A. Caruana, L.J. Eshelman, and R. Das. A study of control parameters affecting online performance of genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 51–60. Morgan Kaufmann, 1989.

[215] J.David Schaffer and Amy Morishima. An adaptive crossover distribution mechanism for genetic algorithms. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 36–40. Lawrence Erlbaum Associates, 1987.

[216] Alan C. Schultz. Adapting the evaluation space to improve global learning. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 158–164. Morgan Kaufmann, 1991.

[217] T. Schultz, D. Mareschal, and W. Schmidt. Modeling cognitive development on balance scale phenomena. *Machine Learning*, 16:59–88, 1994.

[218] D. Schuurmans and J. Schaeffer. Representational difficulties with classifier systems. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 328–333. Morgan Kaufmann, 1989.

[219] Tod A. Sedbrook, Haviland Wright, and Richard Wright. Application of a genetic classifier for patient triage. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 334–338. Morgan Kaufmann, 1991.

[220] B. Selman and G. Hirst. Parsing as an energy minimization problem. In L. Davis, editor, *Genetic algorithms and simulated annealing*, chapter 11, pages 141–154. Morgan Kaufmann, 1987.

[221] C.G. Shaefer. Directed tree method for fitting a potential function. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 208–226. Lawrence Erlbaum Associates, 1985.

[222] Craig G. Shaefer. The argot strategy: adaptive representation genetic optimizer technique. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 50–58. Lawrence Erlbaum Associates, 1987.

[223] R. Shonkwiler. Parallel genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 199–205. Morgan Kaufmann, 1993.

[224] R. Shonkwiler, F. Mendivil, and A. Deliu. Genetic algorithms for the 1-d fractal inverse problem. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 495–501. Morgan Kaufmann, 1991.

[225] L. Shu and J. Schaeffer. Vcs: variable classifier system. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 334–339. Morgan Kaufmann, 1989.

[226] Lingyan Shu and Jonathan Schaeffer. Hcs: adding hierarchies to classifier systems. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 339–345. Morgan Kaufmann, 1991.

[227] W. Siedlecki and J. Sklansky. Constrained genetic optimization via dynamic reward-penalty balancing and its use in pattern recognition. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 141–150. Morgan Kaufmann, 1989.

[228] R.S. Siegler. Three aspects of cognitive development. *Cognitive Psychology*, 8:481–520, 1976.

[229] David J. Sirag and Paul T. Weisser. Toward a unified thermodynamic genetic operator. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 116–122. Lawrence Erlbaum Associates, 1987.

[230] Alice Smith and David M. Tate. Genetic optimization using a penalty function. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 499–505. Morgan Kaufmann, 1993.

[231] D. Smith. Bin packing with adaptive search. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 202–206. Lawrence Erlbaum Associates, 1985.

[232] Joshua R. Smith. Designing biomorphs with an interactive genetic algorithm. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 535–538. Morgan Kaufmann, 1991.

[233] R.E. Smith and M. Valenzuela-Rendón. A study of rule set development in a learning classifier system. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 340–346. Morgan Kaufmann, 1989.

[234] S.F. Smith. *A learning system based on genetic adaptive algorithms.* PhD thesis, University of Pittsburgh, 1980.

[235] S.J. Smith and S.W. Wilson. Rosetta: toward a model of learning problems. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 347–350. Morgan Kaufmann, 1989.

[236] William M. Spears and Kenneth A. De Jong. On the virtues of parameterized uniform crossover. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 230–236. Morgan Kaufmann, 1991.

[237] Piet Spiessens and Bernard Manderick. A massively parallel genetic algorithm: implementation and first analysis. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 279–286. Morgan Kaufmann, 1991.

[238] Irene Stadnyk. Schema recombination in a pattern recognition problem. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 27–35. Lawrence Erlbaum Associates, 1987.

[239] T. Starkweather, S. McDaniel, K. Mathias, D. Whitley, and C. Whitley. A comparison of genetic sequencing operators. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 69–76. Morgan Kaufmann, 1991.

[240] Guy L. Steele, Jr. *Common Lisp: The Language.* Digital Press, 2 edition. 1990.

[241] Jung Y. Suh and Dirk Van Gucht. Incorporating heuristic information into genetic search. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 100–107. Lawrence Erlbaum Associates, 1987.

[242] Keiji Suzuki and Yukinori Kakazu. An approach to the analysis of the basins of the associative memory model using genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 539–546. Morgan Kaufmann, 1991.

[243] G. Syswerda. Uniform crossover in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2–9. Morgan Kaufmann, 1989.

[244] Gilbert Syswerda and Jeff Palmucci. The application of genetic algorithms to resource scheduling. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 502–508. Morgan Kaufmann, 1991.

[245] M. Tan and L. Eschelman. Using weighted networks to represent classification knowledge in noisy domains. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 121–134. Morgan Kaufmann, 1988.

[246] R. Tanese. Distributed genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 434–439. Morgan Kaufmann, 1989.

[247] Reiko Tanese. Parallel genetic algorithm for a hypercube. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 177–183. Lawrence Erlbaum Associates, 1987.

[248] David M. Tate and Alice E. Smith. Expected allele coverage and the role of mutation in genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 31–37. Morgan Kaufmann, 1993.

