RECOMBINATION, SELECTION, AND THE GENETIC CONSTRUCTION OF COMPUTER PROGRAMS

by

Walter Alden Tackett

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Computer Engineering)

April 1994

Copyright © 1994 Walter Alden Tackett

RECOMBINATION, SELECTION, AND THE GENETIC CONSTRUCTION OF COMPUTER PROGRAMS[1]

by

Walter Alden Tackett

_____

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Computer Engineering)

April 1994

Copyright © 1994 Walter Alden Tackett

[1]This research was sponsored in part by the University of Southern California Zumberge Research and Initiation fund.

Dedication

To my parents and siblings.

Acknowledgments

I know quite a few people who started in pursuit of a Doctorate and didn't make it. From where I am now, looking back I see many pitfalls and stumbling blocks that I averted. I did not avoid them by myself, but rather by the grace of others who helped me along my way. Among these people, one of the most prominent is my Advisor, Committee Chair, and *Friend*, Professor Jean-Luc Gaudiot. Many in his position insist that their students perform research which extends the Advisor's agenda with little regard for the interests of the victim -er, I mean *student*. I came to Jean-Luc's group with the intent of studying parallel processing and meandered from that point across the intellectual landscape, pausing at many local optima before settling on the work you hold before you. Along the way I received nothing but encouragement, punctuated by the occasional query, "Well, are you ready to write your outline *now*?" So far as parallel processing goes, I have devoted a full two paragraphs at the end of Chapter 2.

I would also like to thank Paul Rosenbloom and Keith Price, committee members who met my eccentric taste in dissertation topics with great aplomb, as did Len Adleman, Viktor Prasanna, and Dan Moldovan ("the carjacker's worst nightmare"), who served upon my qualifying committee.

Tremendous support was provided along the way by my friends in the technical community. John Koza started this branch in the family tree of Evolutionary Computation, and James Rice also played an instrumental role in the origins of Genetic Programming (GP). Together the two of them have supported my research in every possible way from day one. Aviram Carmi learned about Genetic Programming the hard way, that is to say by working with me. He was a big help in the construction of the GP/C system (known to the GP community in a simpler incarnation as SGPC). He was also instrumental to the research that eventually led to Chapter 4, and some results from his Master's Thesis, included by kind permission, make a nice addendum to the end of that chapter. Lee Altenberg has been a tremendous influence, most evident here in the ideas concerning brood selection, which formed the basis of Chapters 5 and 6. Paul Rosenbloom brought "an AI perspective" which has permanently changed my thinking about the evolutionary method. He suggested the search analyses which begin in Chapter 2 and recur throughout the text. Kim Kinnear and Jerry Burman have provided a great deal of input

and discussions on a variety of subjects. Together with Lee, I have been able to count on them often as an impromptu review board for various and sundry write-ups.

Tom Ray is responsible for getting me started in the Artificial Life field, after I happened to encounter an article describing Tom's work with Tierra in *Science News*. This was at a time when I had decided that my work (with neural networks) was rapidly proceeding in a direction that I didn't like. I had read about A-Life before, but Tom's work had a great freshness and appeal which made me want to get involved. When I called him he provided tools, encouragement, and friendship. I became certain from that time on that I had a subject matter worthy of investigation.

I was probably the second person, after James Rice, to get paid to do Genetic Programming. I would like to issue thanks to the many individuals who supported those activities at the Hughes Aircraft Company (in chronological order): Jim Alves, Tony Baraghimian, Chuck McNary, Knut Kongelbeck, Kenneth Friedenthal, Greg Shelton, and Al Gebbie. I would particularly like to thank John Bartelt, whose interest is keen enough to make him an active participant in technology development, and whose advice has been most valuable in matters both technical and non-technical.

I was also the beneficiary of generous donations of hardware and software by my parents, Walter and Dorothy Tackett, and by Sid Maxwell. This rendered me "computationally independent." The Northridge Quake would have set me back much longer than it did if I did not have my code, data, and documents backed up and ready-to-run in my home.

Not all the assistance I got was technical. I owe a tremendous debt to my friends Tom James, Paul Green, and Bill Wilson. They have provided unflagging support and lots of good advice. I thank them for convincing me that the best way to be successful in my work is to make a well-balanced life my first priority.

And finally, thanks to Jenny for everything else.

"To those we have omitted in our haste, God bless them as well."

Walter Alden Tackett
Canoga Park, California
April 17th, 1994

Contents

List of Tables

List of Figures

Abstract

Computational intelligence seeks, as a basic goal, to create artificial systems which mimic aspects of biological adaptation, behavior, perception, and reasoning. Toward that goal, genetic program induction–"Genetic Programming"–has succeeded in automating an activity traditionally considered to be the realm of creative human endeavor. It has been applied successfully to the creation of computer programs which solve a diverse set of model problems. This naturally leads to questions such as:

- Why does it work?
- How does it fundamentally differ from existing methods?
- What can it do that existing methods cannot?

The research described here seeks to answer those questions. This is done on several fronts. Analysis is performed which shows that Genetic Programming has a great deal in common with heuristic search, long studied in the field of Artificial Intelligence. It introduces a novel aspect to that method in the form of the *recombination operator* which generates successors by combining parts of favorable strategies. On another track, we show that Genetic Programming is a powerful tool which is suitable for real-world problems. This is done first by applying it to an extremely difficult induction problem and measuring performance against other state-of-the-art methods. We continue by formulating a model induction problem which not only captures the pathologies of the real world, but also parameterizes them so that variation in performance can be measured as a function of confounding factors. At the same time, we study how the effects of search can be varied through the effects of the *selection operator*. Combining the lessons of the search analysis with known properties of biological systems leads to the formulation of a new recombination operator which is shown to improve induction performance. In support of the analysis of selection and recombination, we define problems in which structure is precisely controlled. These allow fine discrimination of search performance which help to validate analytic predictions. Finally, we address a truly unique aspect of Genetic Programming, namely the exploitation of symbolic procedural knowledge in order to provide "explanations" from genetic programs.

Chapter 1

Introduction and Contributions of the Research

"O.K Stimpy. Now it is time for our evolving *lesson!"*

Ren Höek

1.1 Introduction

Adaptation is a primary property of living systems which may take place at many levels. In even the simplest living things, adaptation acts on a small time scale through the reaction of an organism to its immediate conditions. For more sophisticated animals, adaptation may also act on an intermediate time scale in which an organism learns to respond to conditions based on repeated experiences. On a very large time scale acting over many lifetimes, the very forms of organisms are altered by the forces of evolutionary adaptation. Adaptation has likewise been applied on many fronts to artificial systems, ranging from simple thermostatic controls for regulation of climate in buildings to sophisticated "Computationally Intelligent" systems applied to automated reasoning, pattern recognition, and induction.

The creation of computer programs is an area of human endeavor in which a great deal of strategy, re-use of successful solutions, and trial-and error are involved. One approach to the adaptive construction of computer programs which embodies the re-use of solutions, as well as a great deal of trial-and-error, is the "Genetic Programming" method introduced in (Koza 1992). It applies the principles of evolutionary adaptation, borrowed directly from Holland's Genetic Algorithm (Holland 1975; 1992), to the construction of procedural code. The landmark work presented in (Koza 1992) has provided an important longitudinal study of a large number of problems to which Genetic Programming can be successfully applied, and the spread of interest in the subject has been exponential. The 1991 International Conference on Genetic Algorithms (ICGA) contained a single paper concerning GP, by Koza. Six months after the publication of Koza's book, the 1993 ICGA proceedings contained six papers on GP, including (Tackett 1993a) which eventually became Chapter 3 of this volume. At the time of this writing, a

workshop proceedings dedicated entirely to Genetic Programming is in press (Kinnear 1994a). That proceedings contains some 20 or so papers, including (Tackett and Carmi 1994a) which is presented in a much revised form as Chapter 4. Finally, the 1994 IEEE World Conference on Computational Intelligence in Orlando has provided a separate Genetic Programming Track, also containing some 25 papers, among them (Tackett 1994b), which comprises portions of Chapter 5.

Whereas (Koza 1992) has provided a longitudinal study of GP applications, the work presented here seeks to provide a longitudinal study of the mechanisms underlying genetic program induction. These mechanisms can be divided roughly into two categories, namely *selection* and *recombination*. Chapters 4, 5, and 6 explore the properties of a variety of selection and recombination methods, advancing the performance capabilities of the GP method.

Another goal of this work is to demonstrate an in-depth study of some difficult applications which are taken from the "real world" and/or which model in a parametric way the characteristics that make the real world a hard thing to deal with. Critics of Koza's original work have postulated that the successes there were due to judicious choice of problems so that prior knowledge of solutions could be used to tailor the approach. The work described in Chapters 3 and 4 goes a long way towards addressing those criticisms. The problems there are explicitly tailored so as to have no perfect solution, nor even an optimal solution which can be constructed from the given elements. It is important to note that recent work described in (Koza 1994) also has adopted some very difficult problems, different from those presented here, which address some of the early criticisms.

Finally, no great consideration has been given in GP literature to the exploitation of the symbolic representation which the method employs. A goal of this work is to initiate research in the area of "trait mining," which is the extraction of salient problem elements and relationships from the population of genetically induced programs. Algorithms and techniques presented in Chapter 7 provide foundations for that undertaking.

1.2 Contributions of the Research

The contributions of the research are listed here in the order that they appear.

- *Characterization of genetic program induction as beam search*. Genetic program induction is most frequently described in terms of biology and quasi-biological properties.

Analysis in Chapters 2, 4, and 5 provides an understanding in terms of AI search. This opens the way for the Machine Learning community at large to gain an understanding of GP in their own terms, opening avenues for interchange. Likewise, the understanding of GP in terms of an existing and well-understood technology allows explanations of its power and also helps to point out areas for improvement. For example, the power of the greedy recombination method introduced in Chapter 5 is easily understood when considered in terms of search. Credit goes to Paul Rosenbloom of USC Information Sciences Institute for suggesting this analysis.

- *Application to a difficult real-world induction problem.* Chapter 3 introduces a problem which uses large amounts of real data from a domain chosen specifically because of its difficulty. Since there is no solution, only comparative performance with other state-of-the art induction technologies can be used to gauge performance. The fact that GP methods perform favorably in a fair comparison says to the practicing scientist and engineer that genetic program induction is not just a clever toy, but rather a serious method worthy of investigation.
- *A new induction problem which models real-world pathologies*. The "Donut Problem" introduced in Chapter 4 is a contribution in and of itself. It is scaleable in several measures of difficulty, and is biased against being separated by any of the classical discriminants, including hyper- planes, spheres, rectangles, or cones.
- *Longitudinal studies of selection and generalization*. This is the first published work to demonstrate the comparative properties of spatially distributed and steady-state selection of populations in the context of Genetic Programming. Among the literature of Genetic Algorithms it is the first such study which is primarily concerned with generalization, which concerns performance against data not used in training the system. Typical problems previously studied focus on optimization tasks such as graph partitioning.
- *Demonstration of generalization and noise tolerance of GP methods*. Chapter 4 shows that Genetic Programming produces solutions which are within a constant margin of the Bayesian limit for the Donut Problem, and that GP solutions consistently converge to those performance levels. In tests of generalization ability, the training sample set can be reduced to a very small size while still retaining reasonable performance. These results were not

anticipated in the experimental design, which was intended to show the point at which GP performance "breaks down." More importantly, in recent work (Carmi 1994) repeats the Donut Problem experiments using the Multilayer Perceptron. Despite two-hundred-plus hours of labor dedicated to finding the optimal number of hidden layers and nodes, those networks demonstrate inconsistent convergence and poor generalization in comparison to the results obtained through genetic program induction.

- *The "Greedy Recombination" genetic operator*. Chapters 5 and 6 introduce the greedy recombination operator, providing theory and experiments which demonstrate improved properties relative to standard GP recombination. It is important to give proper credit to Lee Altenberg (Altenberg 1993) for originally suggesting the use of "Soft Brood Selection." The unique and original contribution added to that concept by this work is the use of a reduced-cost fitness evaluation for members of the brood. [2]
- *Constructional problems*. Important theoretical tools are carried over from the literature of GA to the literature of GP. The effectiveness of constructional problems in demonstrating exploratory and exploitative properties of genetic operators is shown.
- *The Gene Banker's Algorithm*. In order to analyze the traits, or subexpressions, occurring in the population they must first be recorded. This algorithm can place all unique subexpressions in the GP population into a hash table using time and space which are linearly proportional to N*S, where N is the size of the population and S is the average number of nodes in the parse-tree representation of a population member. Naive methods require time and space proportional to the square of N*S.
- *The trait-mining approach*. We introduce a method by which salient elements of the solution to a problem can be extracted from the larger and more complex expression in which they are contained. This is the first work to address the problem of how knowledge can be extracted from the symbolic representation employed by the GP method.
- *Demonstration of hitchhiking*. The hitchhiking phenomenon known to exist in binary-string Genetic Algorithms is shown to also exist in genetically induced programs.

[2] Well, actually it was Mother Nature's idea, but we in the adaptive computation field are in the business of plagiarizing her work.

Furthermore, evidence is provided that it is a prevalent phenomenon of the open-ended development of genome structure.

1.3 Outline of the Dissertation

The remainder of this dissertation is organized as follows:

Chapter 2) Artificial Genetic Methods of Adaptation

A background is provided in the field of Genetic Algorithms and Genetic Programming. A new analysis is performed which shows that Genetic Programming is closely related to the "beam search" methods of AI (Lowerre & Reddy 1980) applied to the space of computer programs. A comparison is made with alternative methods of adaptive programming including Tierra and FOIL. We conclude with a discussion of some detailed issues in Genetic Algorithms and the relation of genetic methods to some other strategies for adaptation.

Chapter 3) Genetic Programming for Image Discrimination

Genetic Programming is investigated through comparative performance against other methods of machine learning on a difficult induction problem in which features derived from infrared image data are classified. The GP method is used to perform discrimination using "statistical" features derived from an external source, and in a second experiment is allowed to formulate its own features from primitive intensity measures. Training is performed using about 2000 sample feature-vectors, and performance figures are reported using a separate out-of-sample set consisting of 7000 feature-vectors. The same experiments are repeated using a Multilayer Perceptron/Backpropagation network and a variant of Quinlan's ID3 decision tree, demonstrating that GP possesses a number of favorable properties. This is a large scale real-world problem, and probably represents the most significant demonstration of GP performance on real induction problems to date.

Chapter 4) The Effects of Selection Operators in the Context of a New Induction Problem

A variety of selection operators, including steady-state, distributed ("demes"), and panmictic selection are compared. The beam-search analysis of Chapter 2 is extended to show how these selection operators vary the search characteristics of Genetic Programming. In order to compare selection operators a scaleable and pathological induction problem is devised, and the optimal Bayesian solution for the problem is formulated as a basis for comparison. This "Donut

Problem" requires discriminating between samples drawn from two distributions, each a toroidal cloud of points, which are interlocked like links in a chain. To simulate real-world conditions, the Genetic Programming system is provided with a limited function set that cannot express the Bayesian solution, and an imperfect fitness function which can lead to over-fitting of data is used. Noise in the sample distributions is parametrically varied, creating ambiguities such that the Bayesian solution will achieve a non-zero error rate. The ability of Genetic Programming to generalize solutions is examined by varying the number of training samples provided while measuring performance against a fixed out-of-sample validation set. This is currently the only major longitudinal study of selection operators in Genetic Programming. Among the general GA literature it is one of the few studies of selection which deals with out-of-sample induction performance rather than optimization performance.

Chapter 5) An Introduction to the Greedy Recombination Operator

Many natural organisms overproduce zygotes and subsequently decimate the ranks of offspring at some later stage of development. This "brood selection" is done in order to reduce parental resource investment in inferior offspring. We introduce the recombination operator $R_B(n)$ which is parameterized by the brood size "n" and the brood culling function F_B. The computational "investment" of CPU and memory resources are characterized in terms of brood size, brood fitness evaluation cost (cost of F_B), and the fitness evaluation cost for full-fledged population members. The beam-search analysis of Chapter 2 is extended to show that $R_B(n)$ is performing a greedy search of potential recombination sites, as opposed to the random search of recombination sites performed in standard Genetic Programming. Subsequent tests of $R_B(n)$ using the induction problem of Chapter 4 demonstrate that by using smaller population sizes with large broods, equivalent or improved performance can be achieved using $R_B(n)$ while reducing the CPU and memory requirements relative to Genetic Programming with "standard" (random) recombination. Evaluation of the properties of $R_B(n)$ is continued in Chapter 6.

Chapter 6) Constructional Problems

A new class of problems is introduced to Genetic Programming in which fitness is based strictly upon the syntactic form of expressions rather than semantic evaluation: a certain target expression is assigned perfect fitness while those subexpressions resulting from its hierarchical decomposition comprise intermediate fitness values. This allows precise control over the

structure of search space thereby providing a mechanism with which to test search properties of operators.

Two problems are constructed, analogous to the "Royal Road" (Forrest & Mitchell 1993) and "deceptive" (Goldberg 1989) problems previously applied to binary-string Genetic Algorithms. Greedy and random recombination methods are tested in combination with several selection methods. The algorithmic search properties of these operators previously predicted in Chapters 2, 4, and 5 are clearly demonstrated through the analysis of results.

Chapter 7) Gene Banking and Trait Mining

A criticism of connectionist learning by the symbolic AI community is that neural methods are opaque with respect to providing insight into what they have learned about a problem. Machine learning has successfully produced alternative systems which can learn parsimonious symbolic rules of induction through hill climbing (Quinlan 1990). Other work has shown how such learning may be readily integrated with preexisting expert knowledge (Pazzani and Kibler, 1990). Genetic Programming is likewise a symbolic method of induction, and so has potential to feed symbolic knowledge about what it has learned back into the user environment. Among the potential advantages are that the genetic search may be more powerful than other methods applied in symbolic learning to date. An observable challenge is that genetically induced programs do not yield readily to inspection for many problems. The "Gene Banker's Algorithm" is introduced, which hashes all expressions and subexpressions (traits) occurring in the population in time linearly proportional to the number of functions and terminals in the population. A variety of statistics including conditional ("schema") fitness of each trait are computed and tracked over time. After a run is completed, the collection of traits can be mined in order to try and determine which traits and relationships are salient. For the purposes of this simple experiment traits are primarily extracted by sorting on conditional fitness and on frequency of occurrence. It is demonstrated that for simple problems such as those of Chapter 6 the extraction of salient expressions is readily achievable, although for many problems such as that of Chapter 4, interpretation of salient expressions may still be problematic. "Hitchhiking," which describes the artificial inflation of fitness estimates for useless expressions which embed themselves among salient expressions, is shown to be a primary confounding factor in this analysis.

Chapter 8) Conclusions and Directions for Future Research

The final chapter provides a perspective on the results obtained here, and upon the currents of Genetic Programming research. We then go on to propose a variety of suggestions for follow-on research, many of which concern ways of strengthening and extending the trait mining approach.

Chapter 2

Artificial Genetic Methods of Adaptation

Background and Comparison to Other Methods of Learning, Search, and Induction

A background is provided in the field of Genetic Algorithms and Genetic Programming, which are compared to other methods of adaptation such as neural networks and simulated annealing. Genetic Programming is then characterized in detail as a search on the space of computer programs. A comparison is made with other methods which search the space of computer programs.

2.1. Genetic Methods

Evolution as a method of Machine Learning has been around in many incarnations since the 1960's, including Evolutionary Programming, Evolution Strategies, and Holland's Genetic Algorithm. An excellent history and comparison of these approaches is available in (Back and Schwefel 1993). We first discuss the basic premises of the Genetic Algorithm (GA) and then proceed to outline the modifications which lead to Genetic Programming (GP) (Koza 1992), which forms the basis for this research.

2.1.1. Introduction: The Genetic Algorithm

The Genetic Algorithm was developed by John Henry Holland of the University of Michigan beginning in the early 1960's. In the early 1970's, Holland formulated his "Schema Theorem," which provided a mathematical basis for understanding the importance of genetic recombination to evolution and adaptation. This in turn formed the basis for his landmark book, *Adaptation in Natural and Artificial Systems* (Holland 1975).

The key elements of GA center around the selective breeding of a population of individuals (Figure 2.1). Fitness proportionate selection, which embodies the concept of "survival of the fittest," is used to select parents from the population. Genetic recombination ("crossover") is

applied to pairs of parents to create offspring which will be inserted into a new population, forming the next generation of individuals. We now examine this process in detail.

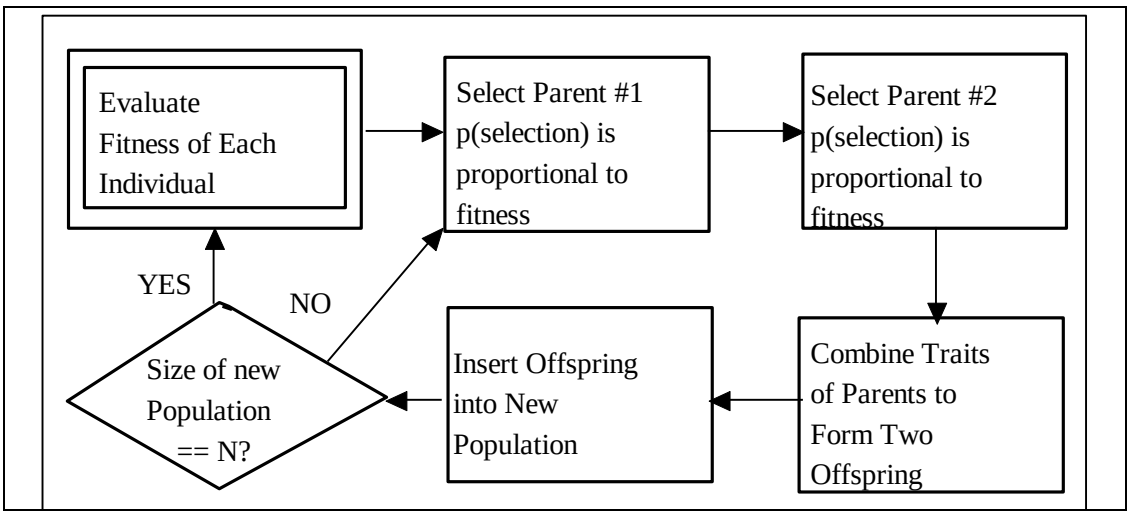


Figure 2.1: Starting from a random initial population, each individual is evaluated for fitness. Parents are selected randomly in proportion to their fitness and combined to form offspring which are inserted into the population of the next generation. When the new population is complete, the members are evaluated and the process begins again.

0 0 0 1 0 1 1 1	1 0 1 1 1 1 0 0
1 1 0 0 0 1 0 1	0 1 0 0 0 1 0 0
1 0 0 1 1 0 0 0	0 1 1 1 0 1 0 1
0 0 1 0 1 1 1 0	1 0 1 1 0 0 1 1

Figure 2.2: A population of N=8 binary strings each with a "length" of eight bits.

2.1.1.1. Population

A key to the Genetic Algorithm is the maintenance of a population of individuals. This population is generally of a fixed size *N* which does not change over time. As in populations of living organisms it is unusual to find two individuals which are exactly alike. Diversity within the population is very important, in fact, for reasons that will become clear in the next section. In "classical" GA the members of the population are represented as fixed-length strings of binary

digits as shown in Figure 2.2. The length of the strings and the population size N are completely dependent on the problem. Either may range from a few tens to many thousands (Collins 1992). In genetic terms, we say that each binary string represents a *genotype*. The genotype is the basic blueprint - or more properly, the recipe (Dawkins 1987) - for construction the individual, and it is what will be manipulated by the genetic process.

2.1.1.2. Fitness of the Individual

The genotype is decoded to form the *phenotype* of the individual: this is the incarnation which will be tested to determine the fitness of the genotype to reproduce. The phenotype is totally problem dependent: decoding of the genotype may represent control settings in a steam plant (Goldberg 1983), strategies in a 2-player game (Miller 1989), the transition table of a finite state machine (Collins 1992), or any of a variety of systems. Once the phenotype has been defined, we must apply a method of testing individuals to determine their fitness. In a power plant we might measure the thermal efficiency resulting from the valve settings. In a game strategy we might use a relative fitness measure, where each individual is rated based on the number of games it wins when playing against the other members of the population. As a simple example we formulate a problem in which the eight bit strings of Figure 2.2 are decoded to form integer phenotypes. This is illustrated in Figure 2.3.

23 (19)	188 (146)
197 (155)	68 (26)
152 (110)	117 (75)
46 (4)	179 (137)

Figure 2.3: Eight binary genotypes decoded to form their integer phenotypes (fitness values shown in parentheses).

The phenotype must be evaluated in context of the fitness measure. Using an example from (Reynolds 1992), we base the fitness of the phenotypes on how close they come to the answer

for "life the universe and everything" which is known to be 42 (Adams 1982). The fitness evaluation procedure could then be an equation of the form:

$$fitness = ABS(phenotype - 42).$$

Here, "greater fitness" would imply values closer to 0. Note that larger-is-better vs. smaller-is-better fitness values are dependent on the problem and, to a large degree, on personal taste. For example, although greater-is-better fitness is more intuitive, there are many problems in which fitness is most naturally expressed in terms of error minimization. Throughout Chapters 3-7 both conventions will be used.

2.1.1.3. Selection

"Survival of the fittest" is a familiar concept known to drive evolution. What is meant by *survival* in a qualitative and quantitative sense? The answer is that a *genotype survives* across generations through the production of like offspring, and the production of offspring is coupled to the fitness of the corresponding phenotype. In nature this coupling may be achieved by the ability of an individual to beat up its competitors of the same species in a contest for mating rights, or it may be achieved through the ability of the of the individual to run and/or hide from predators of other species in order to live long enough to mate. Most often, survival is a combination of many factors, with only one in common across all species and environments: random chance.

In Genetic Algorithms the fitness criteria are far more singular and well defined than in nature, but the stochastic element is preserved intact. In the classical GA method a "roulette wheel" is formed which has one slot for each member of the population. Individuals with higher fitness are allocated a wider slot and are more likely to be chosen by a spin of the wheel. The first step for generating a new population of size N is to perform N spins of the wheel, drawing parents from the population (with replacement). Thus individuals with higher fitness are selected to participate in reproduction more frequently. One drawback with this method is that the appropriate mapping from fitness values to "slot size" is unclear. This is further compounded in cases of smaller-is-better fitness criteria, where the fitness measure is inversely proportional to slot size. A more popular method in recent years is *K-tournament* selection, which is widely used by researchers and was originally proposed by (Wetzel 1979). In K-tournament selection,

K individuals are drawn from the population with replacement. The most fit individual among these is chosen as the "winner" of the tournament and becomes one parent for the next generation. The process is repeated N times. In (Brindle 1981) tournament and a variety of other methods were compared with roulette selection and generally found to be superior. The K-tournament is one example of rank-selection methods, for which some analytic justification is provided in (Whitley 1989). GP investigations pursuant to the work reported in Chapter 3 and in (Tackett 1993a) confirm this.

2.1.1.4. Trait Inheritance and Recombination

Once parents have been selected from the population their genetic material is combined to form offspring. Holland's Schema Theorem (Holland 1975) states that the power of genetic search lies in the continual recombination of genetic traits, or *schemata*, in parallel. Holland shows that through recombination a population of N individuals searches on the order of N^3 traits simultaneously. This property of genetic search is also demonstrated elegantly in (Goldberg 1989) and (Koza 1992).

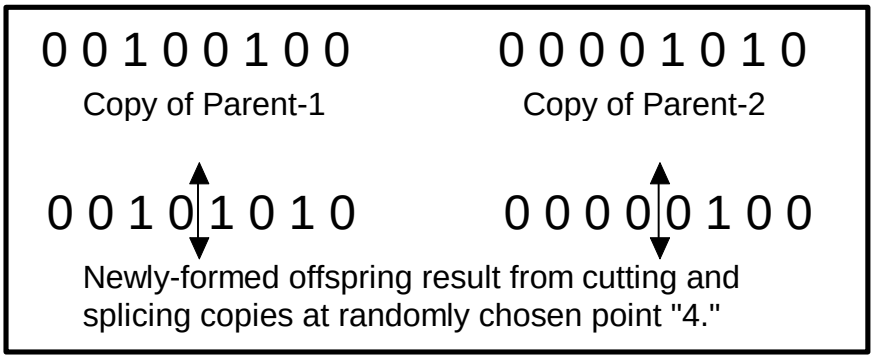


Figure 2.4: Recombination of genotypic traits in the Genetic Algorithm.

Schemata are substrings contained within the genotype whose particular values contribute to the overall fitness of the phenotype. Examples of schemata would be the strings "0 0 1 * * * * *", "* * 0 1 0 * 1 *", and "1 0 0 0 * * * *," where the asterisks (*) indicate "don't care" conditions. Note that schemata do not imply a particular length of substring nor that the fixed elements of the substring be contiguous within the genotype. Different schemata have varying impact upon the fitness of the genotypes which contain them: in our "42" problem, strings containing the schema "1 0 0 0 * * * *" would necessarily be more unfit, with an error of at least 86, than those

containing the schema "0 0 1 * * * * *", which could have an error of at most 21. Because of this, genotypes containing the latter schema are more likely to reproduce. Over time we expect the "0 0 1 * * * * *" to become predominant in the population.

In GA, recombination is a simple matter of splicing segments copied from parents at a random point. This is shown in Figure 2.4. In the first stage of the reproductive process, two parents which were selected from the population are copied. Next, the "crossover point" is selected at random: for genotypes of length **L**, the crossover point is chosen as a uniformly distributed random integer c such that $1 \leq c \leq \mathbf{L}\text{-}1$. If we consider the bits of the genotype to be numbered from 0 to **L**-1, then c will indicate that the genotypes will be cut at the point between bit (c-1) and bit c. The bits 0 to (c-1) of the first parent will be spliced to bits c through (**L**-1) of the second, and vice versa, producing two new offspring of length **L**. Most variants of the genetic algorithm allow some mechanism for copies of parents to occasionally be reproduced verbatim ("cloning"), either as a special case allowing for c = 0 or as an event whose likelihood is under control of separate parameters.

2.1.1.5. Mutation

Mutation is the means by which fundamentally new traits are introduced to the population. It is possible that for a given population, no members contain a desirable bit value in a particular place. For example, consider the case where the schema "* * 1 * * * * *" does not occur in any member of the population. There is no way that the recombination operation described in the previous section can rectify this. Mutation occurs randomly and very rarely both in natural and artificial genetic systems, but when it does it may cause genes to take on new values which have never occurred in the population before. Typically in GA, only one in many thousands of genotypes are affected by mutation. When an individual is selected to undergo mutation, one bit $0 \leq b \leq \mathbf{L}\text{-}1$ of the genotype is chosen and set to its complementary value. In a population with a high genetic diversity mutation becomes relatively unimportant and as a matter of fact is much more likely to do harm than good, since any complex system is likely to be degraded rather than enhanced by random changes.

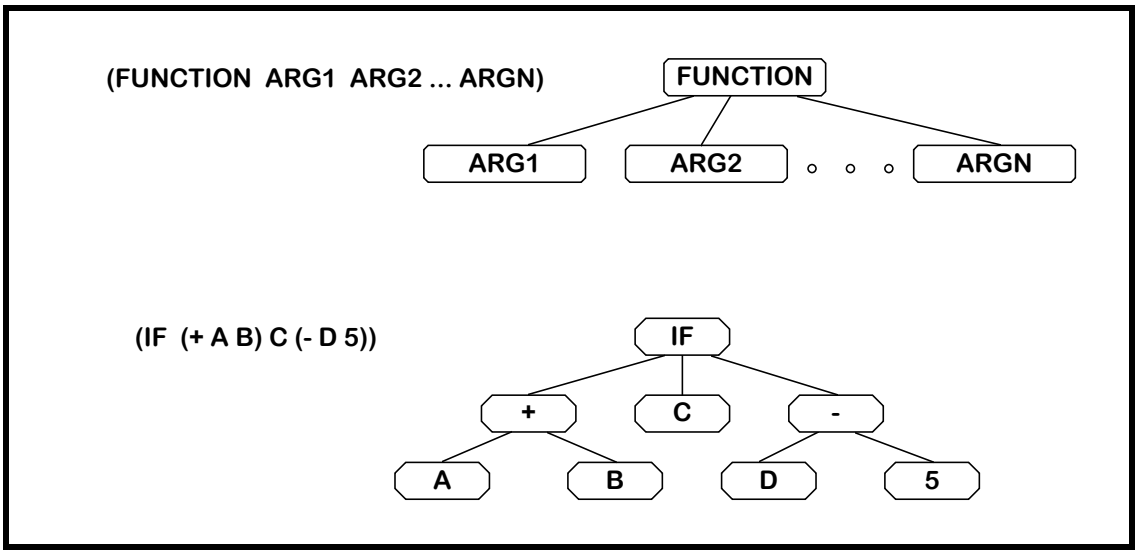


Figure 2.5: Some LISP programs and their parse tree representations.

2.1.2. The Genetic Programming: A New Representation

2.1.2.1. GP vs. GA

The primary difference between Genetic Programming and "classical" Genetic Algorithms is the representation of the structures they manipulate. Classical GA operate on a population of fixed-length binary string genotypes which typically (but not necessarily) represent numeric parameters. Genetic Programming by contrast manipulates a population of genotypes represented as "trees," which in turn may be decoded into *computer programs*. These programs may have an arbitrary size and structure. The definition of phenotype in GP then becomes more abstract than in GA, since it is the behavior, or *semantics*, of the computer programs which are of interest. In particular, these programs are expressed in the computer language LISP, as shown in Figure 2.5, and their genotypes represent their LISP *parse trees* (Pagan 1981). There is in theory no restriction on the language which may be expressed: FORTRAN, C, and all other computer languages may be represented by parse trees as well. In practice, the simplicity and uniformity of LISP syntax makes it a choice of convenience. In LISP, all operations are implemented as function calls. The syntax of the function call consists of a list of elements enclosed by parentheses. The first element within the list is the name of the function which is in turn the root of the corresponding parse tree. The rest of the elements within the list are arguments to the function, which are the children, or branches, at the next level down within the parse tree. The functional arguments may be variables and constants, or

they themselves may be function calls. In the latter case the same rules apply recursively, with sub-functions forming trees at greater depths.

2.1.2.2. Function and Terminal Sets

Trees consist of *functions* (internal nodes) and *terminals* (leaf nodes). The set of functions and terminals is chosen by the user according to the problem to be solved. For example, we might want programs which classify samples consisting of a set of features {A, B, C, D}. These variables would form the terminal set for the population of expression trees. Likewise, we might choose the four math operations and a conditional branch {+, -, x, /, IF} as the function set with which to combine the input variables. There is no limit to the complexity of the functions used, however. Iteration, "side-effect" functions such as assignment, and a wide variety of problem-specific functions are demonstrated in (Koza 1992) and (Koza 1994). A population of trees can be constructed using the designated functions and terminals via a random, recursive generation process as shown in Figure 2.6.

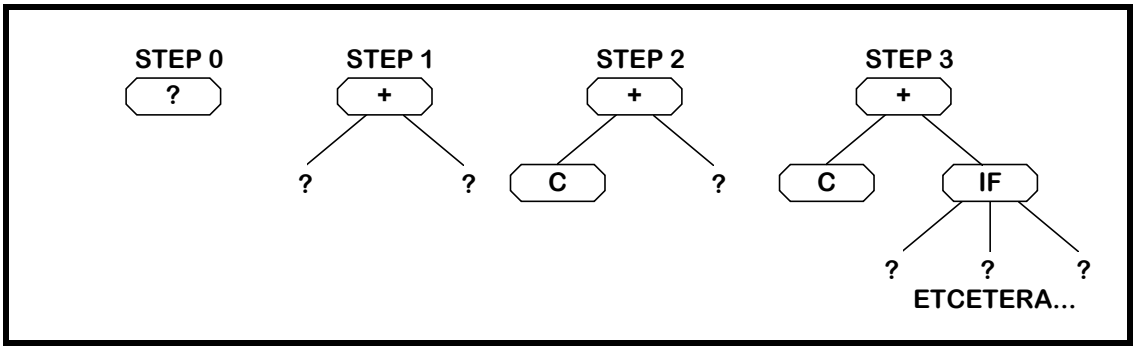


Figure 2.6: Starting with a function node, the parse tree is grown recursively by choosing functions and terminals at random.

2.1.2.3. Recombination of Genetic Traits in Parse Trees

Swapping of subtrees replaces the string splicing of conventional GA as the method for combining parental traits (Figure 2.7). In contrast to GA two crossover points are chosen, one within each parent. This is necessary since parent trees and corresponding programs are of a different "size" and "shape" from each other, unlike the GA strings which are of a uniform length. Likewise, the subtrees which will be cut from the crossover points will in general differ from each other in their size and shape. Recombination then consists of exchanging subtrees between copies of the original parents, resulting in offspring which are generally different in size

and shape from their parents and from each other. By imposing the constraint that all functions return the same type, syntactic correctness under these operations is guaranteed.

There is, of course, no guarantee as to the semantic correctness of the generated programs. Chapter 5 will discuss methods that reduce the randomness of recombination, resulting in improved performance and efficiency.