[249] Philip Thrift. Fuzzy logic synthesis with genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 509–513. Morgan Kaufmann, 1991.

[250] Peter M. Todd and Geoffrey F. Miller. On the sympatric origin of species: mercurial mating in the quicksilver model. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 547–554. Morgan Kaufmann, 1991.

[251] Jan Torreele. Temporal processing with recurrent networks: an evolutionary approach. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 555–561. Morgan Kaufmann, 1991.

[252] D.S. Touretzky and G.E. Hinton. Pattern matching and variable binding in a stochastic neural network. In L. Davis, editor, *Genetic algorithms and simulated annealing*, chapter 12, pages 155–169. Morgan Kaufmann, 1987.

[253] M. Valenzuela-Rendón. Boolean analysis of classifier sets. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 351–358. Morgan Kaufmann, 1989.

[254] Manuel Valenzuela-Rendón. The fuzzy classifier system: a classifier system for continuously varying variables. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 346–353. Morgan Kaufmann, 1991.

[255] Gregor von Laszewski. Intelligent structural operators for the k-way graph partitioning problem. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 45–52. Morgan Kaufmann, 1991.

[256] Michael D. Vose and Gunar E. Liepins. Schema disruption. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 237–242. Morgan Kaufmann, 1991.

[257] D.A. Waterman. Generalization learning techniques for automating the learning of heuristics. *Artificial Intelligence*, 1:121–170, 1970.

[258] T.H. Westerdale. The bucket brigade is not genetic. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 45–59. Lawrence Erlbaum Associates, 1985.

[259] T.H. Westerdale. A defense of the bucket brigade. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 282–290. Morgan Kaufmann, 1989.

[260] T.H. Westerdale. Redundant classifiers and prokaryote genomes. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 354–360. Morgan Kaufmann, 1991.

[261] Thomas H. Westerdale. Altruism in the bucket brigade. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 22–26. Lawrence Erlbaum Associates, 1987.

[262] D. Whitley. The genitor algorithm and selection pressure: why rank-based allocation of reproductive trials is best. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121. Morgan Kaufmann, 1989.

[263] D. Whitley and T. Hanson. Optimizing neural networks using faster, more accurate genetic search. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 391–396. Morgan Kaufmann, 1989.

[264] D. Whitley, K. Mathias, and P. Fitzhorn. Delta coding: an iterative search stategy for genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 77–84. Morgan Kaufmann, 1991.

[265] D. Whitley, T. Starkweather, and D. Fuquay. Scheduling problems and traveling salesmen: the genetic edge recombination operator. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 133–140. Morgan Kaufmann, 1989.

[266] Darrell Whitley. Using reproductive evaluation to improve genetic search and heuristic discovery. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 108–115. Lawrence Erlbaum Associates, 1987.

[267] Darrell Whitley, Stephen Dominic, and Rajarshi Das. Genetic reinforcement learning with multilayer neural networks. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 562–569. Morgan Kaufmann, 1991.

[268] Stewart W. Wilson. The genetic algorithm and biological development. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 247–251. Lawrence Erlbaum Associates, 1987.

[269] Stewart W. Wilson. Ga-easy does not imply steepest-ascent optimizable. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 85–89. Morgan Kaufmann, 1991.

[270] S.W. Wilson. Adaptive "cortical" pattern recognition. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 188–193. Lawrence Erlbaum Associates, 1985.

[271] S.W. Wilson. Knowledge growth in an artificial animal. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 16–23. Lawrence Erlbaum Associates, 1985.

[272] S.W. Wilson. Classifier systems and the animat problem. *Machine Learning*, 2:199–228, 1987.

[273] S.W. Wilson. Hierarchical credit allocation in a classifier system. In L. Davis, editor, *Genetic algorithms and simulated annealing*, chapter 8, pages 104–115. Morgan Kaufmann, 1987.

[274] S.W. Wilson and D.E. Goldberg. A critical review of classifier systems. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 244–255. Morgan Kaufmann, 1989.

[275] Peter Wyard. Context free grammar induction using genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 514–518. Morgan Kaufmann, 1991.

[276] H.H. Zhou. *CSM: a genetic classifier system with memory for learning by analogy*. PhD thesis, Vanderbilt University, 1987.

[277] H.H. Zhou. Csm: a computational model of learning. *Machine Learning*, 5(4):383–406, 1990.

[278] H.H. Zhou and J.J. Grefenstette. Learning by analogy in genetic classifier systems. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 291–297. Morgan Kaufmann, 1989.

[279] Raed Abu Zitar and Mohammad H. Hassoun. Regulator control via genetic search assisted reinforcement. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 254–262. Morgan Kaufmann, 1993.

VITA

Clayton Matthew Johnson was born in Peoria, Illinois, on December 22, 1964, and graduated from Covington Latin High School in Covington, Kentucky, in 1982. He earned a B.S. in Computer Science and a B.A. in International Affairs from Xavier University in Cincinnati, Ohio, completing many of the requirements for these degrees at the Universit"*at* Salzburg in Salzburg, Austria. The author received an M.S. in Computer Science from Michigan State University in East Lansing, Michigan, in 1988, before entering the College of William and Mary as a research assistant in the department of Computer Science. In addition to work in government and industry, he has held the rank of assistant professor at Indianola College in Indianola, Iowa, and at Virginia Commonwealth University in Richmond, Virginia.