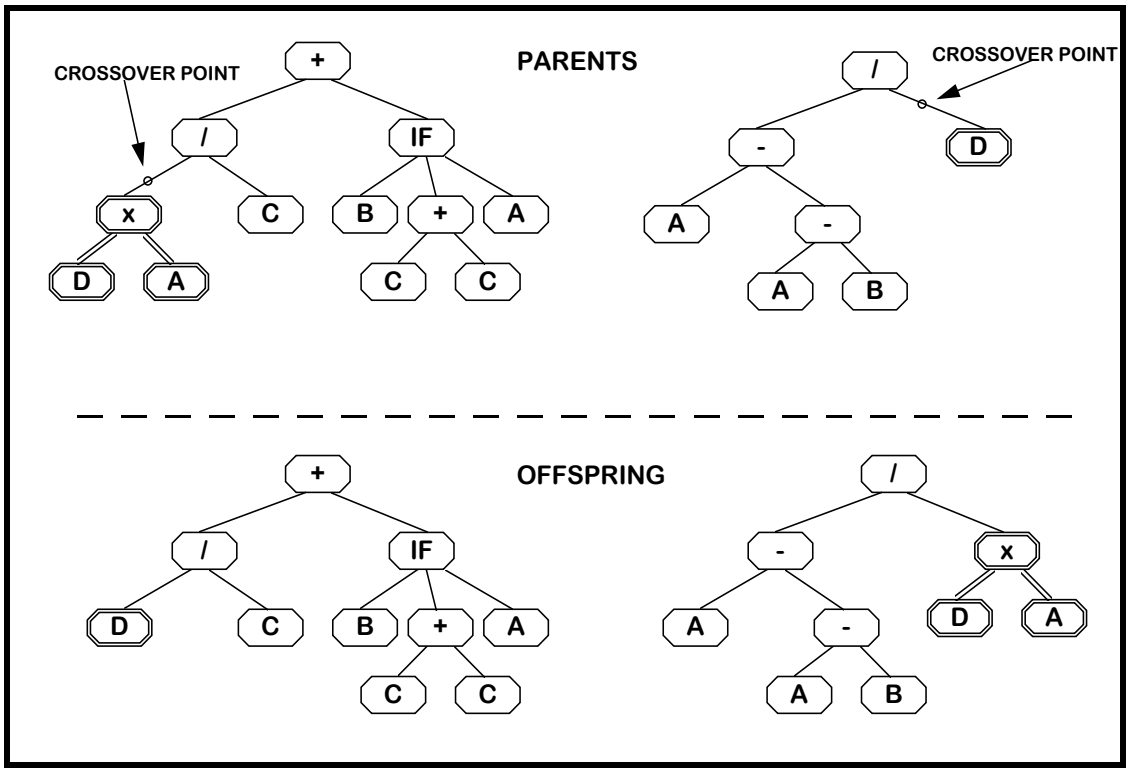


Figure 2.7: The recombination operator creates offspring which combine traits of parents by swapping subtrees at randomly chosen "crossover points."

2.1.2.4. Mutation in Parse Trees

Mutation is considered to be of relatively low importance to Genetic Programming. Mechanically, the operation is similar to crossover, except that a single parent tree is chosen, as is a single "mutation point" which is just a leaf or subtree of the parent. The subtree at the mutation point is replaced by a new, randomly generated subtree. Koza (Koza 1992) has empirically demonstrated that mutation is unnecessary for evolution to proceed in GP: with no mutation whatsoever a run with a sufficiently diverse initial population will converge to fit solutions as quickly as an equal sized population employing the mutation operation. If we consider the mitigating case for mutation presented in the previous section, we may see an

intuitive reason why this is so: before, we considered a fixed location on a genotype of fixed length, stating that if this locus contained the same value for all individuals in the population, then there was no way change could occur in the absence of mutation. This is not a danger in GP: since programs of new sizes, shapes, and configurations are being generated continuously, there is no concept of a fixed locus in the genotypes of the population. The only possible analogy to "losing" values would be if a member of the function set or terminal set were to completely disappear from the entire population. But a "reasonable" sized population might consist of hundreds, or perhaps thousands, of individuals each of which contains tens or hundreds of nodes. At a minimum, tens of thousands of nodes exist in the population, each of which must be one of several (say 10-20) unique functions or variables (terminals). If 20 functions and variables were initially distributed uniformly among 10,000 nodes within the population of trees, this would say that there were about 500 of each type. For all of any one type to go completely extinct is unlikely. The only situation where this would be likely to occur is one where the presence of a particular function or terminal is detrimental to the individuals which contain it.

2.2. Genetic Program Induction as Heuristic Search

Because Genetic Programming operates on symbolic information, which is the traditional domain of classical Artificial Intelligence (AI), it is useful to provide an understanding of GP in the terms the rich body of research associated with AI. This benefits the AI researcher because it demystifies some concepts of genetic methods which are cloaked in biological jargon. It is also of benefit to the practitioner of GP, since it provides a new perspective on the mechanisms at work.

2.2.1. Search on State Space Trees

Search is one of the primary tools used in the field of Artificial Intelligence. One classical model of search involves the traversal of a state-space tree (Korf 1987). Initially, the system is in a "start state" which corresponds to the root of the tree ("level 0"). Successor states are generated by applying some operator to the previous state, and they are represented as children of the previous state in the search tree. The process of generating the children of a parent state is called *expansion*. Thus operators are applied to level 0 to produce the states of level 1, the states of level 1 produce those of level 2, and so on. This process is carried out until a "goal state" is generated which matches some criteria. In many applications, search is concerned

with the "path," or set of states visited, between the start state and the goal: specifically, the goal is known and the path to the goal is the thing being searched for. In the absence of any information except the desired goal, different search methods are characterized by the order in which states are visited. The two extremes of these are "breadth-first search" and "depth first search." Breadth first search proceeds by expanding a state to produce all possible successors prior to expansion of any of those successors. Depth-first search assumes some arbitrary ordering of states at a given level, and proceeds to search each of these states as it is generated. This means that all the successors of a state *n* at level *l* are generated prior to generating or searching state *n+1* at level *l*. A final point about state space tree search is that although the search is ordered as a tree, there is no inherent guarantee that a state will not be generated as an ancestor of itself. This implies that infinite cycles are possible, manifested as search carried out to an infinite depth. Many practical systems are required to keep track of states already visited in order to prevent this.

2.2.2. Heuristic Search

The undirected ordered search procedures outlined in the previous section comprise a class of methods known as *brute force*. Alternatively, search may be refined when the estimated "distance" from the goal is generated by some function which can be applied to any arbitrary state: this function is referred to as an *heuristic*. Heuristic search creates the possibility of searching for goals which are not known, since the goal can be said to be reached when the heuristic reaches some desired value. By this definition, the fitness measure used in Genetic Programming is an heuristic, and the difference between the fitness value of a program and some "ideal" fitness value may be viewed as distance from a goal. Many variations of heuristic search, most notably the A* method, are concerned with preserving the path of states from the initial state to the goal. Conversely, Genetic Programming is only concerned with achieving a goal state, namely some program which produces a satisfactory fitness value.

Rather than searching the state space tree in some predetermined order, heuristic search proceeds in an order that is determined by the value of the heuristic function. For non-path-preserving search, a primary distinction between methods may be made by the amount of memory that they keep about the states that have already been visited. Hill-climbing is a memoryless method which embodies the graph search version of the *greedy algorithm* (Hu

1982). The successors of the current state are generated and evaluated according to the heuristic. If the successor state having the best heuristic value (closest to the goal) is better than the current state, that successor state is chosen to become the new current state, and the process is reapplied. Otherwise, the process terminates. Thus the name hill climbing, since the algorithm converges to the top of the nearest hill in the fitness landscape (that is, in the case of maximization: if performing minimization the algorithm seeks the bottom of the closest valley). There is no guarantee that this local extremum is the global extremum, and hill climbing may terminate without achieving a satisfactory value of the heuristic function. An alternative method is to store all states which have been heuristically evaluated but not expanded in a priority queue. These states are ordered according to their heuristic value, with the best being first, and the resulting algorithm is called "best-first" search. Best-first search always removes the state at the head of the priority queue for expansion, and so the next state searched is not necessarily a child of the current state. Likewise, when the situation of finding an heuristic "dead end" occurs, as in the hill climbing example, the search can back up to some previously unexpanded state even though that state is less optimal than the terminated branch.

2.2.3. Genetic Programming as Beam Search

Although best-first search is guaranteed to find a globally optimal value of the heuristic function, the size of the priority queue can grow exponentially with the depth of search performed. A compromise to this situation is "beam search" (Lowerre and Reddy 1980). Beam search is very much like best-first search with the exception that the priority queue (memory) is set at some size limit, typically fixed (Rosenbloom 1987). Thus there is a limit to which states the search can be backed up to. The resulting tradeoff is that it is possible for states uniquely leading to the optimal state to be eliminated. The imposition of limits which are high enough not to impede search is entirely problem dependent.

There is a strong correlation between beam search and Genetic Programming, with a fixed population size serving as memory and fitness serving to stochastically assign priority. To avoid confusion, we make strong differentiations in this argument between the terms *trees* and *expressions*. Although expressions may be represented and operated upon as parse trees, we here restrict our terminology so that the term trees refers only to *search trees* as discussed in the previous section. Notice also that we use the term expressions to represent programs, and

subexpressions to denote sub-parts of programs, as opposed to subprograms, which can refer to an entirely different type of structure (Koza 1994).

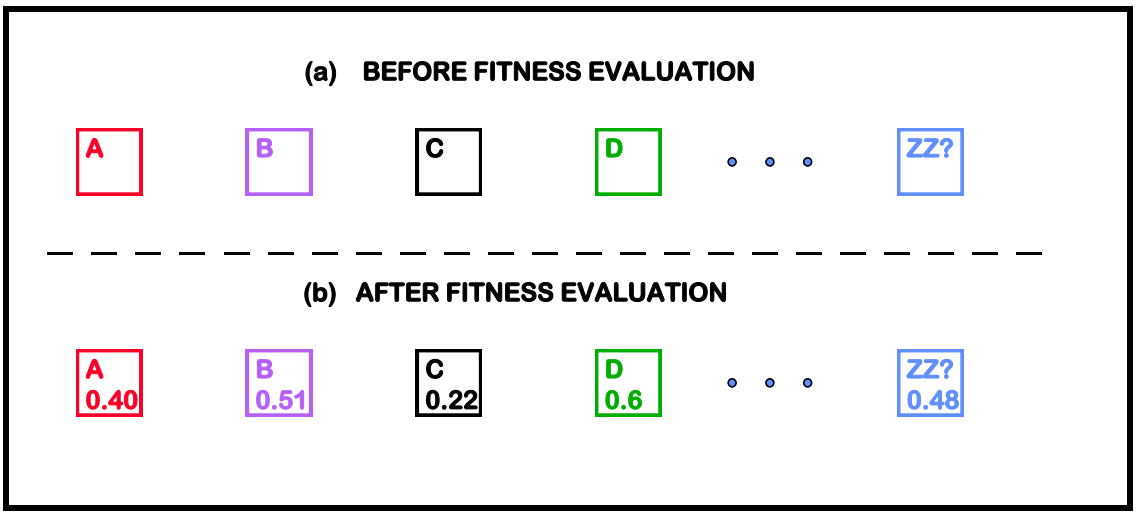


Figure 2.8: Genetic search begins with the random generation of programs which are "nodes" in the state-space search tree. (a) A population of N nodes are generated at random by the algorithm shown in Figure 2.6. (b) Those programs are evaluated to determine their heuristic worth.

The states of the genetic search tree are expressions. Rather. than being the successors of a single initial state, the initial population of N expressions are N randomly generated states (Figure 2.8a). A state is visited by being created and evaluated, as an expression, for fitness (Figure 2.8b). The selection operator works together with recombination, cloning, and possibly mutation operators to create successors of expressions. Successors are created in batches of N new states, called a generation, with these new states replacing the N states of the previous generation. Among the set of N new states may be some arbitrary number of states also contained in the previous generation. The number of states which are different between the previous generation and the new generation is called the generation gap (De Jong 1975; 1993), which can range in number from 1 to N.

2.2.3.1. Selection and Stochastic Priority

The population of N expressions is analogous to a beam search priority queue with limited size N, ordered by fitness (Figure 2.9a). States are selected for partial expansion based upon order in the queue. The analogy is imperfect in the sense that the head is not necessarily chosen to have successors generated, but it is more likely to have successors than any other element. As an example of stochastic selection, we use the K-tournament method (Koza 1992), although

there are many others (Goldberg 1989). This method proceeds by choosing K members uniformly from within the queue, and selecting that member which is closest to the head. In general, selection is performed N times to generate the N members of the queue representing the next generation. During this process no elements of the queue are removed: instead, we can consider the generation of a new "interim" queue containing states for which successors will be generated, tagged by the operators which will be used to generate those successors.

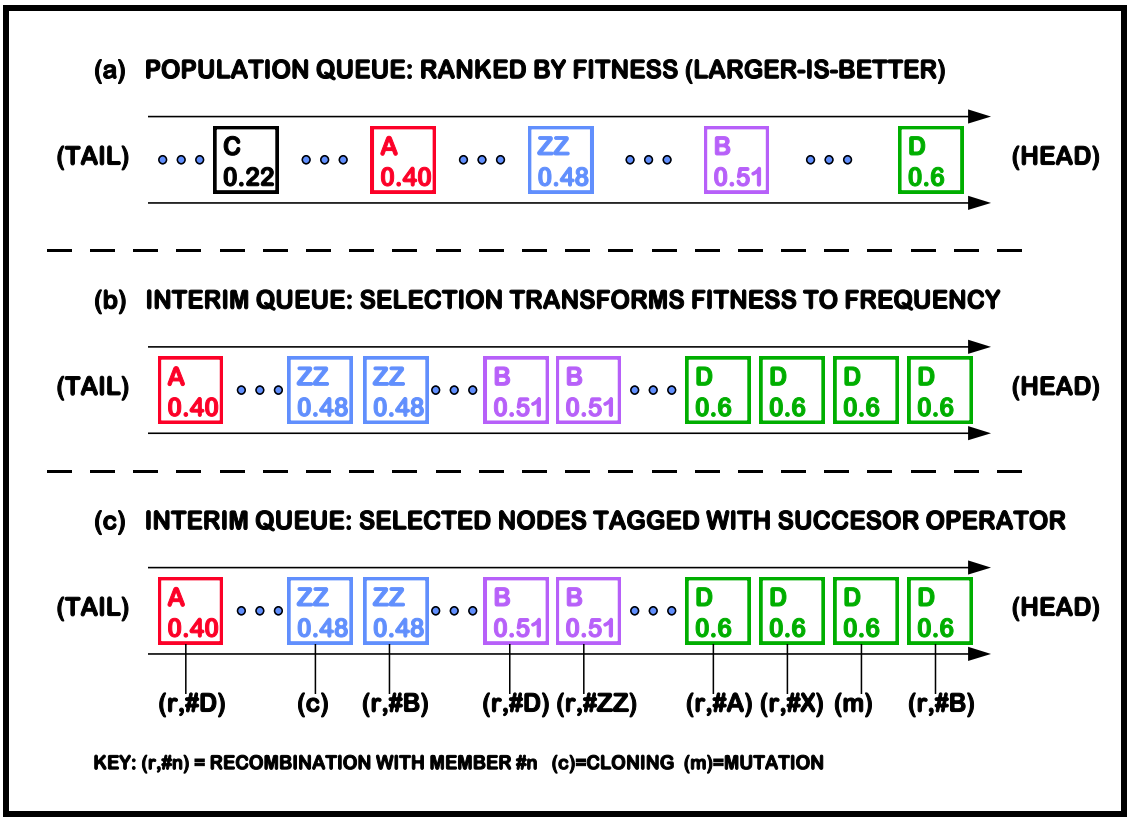


Figure 2.9: Genetic "beam search" on the space of programs: (a) The population of N nodes are ranked by fitness to form a "stochastic queue" in which members near the head are more likely to have successors generated. (b) Selection forms an interim queue of N nodes, drawn with replacement from the N members of the population queue. Those near the head of the population queue are drawn multiple times, on average, while those near the tail are not likely to be drawn. (c) Nodes selected into the interim queue are shown tagged with the operators which will be used to generate their successors. The recombination operator "r" is unique with respect to AI search in that it is a binary operator which combines parts of existing solutions.

The interim queue transforms the distribution of the population queue. States which are near the head of the population queue are represented (in probability) with a high frequency of

occurrence in the interim queue, while those near the tail of the population queue occur with low probability (Figure 2.9b). The selection of states inserted to the interim queue can be considered as a two-step process: first, the *operator* is chosen which will be used to generate successors. This is necessary since an operator may require either one or two states to operate upon. Although there are many possible operators we consider only three here, namely recombination, cloning, and mutation, which are described in detail in the following sections. The operators are generally chosen stochastically according to some a priori distribution which is a program input parameter. The second stage is the selection of the (one or two) states to be partially expanded by the chosen operator. In the case of unary operators we may simply insert the selected state into the interim queue along with a tag representing the operator. In the case of a binary operator such as recombination, the tag must represent both the operation and the chosen "partner" which will participate in the operation (Figure 2.9c).

The result, on the average, of applying selection to form the interim queue is that the number of successors of a particular state in the previous population is proportional to its position in the previous queue. Those states which were at the head of the population queue will likely have many members in the interim queue, and those which were at the tail will likely have none. Since each entry in the interim queue produces one successor this in turn means that the number of successors generated is also proportional to position in the original priority queue. This is analogous to many implementations of beam search, which partially expand the most promising states (Bisiani 1987). It is important to realize that only partial expansion of a state is usually possible in genetic program search, since the number of possible successors for a state (using operations other than cloning) is very large in comparison to the population size, or queue size, N. The approximate branching factors of the search are determined by which successor operators are used and the relative frequency with which they are applied.

2.2.3.2. Cloning

Cloning is the operator which maps a state of the search tree into itself. This is important in the sense that it enables the search to "remember" a state from generation to generation, allowing search to back up to that state. A more subtle effect is that cloning can control the depth-vs-breadth characteristics of search. This is because a state with a high relative fitness may wind up with multiple copies of itself in the following generation (i.e., analogous to multiple entries in

the beam search priority queue). This in turn increases the probability of a state having multiple successors generated - the portion of the search tree descending from that state is "bushier," or subject to less pruning of successors, than other states having a single entry in the queue.

2.2.3.3. Recombination

Recombination is a binary operator which generates a successor for each of two open states, as discussed in Section 2.1.2.3. As illustrated there it operates by swapping subexpressions between copies of two parent expressions. If we consider a state as a "solution" to a heuristic problem, then this operation can be viewed as combining partial solutions from states previously arrived at by search. We can arbitrarily adopt the convention that one successor is generated for each parent, with parent and successor differing by the subexpression which was exchanged. It is easy to compute an upper bound s_i on the number of possible successors for a specific node *i* which can be derived in a given generation:

2.1
$$s_i = SIZE_i \cdot \sum_{j=1}^{N} SIZE_j \approx SIZE_i \cdot N \cdot \overline{SIZE}$$

Where $SIZE_i$ is the count of function and terminal tokens contained in expression *i*. Thus the number of successors of *i* is the product of the size of expression *i*, the population size N, and the average size of expressions in the population. This comes from the idea that an expression contains $SIZE_i$ points into which a subexpression may be inserted from another expression, and likewise each of the N expressions *j* in the population contains $SIZE_j$ subexpressions which may be donated to the insertion site of member *i* in order to form a new offspring. This is a weak upper bound since, for example, subexpressions consisting of a single terminal are not unique in the population. In fact, repeated selection guarantees that many subexpressions occur multiple times according to Holland's schema theorem (Holland 1975; 1992). Regardless, this is useful in computing the upper bound on average branching factor of search in a single generation:

2.2
$$b = \frac{1}{N} \sum_{i=1}^{N} SIZE_i \sum_{j=1}^{N} SIZE_j \approx N \cdot \overline{SIZE}^2$$

This is just the average of all s_i in the population. Therefore, the upper bound on average branching factor of the search tree is greater than the population size by a large factor, namely the square of the average size of expressions in the population. We will see in later chapters, and in references such as (Koza 1992) and (Kinnear 1994) that average expression size can

typically range from tens to hundreds. Note that not all successors of a state will be generated with equal likelihood since selection biases the insertion of subtrees towards those drawn from highly fit individuals (i.e., those at the head of the search queue). Finally, it is important to realize that Equations 2.1 and 2.2 apply only to a single generation. Although the computed value of b will not change if the average expression size remains constant across generations, the shifting set of open states will change the set of successors which are reachable from a given parent state (assuming that the state survives across generations, for example, by cloning).

2.2.3.4. Mutation

Mutation is a unary operator which, like recombination, replaces a subexpression within a copy of a queue member by another. Rather than being donated from a second queue member, however, the inserted code is randomly generated from essentially the same procedure used to create members of the initial population. The number of expressions which can be generated randomly is exponential in the cardinality of the function set raised to a power of the size of the randomly generated expression. This number is of a greater order than the quadratic function of expression size which bounds the number of search trees. Therefore the branching factor of search induced by mutation can be far greater than that due to recombination. As discussed earlier, empirical studies (Koza 1992) indicate that mutation is of little practical benefit in Genetic Programming. Conceptually, this may be due to the fact that mutation inserts subexpressions sampled randomly from the space of all subexpressions, whereas those subexpressions inserted due to recombination are drawn from a much smaller (and fitter) set comprising parts of states which have been generated through prior iterations of search.

2.2.4. Summary

Genetic Programming bears strong similarities to beam search. Probably the most unconventional aspect with respect to existing methods of AI is the concept of the binary recombination operator, which combines "partial solutions" in order to generate successors. It is also somewhat unusual among problems classically studied by AI in the very large number of potential successors which can be generated from each state. Like beam search, Genetic Programming maintains a limited number of states open for further search. Many forms of beam search generate and evaluate all possible successors of a state prior to pruning back those which fall below some threshold of worth. This is not possible in Genetic Programming because

(1) the set of possible successors (due to recombination) is bounded in size by a large multiple of the allowed number of open states and (2) this set will change from generation to generation. Since all possible successors of a state cannot be simultaneously examined there is no guarantee that the successors which are opened are the best of all possible successors. The process of recombination does favor the use of partial expressions which come from states nearer the head of the population "queue," and so the choice of successors from those possible is not uniformly random. Another deviation from the beam search analogy is induced by cloning, which allows states to remain open across generations but may also introduce duplicate entries in the search queue. Finally, the queue implicitly formed by a population in Genetic Programming is stochastic. Rather than having N elements inserted and deleted one at a time it is updated as a batch. The batch update process results in a new queue which looks statistically similar to the queue of a beam search system after being run for N iterations. Specifically, those nodes which were at the head of a beam search queue initially are likely after a number of iterations to have a number of successors near the head of the queue, and may still be present themselves, while those elements initially near the tail of the queue are likely to not have successors, nor to be present themselves.

In later chapters other methods of selection and recombination will be introduced, and the search analysis will be revisited in order to understand them better.

2.3. Other Methods of Adaptive Programming

Two methods of adaptive program construction other than Genetic Programming are briefly discussed in order to illustrate the varieties of approach to this field.

2.3.1. Tierra

Thomas Ray (Ray 1991) pioneered a unique programming paradigm through the creation of the Tierra system, a "world" where organisms written in an assembly language of complexity comparable to (real) genetic code freely evolve. The resulting populations display a wide variety of biological properties, leading Ray to speculate that Tierra-like systems may form an "alternate biology" for comparison and contrast to our own.

The standard version of Tierra breeds digital organisms which vie for memory and CPU time as a metaphor for food and sunlight. It loosely emulates a shared memory MIMD computer with a 5-bit zero-operand instruction set. Each "organism" within this environment is an assembly

language program which has its own virtual CPU with registers, stack, program counter, and flags. The system is initialized with a single self-replicating ``ancestor'' program residing in memory. This program copies itself into a block of free memory and, when done, executes a special instruction which (1) write-protects the new “child copy and (2) allocates to it a virtual processor. The reproductive cycle then begins anew. As memory runs out during the inevitable population explosion, organisms which are the oldest and/or most defective are deleted in order to make room. Mutation and imperfect transcription of offspring cause the population to diversify as generations progress. Organisms resulting from this evolutionary process adapt a rich variety of survival and competition mechanisms. These include code optimizations such as loop unrolling and biological properties such as parasitism and immunity.

External measures of fitness can be incorporated into Tierra, as reported in (Tackett & Gaudiot 1992). In that work Tierra is extended to incorporate environmental stimuli for which there are appropriate and inappropriate responses that have a direct impact upon an organism's rate of metabolism. These might serve as a metaphor for thermal or chemical gradients. Specifically, a two-bit pattern is broadcast by the system and the fitness of each organism is based on its ability to read those bits, compute their XOR value, and return that value to the system. Fitness is tabulated for each organism and genotype during the course of evolution, and the metabolic rate of each organism is determined by the performance of its genotype averaged over the population. The fine granularity of the Tierra system coupled with pressures for survival and reproduction make the simultaneous learning of computational functions a non-trivial computational task.

2.3.2. FOIL

Whereas Tierra uses a finer-grained and more primitive language than the high-level LISP used in Genetic Programming, quite the opposite approach is taken in the FOIL system (Quinlan 1990). FOIL generates declarative code in the form of Prolog programs using a nested loop which performs hill climbing. The outer loop is responsible for generating entire clauses (roughly equivalent to lines of Prolog code), while an inner loop generates the literals which comprise each clause.

A set of training examples drives the program generation process using an heuristic measure during the construction of a clause. Specifically, this heuristic examines the mutual information

gained due to the repartitioning of training data through addition of a literal. This means that if the addition of a literal does not change partitioning then it conveys no information, whereas if it creates a more accurate partition then it increases mutual information. The utility of the method is demonstrated on several machine learning problems.

FOIL learns programs which implement relations that are generally recursive in nature. Because of this there is a danger of generating sets of clauses which will lead to infinite recursion. Quinlan prevents this through restriction of the syntax of generated programs, ordering variables so as to restrict cases which lead to infinite recursion. This stands in contrast to the prevention of infinite iteration in Genetic Programming introduced in (Koza 1992). There the approach is to simply provide an overriding limit on the number of iterations performed.

As proposed improvements to FOIL, Quinlan proposes the addition of backtracking and (interestingly) beam search. Other work (Pazzani & Kibler 1990) combines the learning concepts of FOIL with rules based upon expert knowledge.

2.4 Other Issues in Genetic Methods

For completeness, a brief background is provided with respect to several special topics which are pertinent to the work described in later chapters.

2.4.1. Genetic Programming vs. Random Search

One frequently asked question by newcomers to the concept of Genetic Programming is "how is this different from random search?" The discussion of GP as heuristic search in Section 2.2 illustrated that it is a close variant of the beam search methods used in AI. Chapter 6 purposely reformulates selection in order to induce properties of random search and shows tremendous differences which result. Other researchers, namely (Koza 1992) and (Angeline, personal communication) describe experiments comparing the results of running GP for **G** generations with a population of **N** individuals per generation against the results of generating **NG** random initial individuals (i.e., the same *total number of individuals* are evaluated in each case). Not only are the best individuals resulting from GP far more fit than those randomly generated, but for sufficiently difficult problems the best of the **NG** random individuals are not significantly more fit than the best of the N random individuals comprising the first generation of the GP run.

2.4.2. Comparison to Neural Methods

Neural networks encompass a variety of methods and architectures. A neural network typically consists of two separate entities, the network structure and the learning algorithm. The network structure is not itself adaptively generated but is instead based on a neural model conjectured by researchers It is a collection of numbers, usually in the form of vectors called "weights," and the equations which govern their relations, called "activation functions." Together these define the interaction between inputs, intermediate computations, and outputs. The learning algorithm is likewise dictated by the researcher: it is the method by which the values of the weights are determined adaptively from training data. For a good general treatment of Neural Networks see (Hertz et al. 1991).

By way of comparison, almost all neural learning algorithms require either gradient information in the case of supervised learning such as Backpropagation (Rosenblatt 1962) (Rummelhart et al. 1986) or correlation measures in the case of unsupervised learning such as AVQ (Kohonen 1987) or ART (Grossberg 1988). By the same arguments used in the previous section, neither of these measures are available on the space of program trees: the "classical" methods of neural learning cannot generate programs.

2.4.3 Elements of the Theory of Genetic Algorithms

A theory of the operation of the Genetic Algorithm is presented in (Holland 1975; 1992) and extensions to the theory of GA are the subject of a series of biannual conferences (Rawlins 1991), (Whitley 1993). A brief outline of some of the issues of GA theory are presented here.

2.4.4. Schema Fitness and Building Block Theory

Section 2.1.1.4 introduced the notion of *schemata* in GA. A central idea introduced in (Holland 1975; 1992) is that for a given problem certain schemata form *building blocks* which are important components of the overall solution. It is in the ability of the Genetic Algorithm to locate building blocks as independent entities and then join them through recombination that the inherent power of the algorithm lies. Particular schema will propagate through the population of successive generations, on average, proportional to their *schema average fitness*. The schema average fitness is computed as the average of the fitnesses of all the individual genomes in which the schema appears throughout the population. Schemata with an average fitness greater than the average fitness of the population will increase in frequency with time, while those

below average will decay in frequency. In Chapter 3 we propose that (1) the concept of schema fitness may also apply to subtrees within Genetic Programs, causing those with high fitness to propagate throughout the population. Furthermore, due to the variable size and shape of Genetic Programs, these highly fit subtrees may appear many times even within a single program / tree. This phenomenon is demonstrated empirically in Chapter 3, while Chapter 7 goes on to examine schema fitness directly.

2.4.5. Synthetic Functions: Deception and the Royal Road

A more recent theoretical concept in the world of GA is the notion of *hardness* (Goldberg, 1989). This concept centers around the idea that the schema fitness concept does not apply in the case of *epistasis*, or nonlinear interaction of genes, and purports that the structure of some problems can actively mislead the GA. The most studied area in this field is that of *deception*. In order to study deception, synthetic problems are often constructed which assign specific fitness values to specific bit patterns in order to exercise precise control over problem structure. As an example of a synthetic deceptive function, consider a 4-bit problem where the string "1 1 A B" is assigned a low average fitness, where the string "A B" can be any 2-bit string other than "1 1". Likewise, the strings "A 1 1 B" and "A B 1 1" are given a low average fitness, while the optimal pattern which provides the highest fitness is the string "1 1 1 1". Thus, a high fitness string must be formed from building blocks with a low average fitness. Among other things, this class of problems may be used to help establish a benchmark on upper-bound times to solutions, as well as providing a worst-case comparison with other learning methods.

Royal Road functions are the opposite of deceptive problems: they are constructed in such a way as to be particularly easy for GA to solve. These can be used to check that the best-case performance of GA converges in probability to theoretical predictions. In (Forrest and Mitchell 1993) it was shown that they do not. Subsequent analysis of that work illustrated the susceptibility of GA to "hitchhiking," in which highly fit building blocks become attached, by coincidence, to adjacent unfit building blocks which propagate throughout the population as well, and subsequently prevent highly fit building blocks at adjacent locations on the genome from joining up with each other. It was subsequent investigation in (Forrest and Mitchell 1993) which led to the insertion of introns in GA to combat hitchhiking and other problems. Chapter 6 presents deceptive problems and Royal Road problems in the context of Genetic Programming.

2.4.6. Parallelism in the Genetic Algorithm

The most computationally intensive part of any genetic method is the fitness evaluation of the population. Hundreds or even thousands of training examples must be processed by each individual in each generation in order to get a good statistical picture of the performance (fitness): this can be compounded many fold when each test case consists of many evaluations. As an example, consider a game-playing algorithm where each of the test cases is a game to be played, with many moves, where the phenotype of an individual must be evaluated at each move. Thus the structures (in the case of GA) or programs (in the case of GP) at each generation must be evaluated a total of NCE times, where N is the size of the population, C is the total number of test cases in the training set, and E is the number of evaluations of the individual required for a single test case.

Hillis (1991) used a Thinking Machines CM2 to parallelize the evaluation of binary strings in a GA, where each string represented a sorting network. The same machine has been used in (Collins 1992) with a similarly distributed implementation. Ackley (1992) likewise exploited SIMD parallelism for GA using a MasPar MP1 on a graph partitioning problem. In all cases, the size of the population was set to be a small integer times the number of available processors, with each processor performing the evaluation of a different individual in parallel. Here we see the critical difference between parallelizing GA vs. GP. Genetic Algorithms, used by Hillis and Ackley, operate on *data*: each member of the population is a data object, and all members are evaluated in lock-step by the same program running on all PEs. The population to be evaluated by GP consists of *programs*: not only are each of these programs executing different instructions at any given step, but in addition each program is of a different size and complexity, requiring different amounts of time to evaluate the set of test cases. Therefore, appropriate architectures for parallel GP would be loosely coupled systems including MIMD/SPMD systems such as the Thinking Machines CM-5, or client-server architectures distributed over a network of workstations.

Chapter 3

Genetic Programming Applied to Image Discrimination

Comparative Performance on a Difficult Real-World Induction Problem

We apply Genetic Programming (GP) to the development of a processing tree for the classification of features extracted from images: measurements from a set of input nodes are weighted and combined through linear and nonlinear operations to form an output response. No constraints are placed upon size, shape, or order of processing within the network. This network is used to classify feature vectors extracted from infrared imagery into target/non-target categories using a database of 2000 training samples. Performance is tested against a separate database of 7000 samples. This represents a significant scaling up from the problems to which GP has been applied to date. Two experiments are performed: in the first set, we input classical "statistical" image features and minimize misclassification of target and non-target samples. In the second set of experiments, GP is allowed to form it's own feature set from primitive intensity measurements. For purposes of comparison, the same training and test sets are used to train two other adaptive classifier systems, the binary tree classifier and the Multilayer Perceptron / Backpropagation neural network. The GP network achieves higher performance with reduced computational requirements. The contributions of GP "building blocks," or subtrees, to the performance of generated trees are examined.

3.1. Introduction

Genetic Programming has been demonstrated in a variety of applications, many of which have known optimal solutions determined in advance. This leaves open the question as to whether GP can "scale up" to real-world situations, where answers are not known, data is noisy, and measurements may be of poor quality. We attempt to address this by constructing such a problem: noisy image data are segmented and processed into statistical features. These features are used to assign each image to a "target" or "non-target" category. Because they are

constrained by processing requirements, the segmentation and feature measurement are of a coarse and sometimes unreliable nature. Because of this it is likely that overlap between classes in feature space exists, and hence discovery of a 100% correct solution is unlikely. Two experiments are performed: in the first we insert a GP-generated tree into an existing system in order to test it against other well-established adaptive classifier technologies, specifically Multilayer Perceptron/Backpropagation (Rummelhart and McClelland 1986) and decision tree (Quinlan 1983) methods . We then proceed to examine whether GP can be inserted at an earlier stage of processing, obviating costly segmentation and feature extraction stages.

3.2. The Problem

3.2.1. Target / Non-Target Discrimination

Target / non-target discrimination is an important first stage in Automatic Target Recognition (ATR), and in general for systems which require attention to a small sub-area (or areas) within a much larger image. Whereas a highly sophisticated pattern recognizer may make fine discriminations between subtly different patterns (Daniell et al. 1992), it generally does so at a very high computational cost. A much more efficient way to process information is to employ a simpler "*detection*" algorithm to the entire image, identifying subareas of interest. These areas are only coarsely classified as "target" or "non-target," and with lower reliability than the recognizer. Only target areas are passed from the detector to the recognizer, reducing image bandwidth and throughput requirements. Thus there is a constraint on this problem that GP must produce a solution which is computationally efficient in order to be useful. There is further advantage to be gained if GP is able to bypass costly processing associated with feature extraction.

3.2.2. Image Database

Data was taken from US Army NVEOD Terrain Board imagery. These are 512x640 pixel images which simulate infrared views of vehicles, including tracked and wheeled vehicles, fixed- and rotary- wing aircraft, air defense units, and a wide variety of cluttered terrain. The range, field of view, and sensor resolution are such that individual targets occupy between 100-300 pixels. There are a total of about 1500 images containing multiple target and clutter objects, providing a total of about 13,000 samples to be classified. Training and test data for the experiments described here were drawn randomly from these samples.

3.2.3. Feature Extraction

In the experiments described here we have used Genetic Programming to construct classifiers which process the feature vectors produced by an existing algorithm - specifically the Multi-function Target Acquisition Processor (MTAP) ATR system (Hughes 1990). This system performs two sequential steps of image processing.

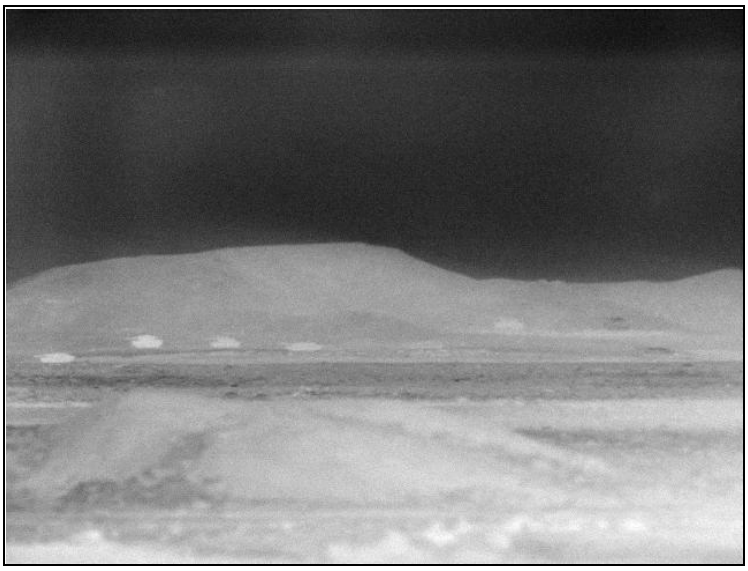


Figure 3.1: Eight (!) tank targets, light clutter (from US ARMY NVEOD Terrain Table database).

3.2.3.1. AntiMean Detection Filter

This system uses an antimean filtering to extract "blobs" in a range of sizes conforming to those of expected targets. The image is divided into a mosaic of 62x62 pixel overlapping regions, or "large windows." These regions are themselves divided into overlapping 5x5 "small windows." When a blob of appropriate size is detected, it is described in terms of seven primitive features: contrast with the local background, global image intensity mean and standard deviation, and the means and standard deviations of the "large window" and "small window" in which it is centered. Later we will apply GP to the classification of these features. In the conventional MTAP algorithm, however, they are passed to a second processing stage.

3.2.3.2. Segmentation and Feature Extraction

In the MTAP system, the seven features from the antimean detection filter are passed through a simple linear classifier in order to determine whether segmentation and feature extraction should be performed upon the blob. If so, the large image window undergoes a 3:1 decimation

filtering to reduce bandwidth and statistical measures of local intensity and contrast are used to perform a binary figure / ground segmentation, resulting in a closed-boundary silhouette. Twenty moment- and intensity-based features are extracted from this segmented region. They are summarized in table 3.2. Because this segmentation scheme depends on fixed thresholds under varying image conditions, silhouettes for a given target type can vary significantly. Likewise, since features do not encode detailed shape properties it is possible for non-target objects such as oblong rocks to be encoded into target-like regions of feature space. Thus these feature vectors may display significant variance as well as overlap between target and non-target classes.

Table 3.1: Seven primitive features.

F00	Size Filter Value (“Blob Contrast”)
F01	Global Image Intensity Mean
F02	Global Image Intensity Standard Deviation
F03	“Large Window” Intensity Mean
F04	“Large Window” Intensity Standard Deviation
F05	“Small Window” Intensity Mean
F06	“Small Window” Intensity Standard Deviation

Table 3.2: Twenty statistical features from segmented image.

F00	Radius of Gyration
F01	Rectangularity
F02	$Height^2$ / $Width^2$
F03	$Width^2$ / $Height^2$
F04	Normalized Contrast
F05	Symmetry
F06	Range to Target
F07	Depression Angle
F08	$Perimeter^2$ / Area
F09	Normalized Average Segmentation Greylevel
F10	Area
F11	$Height^2$ / Area
F12	$Height^2$ / $Range^2$
F13-F17	Higher-Order Moments
F18	Area / $Range^2$
F19	Polarity of Contrast

3.3. Approach

The fitness of an individual tree is computed by using it to classify about 2000 “in-sample” feature vectors. At each generation, the individual which performs best against the training set is additionally run against 7000 out-of-sample feature vectors which form a validation test set. Only results against the out-of-sample validation set are reported.

3.3.1. Function Set

Both experiments share a common function set: **F** = {+, -, *, %, **IFLTE**} represents four arithmetic operations which form 2nd order nodes (i.e., 2 arguments) and a conditional operation which forms 4th order nodes (4 arguments). The +, -, and * operators represent common addition, subtraction, and multiplication, while % indicates "protected" division: division by zero yields a zero result without error (Koza 1992). The conditional IFLTE returns the value of the third argument if the first argument is less than the second, otherwise the fourth argument is returned.

3.3.2. Terminal Set and Fitness Function for Experiment 1

The terminal set **T** = {**F00**...**F19**, **RANFLOAT**} for the first experiment consists of the 20 segmented "statistical" features shown in Table 3.2 as well a real random variable RANFLOAT, which is resampled to produce a constant value each time it is selected as a terminal node. The resulting tree takes a bag of floating-point feature values and constants as input, and combines them through linear and nonlinear operations to produce a numeric result at the root of the tree. If this result is greater than or equal to 0, the sample is classified as a target, otherwise it is classified as clutter (non-target). Testing of trees against the 2000-sample set results in two measurements: the first is probability of incorrect classification, or the total fraction of samples assigned to the incorrect category. It had previously been observed that performance could be enhanced by including a larger number of target samples than clutter samples in the training database (see Section 3.5: Results). This gives rise to the problem that a classifier which says *everything* is a target may get a relatively high fitness score while conveying no information in the Shannon sense (Shannon 1948). To counteract this, a second measure of fitness was added: the a posteriori entropy of class distributions **H**(**class**|**output**) after observing classifier output. These multiple objectives are minimized via mapping to the

Pareto plane and subjecting them to a nondominated sort (Goldberg, 1989a). The resulting ranking forms the raw fitness measure for the individual.

3.3.3. Terminal Set and Fitness Function for Experiment 2

In the second experiment, only the seven primitive intensity features from the antimean discriminant filter are used. The terminal set **T** = {**F00**...**F06**, **RANINT**} consists of these seven integer features and an integer random variable which is sampled to produce constant terminal nodes.

The fitness function is reformulated from experiment 1 for "historical reasons," namely the constraints that were placed on the original MTAP system. These state that the detection filter should be able to find five (5) targets in an image with 80% probability of detecting all of them. This means that individual Probability of Detection (**p(D)**) must be $0.8^{0.2}$, or about 96%. With this figure fixed, we seek to minimize the Probability of False Alarms (**p(FA)**), the chance that non-targets are classed as targets. This is done in the following manner: an expression is tested against each sample in the data base, and the values it produces are stored into two separate arrays, one for targets and the other for clutter. These arrays are then sorted and the threshold value is found for which exactly 96% of the target samples produced a greater output. We compare this value with those in the clutter array in order to determine what percentage fall above this threshold value. This percentage is the False Alarm Rate which we seek to minimize.

3.4. Performance Comparison

3.4.1. Multilayer Perceptron and Binary Tree Classifier

As an experimental control the same test and training database are used to build and evaluate two other classifiers. The first is a multilayer nonlinear Perceptron "neural network" trained via the Backpropagation algorithm with adaptive step size η (Hertz et al. 1991) and 2 output nodes (one for each class). For both experiments, the "desired output" activation for this neural net is {1.0, 0.0} when presented with a target sample, and {0.0, 1.0} when presented with a clutter sample. Differences between these desired outputs and those actually obtained form error terms to be fed back. Weight updates take place after each sample presentation. Training inputs to the network are normalized so that values do not exceed an absolute value of 1.0. A

threshold difference between network outputs is used to determine if a sample is target or clutter. This threshold is 0.0 for experiment 1 and variable for experiment 2. The second classifier tested is a binary decision tree classifier (Quinlan 1985; Kanal 1979) which partitions feature space into hyperrectangular regions, choosing features and their thresholds at decision nodes based upon the maximization of mutual information (also known as information rate) (Papoulis 1965; 1984). This particular binary tree classifier was originally developed and optimized for use with the feature set of experiment 1 (Hughes 1990). It was not used in experiment 2.

3.4.2. A Basis for Comparison

In performing comparisons between methods there is some question as to what is a fair basis for assessing the work which is required to obtain results for each. When this effort was started CPU time was not a fair basis since the original GP system was written in LISP, which is less computationally efficient than the C language used to implement this version of Backpropagation. Instead, we consider that there was equivalent human effort required to make the two methods work correctly. Both the GP and neural net systems were obtained from second-hand sources. GP was run using a standard function set which was included in the software and had previously been used in the "Nested Spirals" classification problem (Koza 1992). All of the GP system parameters were left at their default settings as were those of the Backpropagation algorithm. In both cases, the default parameter settings had previously shown satisfactory performance on several other problems. Some cursory experimentation was performed with GP to determine a minimum population size which produced adequate performance, and likewise some tests were run with the neural network to determine the appropriate number of hidden layers and neurons. A few hours of labor were required to code and debug the fitness functions used for GP in experiments 1 and 2. The Perceptron network required normalization of input data, which was not required by GP. Results for the decision tree used in experiment 1 were obtained previously under a separate study (Hughes 1990).

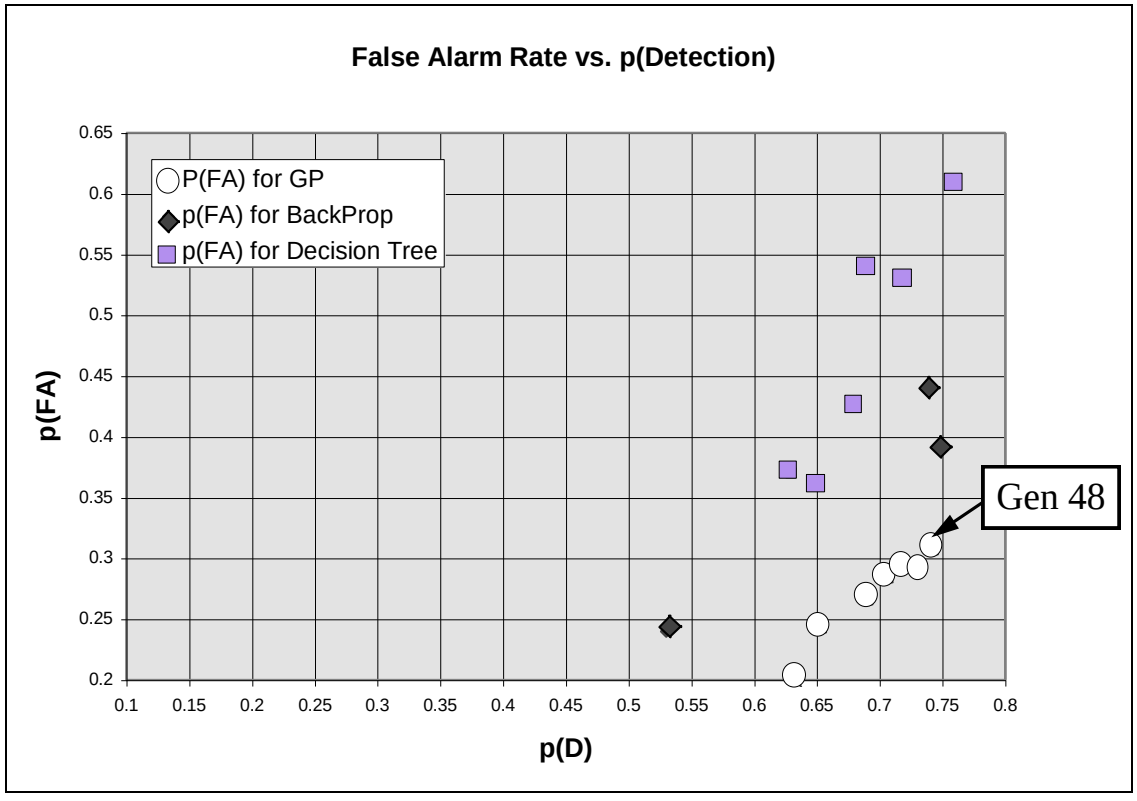
3.5. Results

For all experiments performance was plotted as a function of Probability of False Alarm **p(FA)** vs. Probability of Detection **p(D)**. Probability of False Alarm is the fraction of non-targets classified as targets divided by the total number of non-target samples. Probability of Detection

is the fraction of targets correctly classified divided by the total number of target samples. Ideally a system should achieve a p(FA) of 0.0 and a p(D) of 1.0.

3.5.1. Experiment 1

A total of twelve runs were performed using GP with different random seeds. Three different clutter-to-target ratios (C/T ratio) were used in the training database: 0.5:1, 0.71:1, and 1:1, with 0.5:1 consistently producing the best results.[3] Each run used a population of 500 and run length of 60 generations, resulting in the analysis of a total of 120,000 individuals for each C/T ratio. Backpropagation was similarly tested using three C/T ratios, again with 0.5:1 producing the best results. The network used 20 input nodes (the 20 features) and 10 hidden nodes (empirically determined to be the best number). Four random initial weight sets were used for each C/T ratio, resulting in a total of 12 learned weight sets. After each training epoch each network was tested against the separate validation test set. The best result achieved against this test set was reported as the figure-of-merit.



[3] C/T ratios less than 0.5:1 were found to produce particularly lousy results for both GP and neural approaches.

Figure 3.2: Comparative performance of Genetic Programming, Neural, and Decision Tree methods. The objective is to minimize probability of false alarm p(FA) while maximizing probability of detection p(D).

The binary tree classifier was tested using the three C/T ratios, and additionally tested with an input set that eliminates gray-level features. There is no random initial element. Various system parameters of the tree builder were refined during its development cycle.

Figure 3.2 summarizes the performance of the three classifiers. The three points corresponding to Backpropagation depict the best performance achieved against the 3 C/T ratios. The six points shown for the binary tree correspond to three C/T ratios for the two input feature sets described above (the cluster of three points with higher p(FA) were obtained with gray level features omitted). The 10 points shown for GP are the best-of-generation individuals for selected generations in a single evolution run with C/T ratio of 0.5. All three methods were able to achieve p(D) of 74%-75%. Genetic programming, however, achieves a 31% false alarm rate, some 8% lower than the 39% p(FA) achieved by Backpropagation for the same p(D), and 30% lower than the 61% figure achieved by the Binary Tree.

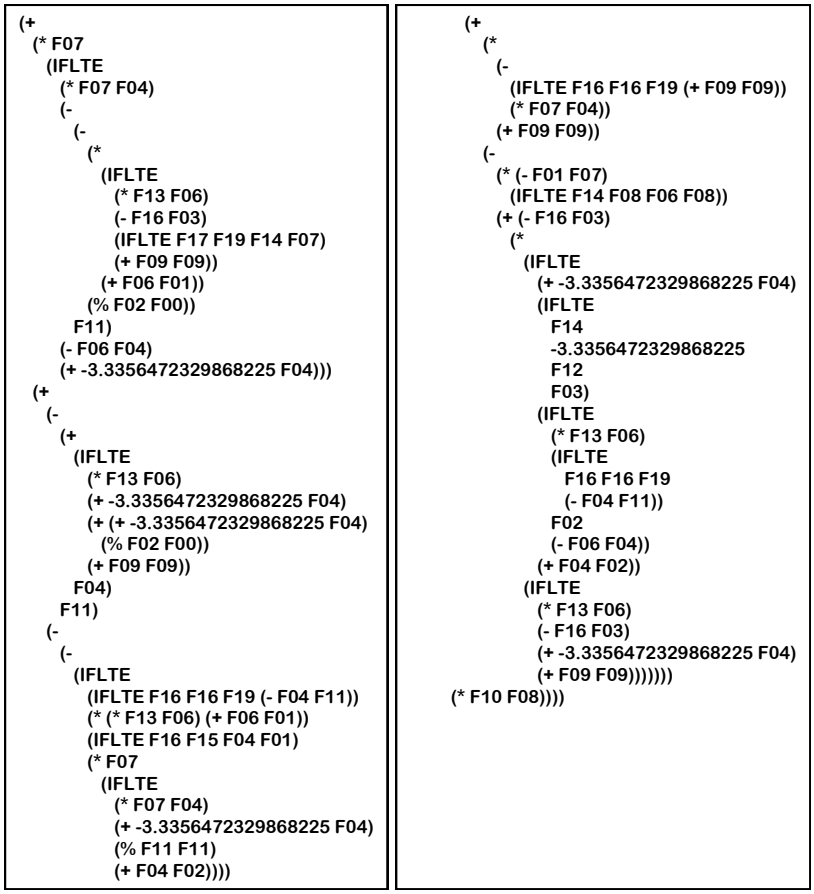


Figure 3.3: Best-of-generation expression for Generation 48 of the "most successful" GP run for Experiment #1.

Figure 3.3 shows the best-of-run individual for generation 48 (indicated by the arrow in Figure 2), with the dendritic tree represented in LISP program format. This program achieved 74.2% correct with a 30.8% false alarm rate. Counting function nodes we see that there are 55 mathematical operations required and 15 logical comparisons / branches. By comparison, the Backpropagation network requires 440 math operations and 12 nonlinearities: about 8x more computation is required in order to achieve 8% *lower* performance.

3.5.2. Experiment 2

As with experiment 1, Genetic Programming was run four times for 60 generations with a population of 500 and three C/T ratios using fitness function described in Section 4.3. Training and testing for the Backpropagation network were performed exactly as described in sections 5 and 6.1, except that this network contained seven input nodes and four hidden nodes. For both GP trees and Neural Networks a C/T ratio of 0.5:1 resulted in best performance.

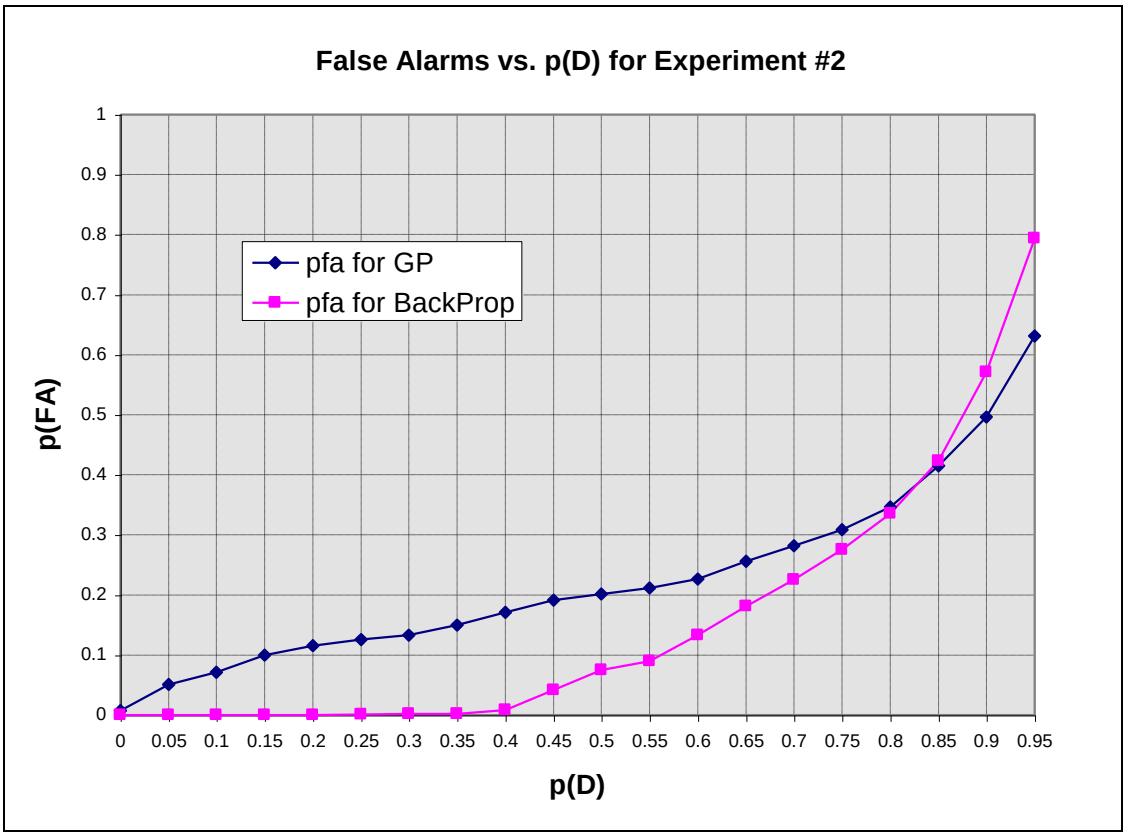


Figure 3.4: A tradeoff between detection and false alarms can be achieved by varying the threshold which distinguishes targets from non-targets. The intent of experiment #2 is to minimize false alarms at the point p(D) = 0.96 using a set of "primitive" intensity measurements.

Figure 3.4 depicts the performance of the Backpropagation and GP systems. The dashed vertical line indicates the desired p(D) of 96%. For this value of p(D) the best expression derived by Genetic Programming produced a 65.8% false alarm rate, while the best network derived by Backpropagation produced 82.6%. By varying the p(D) threshold, we generate curves showing p(FA) for these other values. This provides an insight about the two algorithms: GP used a fitness function which was specifically tuned to the desired p(FA) performance, whereas the Backpropagation training was independent of the threshold (used only *after* training) with no straightforward way to incorporate the 96% constraint. Thus GP chooses a function which specifically trades off performance at high p(D) values for performances at lower p(D) values.

```
(% (- F02 F00)
  (+ (+ (% (- (* F02 F03) F06)
                (+ (- F02 F00) (- F03 F00)))
              (- F03 F00))
    (- F03 F00)))
```

Figure 3.5: Best-of-generation 5 for the "most successful" GP run of experiment #2.

The individual whose performance is depicted in Figure 3.4 is shown in LISP expression form in Figure 3.5. This function requires 12 mathematical operations, compared to the 72 math operations and 6 nonlinear function evaluations required by the Backpropagation network. In and of itself it displays some remarkable properties: it does not make use of the IFLTE conditional branch, and ignores the input features F01, F04, and F05. Likewise, the system makes repeated use of synthesized "metafeatures," (- F02 F00) and (- F03 F00). Clearly this gives insight into which features are and are not useful for the low level discrimination task. Another interesting interpretation of these results is that F03 and F02 are localized measures of intensity, while F00 is a global measure of contrast. In terms of image processing, the expressions (- F02 F00) and (- F03 F00) represent local measures of contrast which are the kind of features a human expert might formulate: indeed, measures of contrast form some of the 20 features used in Experiment 1! In theory, this solution could tell a user with little expertise in image processing that contrast is a powerful feature. Chapter 7 will revisit this issue by examining automated means for extracting and reporting important features and relationships discovered by the Genetic Programming process.

The atypically small size of this expression[4] allows us to make an analysis of its major subcomponents in terms of their individual fitnesses: Figure 6 shows performance obtained by decomposing this expression into it's three unique subtree components, which we postulate comprise "GP building blocks" (Holland 1992). The building block (- F02 F00) appears, at p(D) values below 60%, to display performance in the same ballpark as that of the Backpropagation network. Significantly, all three schemata approach 100% p(FA) at 96% p(D). Thus they are in no way contributing to the fitness of the parent expression by a principle of superposition, but rather by their nonlinear interactions. Although GP may have a different notion of schemata

[4]Other trees of comparable (but slightly lower) performance were usually 2-4 times as large.

and a nonbinary alphabet, we suggest that this observation underscores principles previously set forth in (Goldberg 1989b).

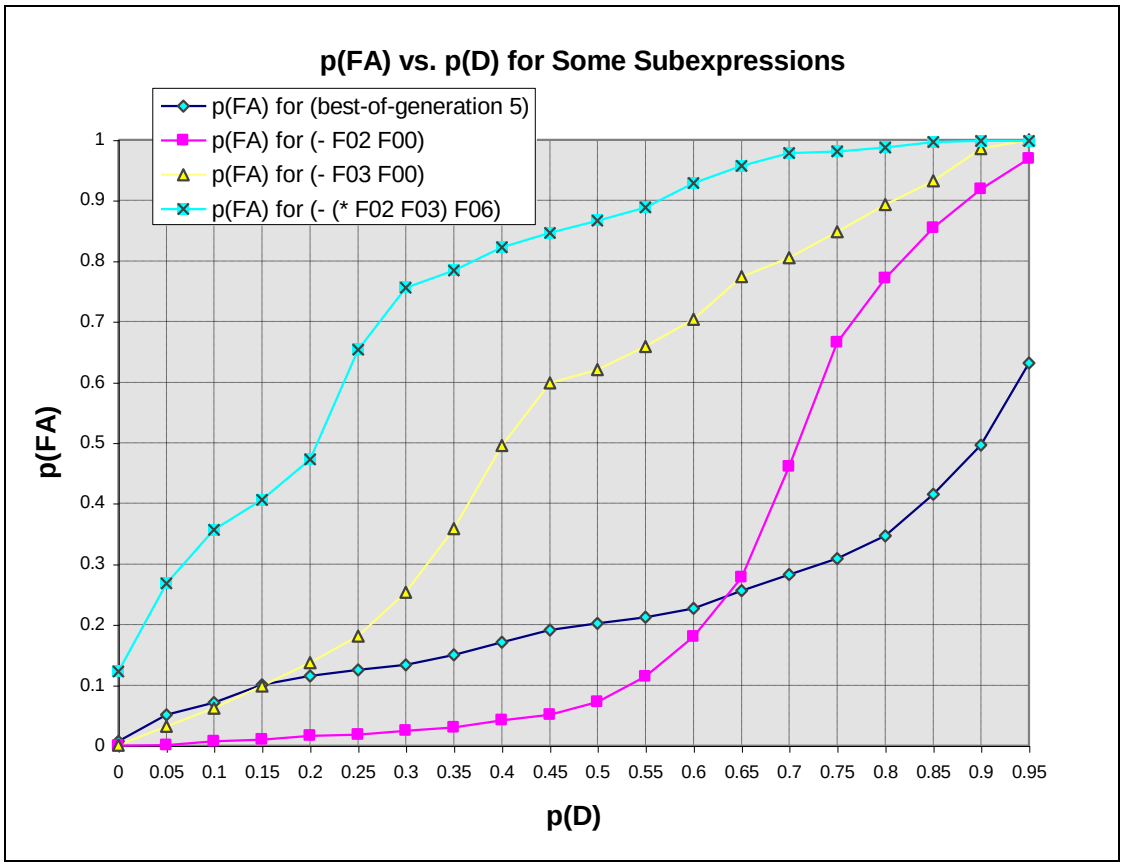


Figure 3.6: Performance of individual subexpressions and the program which contains them.

3.6. Discussion

3.6.1. Bias Sources and Statistical Validity of Experiments

It is usually not too hard to show one's favorite method to be better than others. For this reason we have gone to some length to describe the equivalence of effort put into the methods described (see Section 3.4.2). In the interest of fair comparison of results some further points must be discussed.

First, for both experiments the GP method uses fitness functions more sophisticated than the simple mean-squared-error minimization inherent to the Backpropagation algorithm. Indeed, the extra effort required to do so constitutes the only source of special "hand-tuning" that went into

either method. This cannot be avoided, since GP has no inherent measures of error or fitness: one must always create a mapping from algorithm performance ("phenotype") to a scalar fitness measure. This may be viewed as an asset when one considers that specialized measures of performance such as those used here for GP cannot be easily expressed in terms of the generalized delta rule used for Backpropagation.

In order to achieve a reasonable data generation and reduction task for the given time and resources, only twelve runs each were made for GP and Backpropagation. Of these, only four were performed at each C/T ratio. It would be desirable to create a large number of runs at optimal C/T values with different random seeds in order to obtain statistically significant mean and standard deviation values for performance. Despite the lack of such extensive statistics, the fact that for two separate classification tasks GP achieved a significant performance increase and greatly reduced computational complexity is a forceful argument in favor of the power of *structural adaptation*. Moreover, recent work (Carmi 1994) has put extensive effort into training this same neural network system against the induction problem described in Chapter 4 for purposes of comparison, and the advantage of the GP method is even greater in that problem than in the results shown here.

3.6.2. Primitive Features and Improved Performance

One significant observation is that the performance obtained in Experiment 2 is actually better than that of Experiment 1: for a p(D) of 75%, the GP tree of Experiment 2 obtained 30% p(FA) while the Backpropagation net obtained 27%. This is unlikely to be due to coincidence since both methods showed improved performance. Why else, then? One hypothesis is that both of these powerful nonlinear adaptive methods may be discovering inherent features which are better suited to the data than human-synthesized features based upon measurements (incorrectly) presupposed to be invariant. We may also speculate that the segmentation process introduces artifacts and variations which do not exist in the raw data. Other factors must be considered: we have mentioned that the image resolution is reduced prior to feature extraction, compressing groups of 3x3 (i.e., nine) pixels into one. This reduction in bandwidth may be harmful. Finally, the reduction in the number of input features itself may exponentially reduce the complexity of problem space to be searched, making relatively good solutions easier to find. Regardless of the cause we may conclude that this appears to be a result of direct benefit to the

development of future systems. It also suggests that applying GP to raw pixel imagery is a promising area for follow-on research.

3.6.3. GP Offers Problem Insights

We have seen that in the case of experiment 2, the structure of the solution tree offers clues as to which features assist in pattern discrimination. Although it is more complex, the same analysis can in principle be applied to Experiment 1. This indicates that GP can provide insights into what aspects of the problem are important: we can understand the problem better by examining the solutions that GP discovers. Chapter 7 describes methods by which salient problem elements and relations may be extracted from the population.

3.6.4. A GP Schema Theory?

Upon examination, we see that there are many repeated structures in the tree of experiment 1 as well as that of experiment 2. By probability, these are *not* identical structures randomly generated at the outset, but rather successful subtrees, or *building blocks*, which were frequently duplicated and spread throughout the population at an exponential rate due to their contribution to the fitness of individuals containing them. These suggest that a schema theory may be developed for Genetic Programming analogous to that for binary GA originally proposed in (Holland 1992). The relationship between schema fitness and frequency of occurrence is explicitly studied in Chapter 7.

3.6.5. Parsimony

Parsimony, or simplicity of solution structures, has previously been shown in (Koza 1992) to be achievable by adding the tree size (expression length) as a component of the fitness function to be minimized. In both experiments it was observed that a high degree of correlation exists between tree size and performance: among the set of "pretty good" solution trees, those with highest performance were usually the smallest within that set. It was also observed that most runs achieved a point where the size and complexity of trees eventually began to grow increasingly larger, while performance tapered off to lower values. Although many researchers have postulated that the size of expressions may grow as a result of selection pressure (Singleton 1993; Altenberg 1993), the results presented in this chapter run counter to those ideas. Furthermore, during the experiments described in Chapter 6 an experimental control was performed in which selection was random rather than fitness-driven. In those experiments it

was observed that expressions grew at a greater rate in the absence of selection. This led to speculation that that fitness and selection may act to restrict program size, but that there are forces most likely related to recombination and genetic drift which cause programs to grow despite selection pressure. Chapter 7 presents some evidence that sheds further light on this "size problem."

3.6.6. Your Mileage May Vary

The performance figures here are for comparison only. Specifically, the reader should be cautioned that the individual solutions reported on for GP, Backpropagation, and decision tree methods were *chosen* because they achieved peak performance against a set of out-of-sample test data. An independent measure of the true performance for each method would require a third "validation" data set which has no influence upon the choice of solutions whose performance is reported.

3.6.7. Learning, CPU Time, and LISP

There is a significantly higher computational cost for the generation of GP solutions relative to that of the multilayer Perceptron architecture. A training run (one random seed, one C/T ratio) using the Backpropagation algorithm requires about 20 minutes of CPU time on a Sun SparcStation/2. By comparison, an equivalent training run for GP using compiled LISP can take 40-50 hours of CPU time. This clearly has an impact on the ability to generate large or complex GP runs. Recently, we have created a new C-language implementation, GP/C (Genetic Programming in C). This code represents programs internally as parse trees in tokenized form. For backwards compatibility the trees may be read and written as LISP expressions. For the problem described in experiment 2 we have compared GP/C with equivalent LISP code. A typical run for GP/C requires 2 hours of CPU time for 60 generations and a population of 500, about 25x improvement in throughput relative to the LISP implementation. The 2 hours of CPU required by GP/C is still about 6x greater than is required by the Backpropagation algorithm for the same problem. This provides an interesting tradeoff of training vs. execution time, given the more efficient runtime performance achievable by the GP-generated solutions.

Chapter 4

The Effects of Selection Operators in the Context of a New Induction Problem

Alternative Methods of Artificial Selection and Their Impact on the Robustness and Generality of Genetically Induced Programs

> *Differentiation, due to the cumulative effects of accidents of sampling, may be expected in actual cases to be complicated by the effects of occasional long range dispersal, mutation, and selection but in combination with these it gives the foundation for much more significant evolutionary processes than these factors can provide separately.* Sewall Wright

> *No aphorism is more frequently repeated in connection with field trials, than that we must ask Nature few questions or, ideally, one question, at a time. This writer is convinced that this view is wholly mistaken. Nature, he suggests, will best respond to a logical and carefully thought out questionnaire; indeed, if we ask her a single question, she will often refuse to answer until some other topic has been discussed.* Ronald A. Fisher

The "Donut induction problem" requires separating two toroidal distributions (classes) which are interlocked like links in a chain. The cross-section of each toroid is Gaussian distributed with standard deviation σ. This problem possesses a variety of pathological traits: the mean of each distribution, for example, lies in the densest point of the other.

The difficulty of this problem is scaleable in two different ways: (1) *Overlap*: as we increase the standard deviation, points from the two distributions "intermingle" so that a classifier which optimally minimizes misclassification will still make mistakes in classifying outlying points; (2) *Generalization*: by varying the size of the training set, we control the degree to which it is statistically under-representative. In testing generalization we examine the effects that occur when training data are uniformly sparse as well as when the training set has spatially nonuniform "bites" taken out of it. Although there is no perfect solution for this problem we formulate the optimal Bayesian solution which minimizes error, and use it as the basis for comparison.

We observe the effects of different selection policies upon performance under varying conditions, including panmictic versus distributed populations ("demes") with and without steady-state replacement. For distributed populations, we investigate different sizes and configurations for breeding units. In all, over 2,000 runs are performed, each with a population of 1,024 individuals, over a period of 50 generations.

4.1. Introduction

In order for Genetic Programming (GP) to be useful as an engineering tool it must work on difficult problems. How does performance of GP change as the problem is made more difficult, or easier? It is hard to answer these questions when there is no control over the data: indeed, it is important to understand that the notion of difficulty is problem-specific. In the research described here we formulate a problem so that the factors that make it hard are quantifiable and therefore scaleable. Koza (1992) demonstrates the breadth of applicability of Genetic Programming against many problems which have realizable perfect solutions. This is of value in understanding the ability of GP methods to find optimal programs, but invites criticism that knowledge of the solution may bias the choice of function and variable sets towards those which make solutions trivial.

Chapter 3 concerned a difficult classification problem involving real-world data where the solution is unknown and only an elementary set of functions and variables are used. There, comparison of performance to existing methods indicated that Genetic Programming indeed offers a variety of advantages. This is a promising result, but is it representative? Unfortunately, the elements which make that problem attractive- e.g., significant noise, class overlap, and unknown distributions of classes, also make it hard to analyze.

In this work we set out to construct a problem with elements that make it difficult in ways that are representative of problems found in the real world. Difficulty is quantified and parameterized so that we may adjust the scale of difficulty and analyze how the quality of solutions generated by GP changes in response. Selection policies are studied in this context, particularly the distributed evolution scheme, due to Sewall Wright, as well as the steady state population model (Syswerda 1989; Whitley 1989).

4.2. The Donut Problem

The Donut problem uses artificially generated data that is purposely formulated to be both difficult and scaleable. It requires separating two classes, each toroidal in shape, which are interlocked like links in a chain. For each class, sample coordinates are generated by choosing a point ***p*** from a uniform distribution lying on a circle of radius 1.0. Next a plane is constructed which passes through ***p*** and through the circle's center, perpendicularly bisecting the circle. The final coordinates are then generated by choosing a point that lies in this plane, whose distance from ***p*** is Gaussian distributed with standard deviation σ. Therefore each distribution is a toroidal "cloud" of points with a major radius 1.0, and a cross-section minor radius proportional to σ.

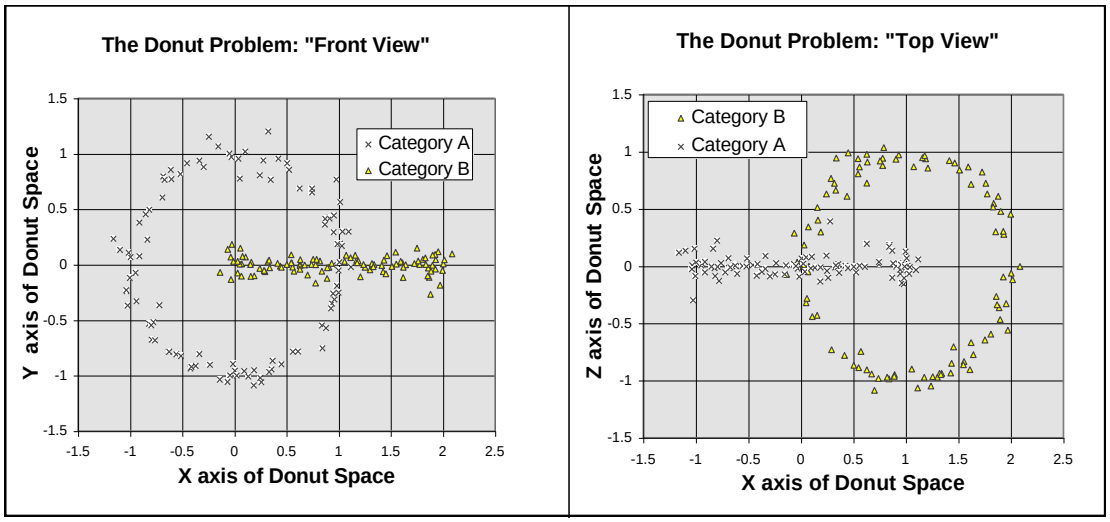


Figure 4.1: Front view and top view of two interlocking toroidal distributions (points from one distribution are shown as circles, the other as squares). For the case shown here, each contains 100 points and has a cross-section that is Gaussian distributed with a standard deviation of 0.1. By increasing the standard deviation we may increase interclass ambiguity to an arbitrary degree.

We designate the two distributions "class A" and "class B." The circular center line of class A lies in the X-Y plane with its center at the origin, while that of class B lies in the X-Z plane centered a the coordinates (1, 0, 0). A front view and top view of this configuration are provided in Figure 4.1.

This problem possesses a variety of pathological traits: the distributions cannot be linearly separated by a Perceptron rule (Rosenblatt 1962, Minsky & Papert 1969), they cannot be covered by cones or hypercones (Fukushima 1989), nor can they be enclosed by a pair of radial

basis functions (Poggio & Girosi 1989). Moment statistics cannot adequately describe them: the mean of each distribution, for example, lies in the densest point of the other. Thus the geometric and topological properties of the problem space are an inherent source of difficulty in the Donut problem.

In real-world problems there is often a degree of ambiguity in class membership because features are unreliable, measurements are corrupted by noise, or because some members of one class truly fit in better with another. The Donut problem models this phenomenon in a scaleable manner. As we increase the standard deviation σ, some points from the two distributions will intermingle more and more. Thus for sufficiently high values of σ, even a classifier that optimally minimizes misclassification will still make many mistakes in classifying outlying points. This is depicted in Figure 4.2.

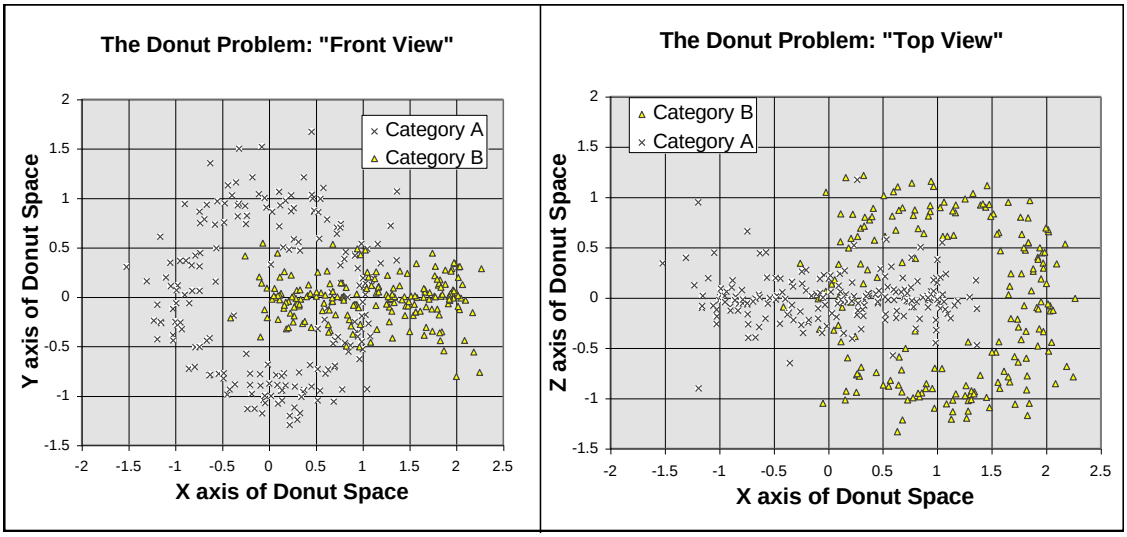


Figure 4.2: By increasing σ to 0.3 a significant degree of overlap develops between the classes.

Another major concern in real-world problems is generalization, which is the ability to deal with sparse and nonuniform statistical covering of sample space. As an example of this consider the case of automotive crash tests. These involve features that consist of factors such as speed, angle, and type of object hit. Not only are these tests expensive to perform and hence subject to small sample size, but experiments are biased towards conditions that have features that conform to particular federal standards (e.g., head-on with a brick wall at 15 mph). It is of

concern to the auto maker how these results extrapolate to crash features that are more commonly found in everyday traffic.

Generalization can be made difficult in two different ways. First, we can provide a sparse set of samples as a training set: the points provided are uniformly representative of the underlying distribution, but are statistically under representative. Second, we can provide data that are not uniformly representative of the underlying distribution (e.g., a training set that looks like the donuts have bites taken out of them). The latter case is shown in Figure 4.3.

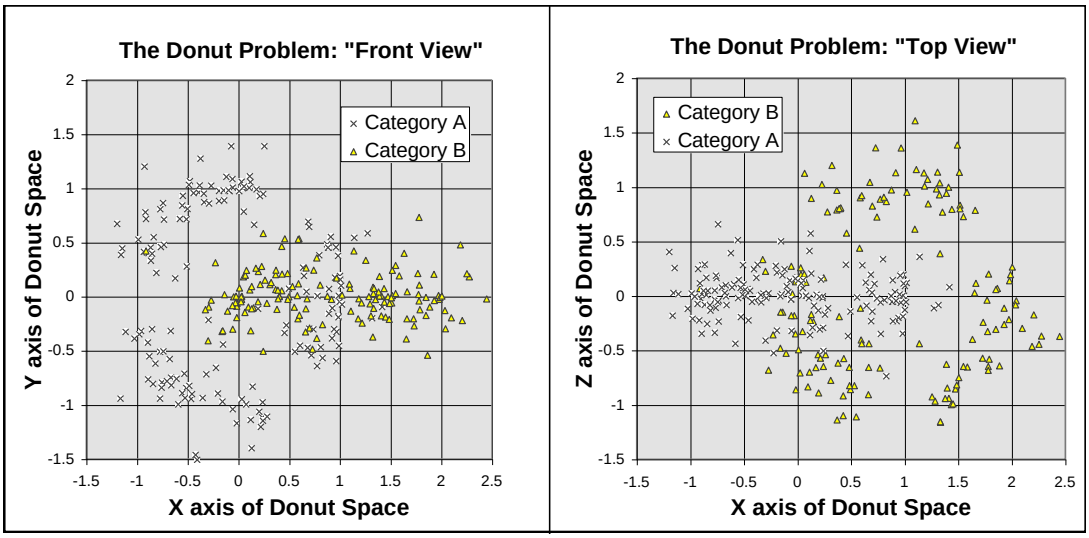


Figure 4.3: The Donut problem training set (σ=0.3) with 3 bites taken out of each class. How well can an evolved function f(x, y, z) which is trained against these data perform against an independent test set containing no gaps?

There is one other form of generalization that is desirable to test as well: in the real world the data available for training may occur in some idealized form that does not contain noise or distortions. Such factors may manifest themselves as class overlap. Therefore we wish to test the ability of GP to generalize with respect to class overlap: for example, how well does a classifier that was trained against the data of Figure 4.1 perform when tested against the data shown in Figure 4.2, and vice versa? This question may be answered using relatively little extra computation while performing the investigations outlined above.

Table 4.1: Tableau for the Donut problem (After Koza 1992).

Objective:	**Evolve an expression that will return ≥ 0.0 if the point belongs to class A, and < 0.0 if the point belongs to class B.**
Terminal set:	**X, Y, Z, Random constant**
Function set:	**+, -, *, %, IFLTE (or SQRT for some experiments)**

Fitness cases:	**A set of points and the class to which they belong, the set size varies with the experiments.**
Raw fitness:	**Number of correct classifications.**
Standardized fitness:	**1 - (Raw Fitness / Number of Fitness Cases)**
Hits:	**Not used.**
Wrapper:	**Not used.**
Parameters:	**Population size (M) = 1,024, Maximum generations (G) = 50.**
Success predicate:	**When one individual has standardized fitness = 0.**

We require Genetic Programming to induce solutions (classifying functions) f, where $f(x, y, z) \geq 0.0$ if the point belongs to class A, and $f(x, y, z) < 0.0$ if the point belongs class B. Fitness is based on minimizing the percentage of samples that are incorrectly classified, which is a form of "smaller-is-better" fitness as defined in Section 2.1.1.2. The function set used throughout the primary experiments is the same as the one used in Chapter 3. It consists of the primitive functions {+, -, *, %, IFLTE}, where % is protected division, IFLTE is if-less-then-else. The terminal set consists of {X, Y, Z, RANFLOAT}, where RANFLOAT is a random floating point constant: whenever this terminal is chosen for insertion into a randomly generated tree, it is replaced by a real-valued constant drawn from a uniform random distribution on the interval [-10.0... 10.0]. Table 4.1 summarizes the key features of the Donut problem.

4.2.1. Purposely Introduced Imperfections

We may expect to see some functions which achieve lower error (greater fitness) against the in-sample training data than is predicted by the optimal Bayesian solution. This is a clear indication of overfitting, The same functions will not do as well against an independent out-of-sample test set. Thus there is some imperfection in the fitness function itself, which is based strictly upon misclassification of the training set, since it may assign higher fitness to some functions than it would to the optimal one. This is in line with what happens in the real world, where information about the optimal solution cannot be used. Similarly, we note that the function set we have provided does not include the square-root function, which is an element of the optimal solution. As an experimental control, we run some tests using the square-root function to determine if it yields significant improvement.

4.2.2. There is a Solution (Sort of)

Given the description of the two distributions we can define the Bayesian decision surface at which a point is equally likely to have come from either distribution. This surface can be computed by measuring distance to the expected value of each distribution. For class A, the expected value of a point lies on the circle $\{(X^2+Y^2 = 1), Z = 0\}$; for class B it lies on the circle $\{((X-1)^2+Z^2 = 1), Y = 0\}$. Given an unlabeled point (X, Y, Z), it is most likely to have come from the class whose locus of expected values lies nearer: thus the best guess is to compute which circle lies nearer. Actually, this best guess is an approximation to the true likelihood, and is strictly true only for $\sigma = 0$. For values of σ less than or equal to 0.5, the range used for our experiments, it is accurate to small fractions of a percent. The distance from any point to the circle μ_A, $(X^2+Y^2 = 1)$, upon which lies the expected value of class A is given by:

4.1 $$\sqrt{X^2 + Y^2 + Z^2 + 1 - 2\sqrt{X^2 + Y^2}}$$

and the distance to the circle μ_B, $((X-1)^2+Z^2 = 1)$, of class B is:

4.2 $$\sqrt{(X-1)^2 + Y^2 + Z^2 + 1 - 2\sqrt{(X-1)^2 + Z^2}}$$

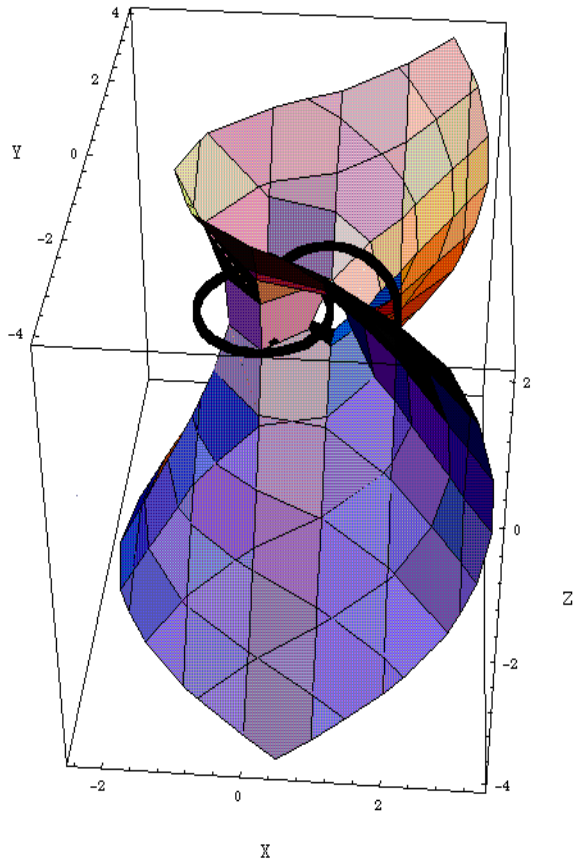


Figure 4.4: The (approximately) optimal solution to the Donut Problem is the locus of points for which Equation (4.3) equals 0: points falling on one side of this surface belong to Class A, while those falling upon the other side belong to Class B.

By subtracting the distance to μ_A from the distance to μ_B we obtain a function that produces a value greater than 0 when the sample is most likely to have come from class A and less than 0 when the sample is most likely to have come from class B. A simpler but equivalent function is obtained by first squaring the distance formulae prior to subtraction and canceling terms:

4.3 $$2\sqrt{X^2 + Y^2} - 2\sqrt{(X-1)^2 + Z^2} - 2X + 1$$

Although no formula can predict the class of an unlabeled point with certainty, this one will do so with the greatest likelihood of being correct. This is true regardless of the value of the point-spread parameter σ, which governs class overlap. The surface defined by setting Equation 4.3 to zero is plotted in Figure 4.4.

4.3. Selection Methods

The method used in (Koza 1992) for selection of mates from the population is similar to that originally proposed in Holland (1975, 1992), with the notable exception that the latter only selects one of the two parents according to fitness. Since then, other selection policies have become popular in the literature of artificial genetics. We introduce these models of population and selection to genetic programming in order to observe their benefits and setbacks.

4.3.1. Spatially Distributed Selection Among "Demes"

Sewall Wright (1889-1988) stands as one of the major figures of evolutionary biology with a career spanning more than 75 years (Provine 1986). Among his many contributions to the field, one of the most important (and controversial) is his "shifting balance theory" of genetic diversity (Wright 1931, 1978). In this theory, populations are spatially distributed throughout their environment into local breeding groups called "demes" (Gilmour and Gregor 1939) between which relatively little migration occurs. Within each deme, selection and genetic drift, (a phenomenon occurring in small populations by which traits may become widespread throughout although they are not particularly more fit than others), combine to cause rapid convergence to an evolutionary stable state. Separation by distance allows exploitation at a local level while individual demes explore separate evolutionary paths. Although it is a subject of controversy in the evolutionary biology community (Fisher 1958), such a spatial organization has proven advantageous in the context of artificial genetics, as demonstrated in (Collins 1992). There the author shows dramatic improvement of distributed over panmictic ("everybody-breeds-with-everybody") selection policies that are commonly used in GA and GP. The balance between exploration and exploitation can be demonstrated in computer simulations as well. This is graphically illustrated in Ackley (1993) and in Collins (1992), where it is shown that migration of individuals between demes causes particularly fit genotypes to spread slowly through the environment. In time, the environment becomes characterized by patches of successful strategies, genetically similar within each patch, but genetically different from other patches separated by boundary regions composed of less-fit hybrids.

4.3.2. Implementation of Distributed Selection

Our method for implementing distributed evolution divides the population evenly among a rectangular grid specified by the user. Thus a population of 2,048 individuals could for example

be allocated a 16x32-deme grid, creating a total of 512 demes, each populated locally by four individuals. Note that this sizing of deme is not particularly realistic in the biological sense, but rather is copied from distributed schemes of GA implemented on SIMD machines (Collins 1991, Ackley 1993). Mating is done via the K-tournament selection method described in Section 2.2.3.1. Empirically, K =6 works well under almost all circumstances, with a modification that accounts for locality of mating. In the scheme outlined by Koza, the individuals for the K-tournament are drawn from the entire population, whereas in the deme-based scheme they are drawn from spatially local sub-populations via a method we have termed "canonical" selection. To understand this, consider a deme located on element (M, N) of the grid. In canonical selection we fire a cannon sitting in the middle of the deme. Since each deme occupies a unit square, this means that the cannon is at location (M+0.5, N+0.5). The cannon is shot in a random direction (azimuth). The distance, which the shot travels, is zero-mean Gaussian distributed, with a standard deviation **D** provided by the user. A larger **D** will cause more shots to land in adjacent demes, on average, while a smaller **D** will cause most shots to land within the local deme (the grid is treated as a toroid for this purpose). An individual is randomly drawn from the local population of the deme in which the shot landed and is added to the local tournament list associated with deme (M, N). This process is repeated K times to form the entire list, and the most fit individual is selected to be a parent. A second parent is chosen by the same canonical-K-tournament process, mating takes place, and two children are inserted into the local new-population list associated with deme (M, N). Thus each deme maintains its own population and its own new-population list, but some of the parents of the new populations may be drawn from neighboring demes. The population of each deme is not replaced by the members of the new-population list until all demes in the grid have been processed.

4.3.3. Steady State Selection

The standard population model used in GA and GP may be considered a batch process in which a new population is created all at once: this is in contrast to most real populations in which births and deaths occur continuously at random intervals. In addition, once the children of the next generation have been bred the previous generation is discarded. Thus it is possible and perhaps even likely that the most fit individuals will not survive into the next generation, although much of their genetic material will be retained. Syswerda attempted to rectify these properties by

proposing steady state genetic algorithms (Syswerda 1989). Around the same time, Whitley proposed the Genitor algorithm which incorporates a similar scheme (Whitley 1989). The idea behind the steady state model is to perform individual selection and mating as previously described, but with the modification that the two newly generated offspring replace two less successful individuals in the existing population. This conserves the total number of individuals while allowing a single population to continuously evolve as a whole. There are many ways in which to choose "who dies." We employ the K-tournament method previously described to select the two parents but in addition conduct a K-tournament which selects the worst two of K individuals as candidates for replacement.

The steady-state method confers immortality upon the best individual in the population: it can never be selected for replacement until a better individual comes along. More generally this gives genotypes varying life spans based upon their fitness relative to others in the population. The fact that the best individuals are preserved in the population gives rise to another term for this family of strategies: "elitism."

4.3.4. Implementation of Steady-State Selection

In our implementation, steady-state populations require a tournament selection method as described in Section 2.2.3.1. It may be used in conjunction with distributed populations, or may be used alone with a panmictic selection policy. Since individuals are independently drawn to form the two K-tournament lists we check and retry-on-failure to ensure that they do not share the same worst member, since that would lead to an attempt to replace a single worst individual with the two new offspring. When this steady state model is applied to distributed populations, there is no new-population list maintained for each deme: offspring produced via crossover are inserted directly into the local or nearby population. Thus, when we perform mating at deme (M, N) it is possible to draw two parents from adjacent demes, breed them, and place their offspring in some other adjacent demes without any change taking place to the population of deme(M, N), which serves only as a sort of cheap motel.

4.4. Selection as Search

Chapter 2 provided an analysis of Genetic Programming in terms of AI search. That analysis is continued here in order to better understand the selection operators introduced in this chapter.

4.4.1. Spatially Distributed Selection

Conceptually, distributed methods of selection attempt to maintain separation between the regions of state space being searched. This is imposed by restricting selection of individuals through physical separation. Recall that Chapter 2 modeled selection, which is the process of choosing states in order to generate their successors, via the formation of an "interim queue" in which the relative frequency of states is proportional to fitness-ranked position in the original queue. The underlying assumption in terms of beam search is that with a single queue containing all population members, a highly fit individual can rapidly generate successors until the queue is filled with only that individual and its ancestors. The search converges to a single branch of the tree and all alternatives are discarded. Distributed selection attempts to remedy this by breaking the queue into many smaller queues and severely restricting the sharing of states between queues. The population of P individuals (which are the *states* visited by the search) is uniformly distributed on a toroidal grid of size MxN, meaning that each sector on the grid is home to P/(MN) individuals. In terms of the beam search outlined in Chapter 2 it can be said that each sector maintains a local queue of size P/(MN) states. For distributed selection this concept must be modified to account for *migration*. Migration rate governs the probability that members of the interim queue for a sector on the grid are drawn from the population queue of a neighboring sector. The likelihood that states from a neighboring sector will be selected into the local interim queue is proportional to their fitness and inversely proportional to the Euclidean distance from that sector. The migration rate controls how much isolation there is between regions of state space searched on the grid: in the upper limit, infinite migration allows selection from any sector with equal probability, resulting in a standard Genetic Programming selection with population size P. The case where migration is zero is the same as carrying out MN totally separate genetic searches in parallel, each with population size P/(MN).

4.4.2. Steady State Selection

Steady state selection has been described here as a method in which there are no discrete generations. Instead, as new individuals are generated (by recombination, cloning, or mutation) other individuals are selected for replacement based on their low fitness. An alternative view is provided by (DeJong 1993), who points out that steady state selection can be thought of as proceeding by generations in which only one or two (depending on the operator) members differ

from one generation to the next. Many such generations must be carried out to achieve the same number of successors that would be visited in a single generation under standard Genetic Programming. In terms of search, one or two successors are generated, replacing one or two states which are removed from the queue. Although it eschews the "batch generation" of successors, this procedure still does not adhere faithfully to the priority queue model of beam search since individuals are not necessarily added to and dropped from the ends of the queue.

Unlike standard "generational" selection, the steady-state method immediately replaces members of the search queue. This has a subtle effect which causes it to be greedier. Under both selection methods there is some algorithm which is used to choose individuals, for example the K-tournament method used in this work. In a population of size P, generational selection draws P individuals *from a queue which does not change in P trials*. Therefore the probability that an individual *i* will be drawn remains fixed throughout the P selection trials and the chance that it will be eliminated from the selection process during P trials is zero. In each trial due to steady state selection some state must also be chosen for replacement in the queue, and for any state *i* the chance of replacement is non-zero. In most implementations the replacement algorithm is similar to the selection algorithm but with a choice based on low fitness rather than high. For those individuals nearer to the tail of the queue (whose chances of replacement are greater than average) this chance of elimination is severely compounded over P trials. The elimination process is accelerated by the fact that if the average child state has a fitness value equal to or better than the mean of the current queue, then members of the queue nearer the tail will get moved even closer to the tail with each trial.

DeJong (1993) describes this in terms of "allele loss," meaning that the code comprising the "losers" is lost from the population. In terms of search, branches of the search tree with the best heuristic evaluation will rapidly be explored at the cost of abandoning search in other areas. This is the opposite of the effect intended in the distributed selection scheme. For many problems, however, steady state may lead to a satisfactory goal more rapidly than other methods of selection.

4.5 Experimental Method

All experiments measure fitness as the total fraction of samples from both classes that are misclassified. Thus we are attempting to minimize the value of the fitness measure. We

examine how the fitness varies as the function of parameter and experimental configurations described in detail below.

4.5.1. Performance of GP as Class Overlap Increases

We have illustrated how to increase the probability of misclassification by increasing the parameter σ, thereby making the "donuts" fatter and increasing overlap between the two distributions. Three separate suites (a variety of training data sets and a single test set) of data are provided: one for donuts with σ=0.1, a second with σ=0.3, and a third with σ=0.5. The optimal minimum misclassifications for these data are 0.00%, 3.45%, and 13.75% respectively.

A major question which we seek to answer is whether the solutions generated by GP increase nonlinearly in their deviation from the optimal solution as σ is increased: does GP become confounded by noise in the form of class overlap?

4.5.2. Generalization and Uniform Undersampling of the Training Set

Generalization is tested by varying the number of samples in the training set for a fixed disjoint test set of 1,000 samples. As few as 40 samples and as many as 1,000 samples are used in training. We observe how misclassification varies as a function of the sparseness of the training set, and how this effect varies jointly with point spread.

4.5.3. Generalization and Nonuniform Sampling of the Training Set

Generalization is also tested by training on distributions that have "bites" taken out of the donuts: large contiguous regions are unpopulated by sample data points, as was illustrated in Figure 4.3. The nonuniformity is varied parametrically by the number of bites: up to three bites may be removed from each donut, each subtending about 30 degrees, or 8.3% of the sample space.

4.5.4. Assessing the Effects of Demes and Elitism

The Donut problem provides a benchmark for the effects of selection policies. In theory, distributed populations should promote exploration of solution space and prevent global convergence to local minima, while elitism accelerates exploitation of successful solutions and speeds convergence. Most problems used to study these selection policies have focused upon fitness relative to the original training set rather than measure performance against out-of-sample data (Goldberg & Deb 1991, Collins & Jefferson 1991, Ackley 1993). In this study we are primarily concerned with the impact of selection on generalization and overfitting of data. It

is important to understand how effects of selection vary jointly with class overlap and undersampling effects as described in previous sections.

Unlike steady-state elitist policy, distributed evolution has some parameters associated with it: recall that in "canonical" selection, the search for mates may examine an adjoining location whose distance is Gaussian distributed with standard deviation **D**. The value of **D** governs the rate at which genetic material may migrate across the landscape. This is roughly an embodiment of Wright's parameter for size of breeding unit. As **D** is decreased, genetic diversity increases with change in location. A second parameter is the geometric configuration of the landscape itself: in particular, we try two configurations: (1) a square toroidal grid of 16x16 demes, and (2) a "coastline" configuration, consisting of a 2x128 toroidal grid. The latter case should promote far greater genetic diversity than the square grid for a given value of **D**, since it provides much greater restrictions on travel between demes. Thus the choice between the square versus linear configurations is postulated as a choice between high mobility and low mobility, while the parameter **D** provides a fine adjustment.

4.5.5. Summary of Experimental Configurations

For each of the two deme configurations three values of **D** are used: 0.25, 0.5, and 0.75, where the distance between two adjacent grid elements is 1.0 units. This translates to hops that have a 2.8%, 26%, and 40% chance of landing outside the local deme, respectively. Three values of **D,** times two geometric layouts (square and coastline) makes a total of six distributed evolution types. In addition, we perform panmictic selection as an experimental control, making a total of seven landscape configurations. Each of these is used with and without steady-state selection policy, providing a total of 14 selection policy combinations. As stated in Section 4.4.1, there are three values of the overlap (a.k.a. point-spread) parameter σ tested. Each of these is used with four different sizes of uniformly sparse training data sets, and eight sets of nonuniformly sparse training data, totaling 3x(4+8) = 36 different data sets. For each combination of selection policy and training set, four independent runs are made using different initial conditions (random seeds). Thus, the total number of runs performed is 14x36x4 = 2,016 runs. Each has a population of 1,024 individuals reproducing over fifty generations, resulting in the analysis of a total of 103,219,200 individuals. It is important to note, however, that only four unique random seeds are used. This means that the four initial populations of 1,024 individuals used for each

experiment are re-used for each configuration, and thus the divergent courses of evolution are due to the changed factors such as deme configuration, migration rate, steady-state, etc., rather than differing initial conditions.

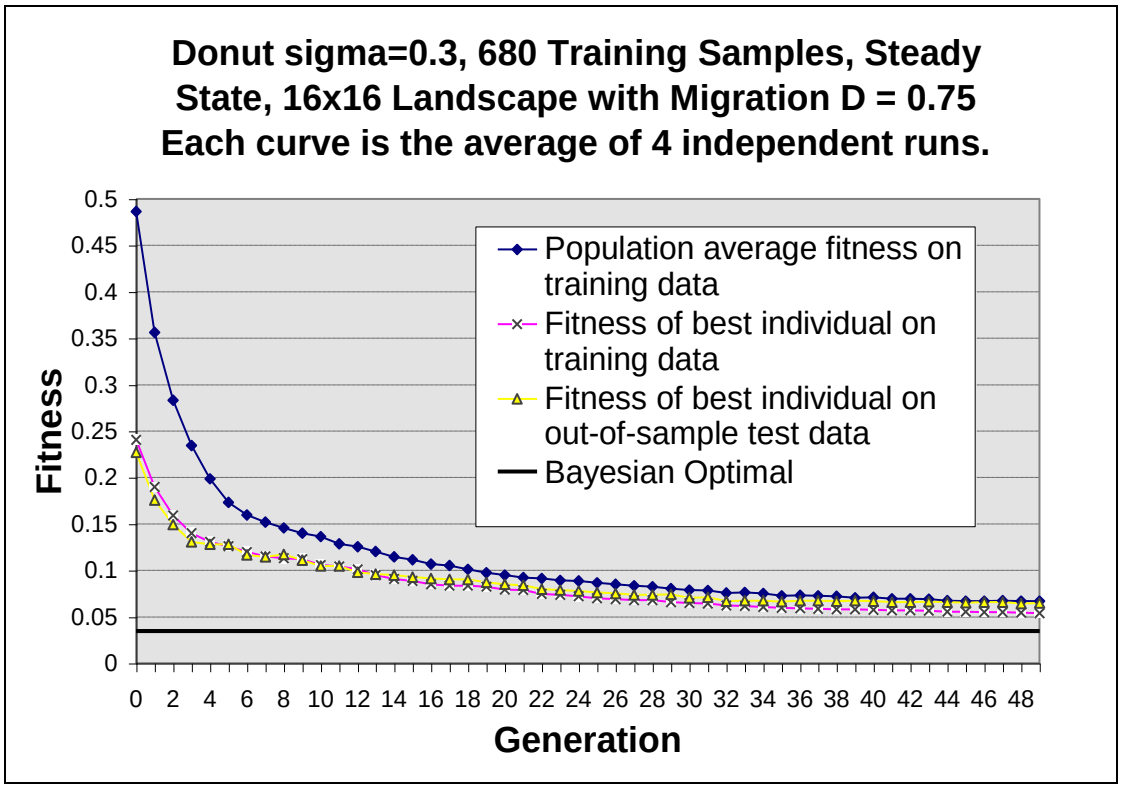


Figure 4.5: Fitness vs. Generation using steady-state population on a 16x16 grid of demes, high migration rate.

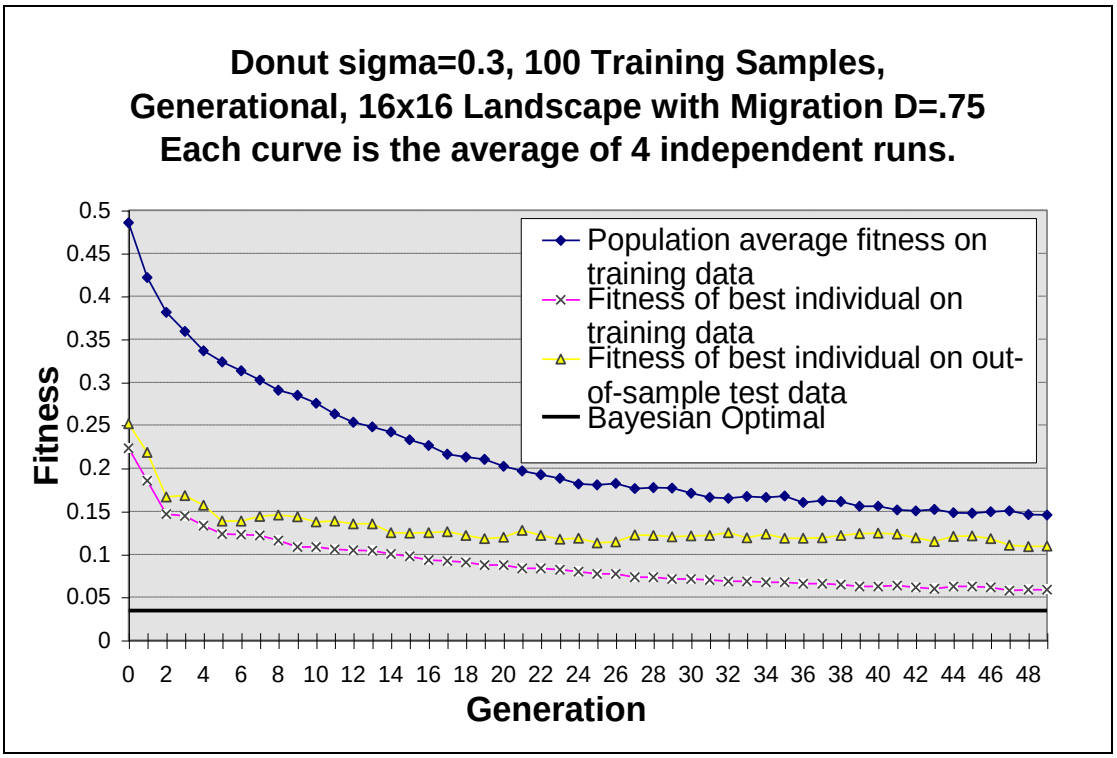


Figure 4.7: Fitness vs. Generation for a small number of training samples and a high degree of class overlap.

n addition, selected experiments are repeated using a function set which contains the SQRT function, in theory providing all mathematical tools necessary to achieve the optimal solution. Also, selected individuals, which are trained against data at a particular value of σ, are additionally evaluated against test data with a different σ value, to test the ability of GP to generalize with respect to class overlap.

4.6. Results

Figures 4.5, 4.6, and 4.7 show the raw data for a sampling of the 2,016 runs performed. Each plot contains a total of 3 curves, representing three figures of merit, plotted against time (in generations). Each curve is the average of four runs having identical parameters and different initial conditions (induced by different random seeds). The three figures of merit are: (1) average fitness over the entire population, as measured against the training data set (plotted with circles); (2) fitness of the individual that achieved highest fitness against the training set as measured against the training set, this is referred to by Koza as the "Best-of-Generation Fitness" (plotted with squares); (3) fitness of the individual that achieved highest fitness against

the training set as measured against the out-of-sample test set. The latter figure is the "Generalization Fitness" (plotted with diamonds). It is the true "figure-of-merit" for induction performance. These curves illustrate some general principles that characterize the various configurations and experimental parameters used.

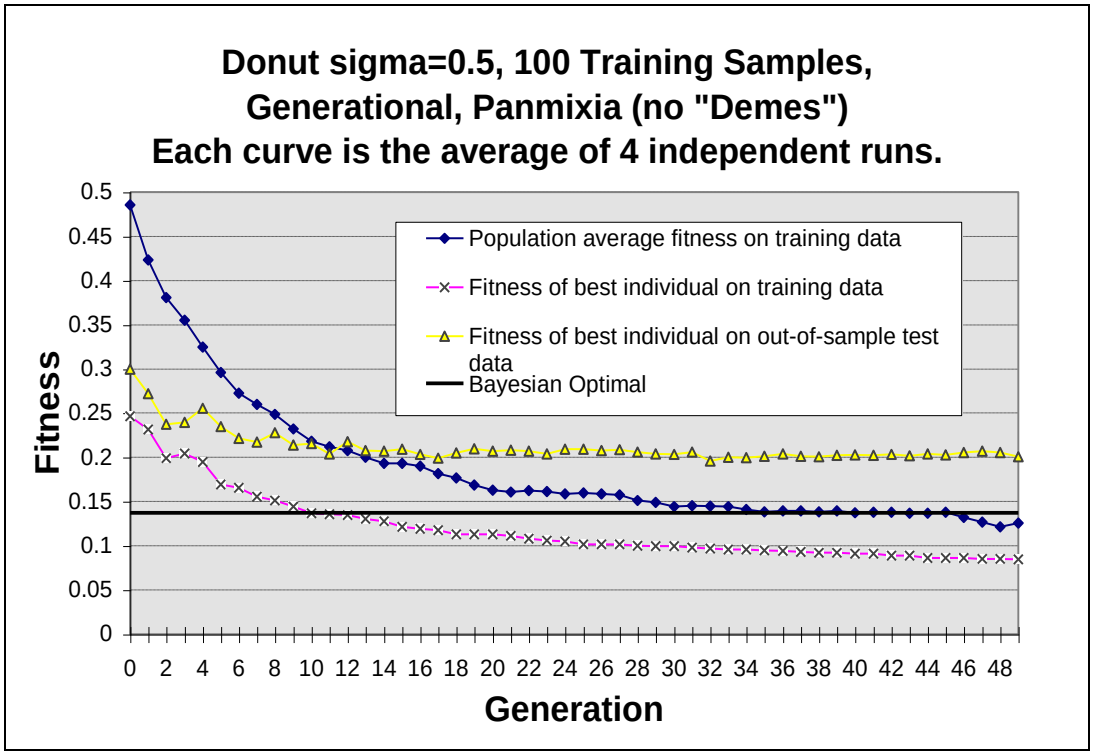


Figure 4.6: Fitness vs. Generations with a relatively small number of training samples.

Figure 4.5 depicts a run using donuts with σ=0.3 and a training set containing 680 samples (this can be considered a very statistically representative training set). It employs steady-state population and a 16x16 distributed population grid with migration parameter **D**=0.75. It possesses some characteristics typical of the steady-state model, notably rapid convergence and average population fitness that approaches that of the best individual fitness (compare to 4.6 and 4.7). It also shows good generalization, with fitness against the test set closely approximating that achieved against the training set: this is a typical consequence of the relatively large number of training samples used.

Figure 4.6 shows a run, also using donuts with σ=0.3, but with a much more sparse 100 sample training set. The batch-population update policy is used with a 128x2 coastline deme-grid

configuration, with **D**=0.75. A much slower convergence rate can be observed relative to the case shown in Figure 4.5, particularly in the average fitness, as is typical of distributed evolution with low mobility and no steady-state selection policy. Although fitness with respect to the training set is similar in Figures 4.5 and 4.6, the performance against out-of-sample data is worse (higher error) due to statistically under-representative training data.

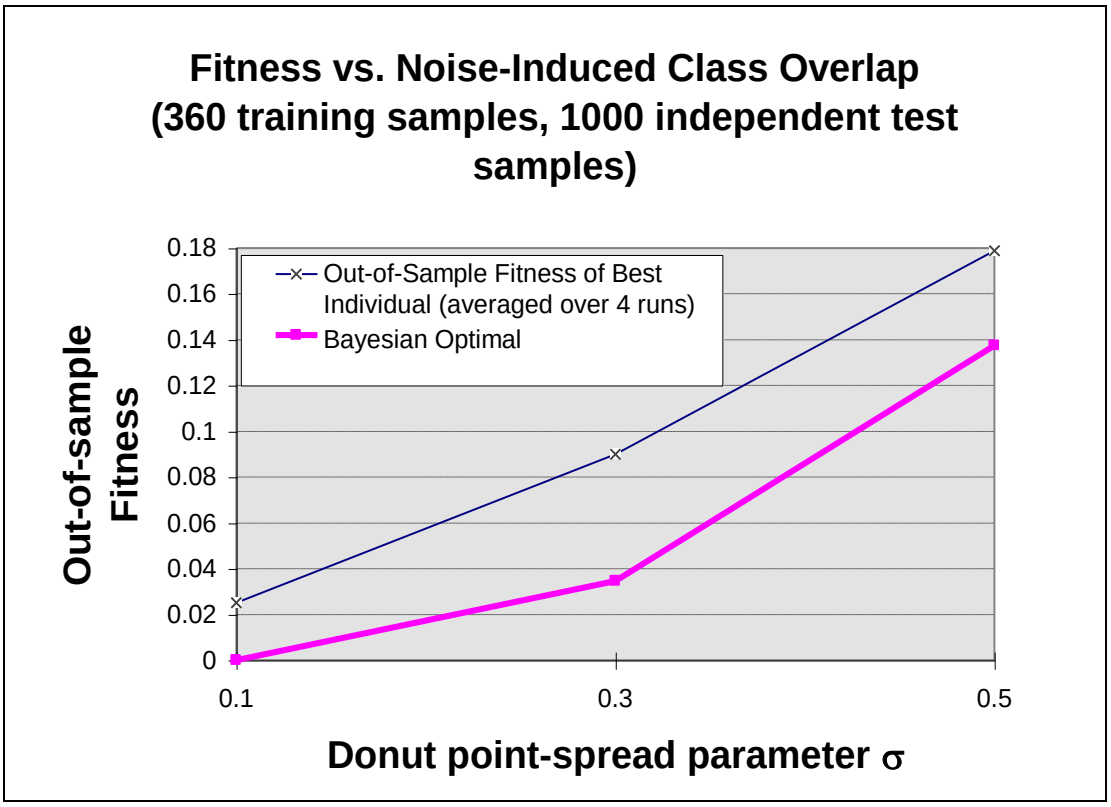


Figure 4.8: Comparison of best average vs. optimal fitness as a function of overlap between classes in training samples.

Figure 4.7 depicts another 100 sample sparse-training-set case, this time with σ=0.5. Panmictic populations and generational selection are used. Note that whereas the σ=0.3 cases depicted in 4.5 and 4.6 may approach a minimum misclassification of about 3.45%, the σ=0.5 case should in theory approach a 13.75% minimum. In practice, we observe *overfitting*, as predicted in Section 4.2.1: the training set fitness and average fitness figures drop well below the theoretical minimum, which indicates that the functions formed by GP have become adapted to the particular samples of the training set, rather than to the true underlying distributions. As

expected, fitness against the out-of-sample test set remains above the theoretical minimum having become fixed at about 20% misclassification around the 15th generation.

Having examined some of the qualitative properties of the experimental variables we now move on to describe specific measures of performance.

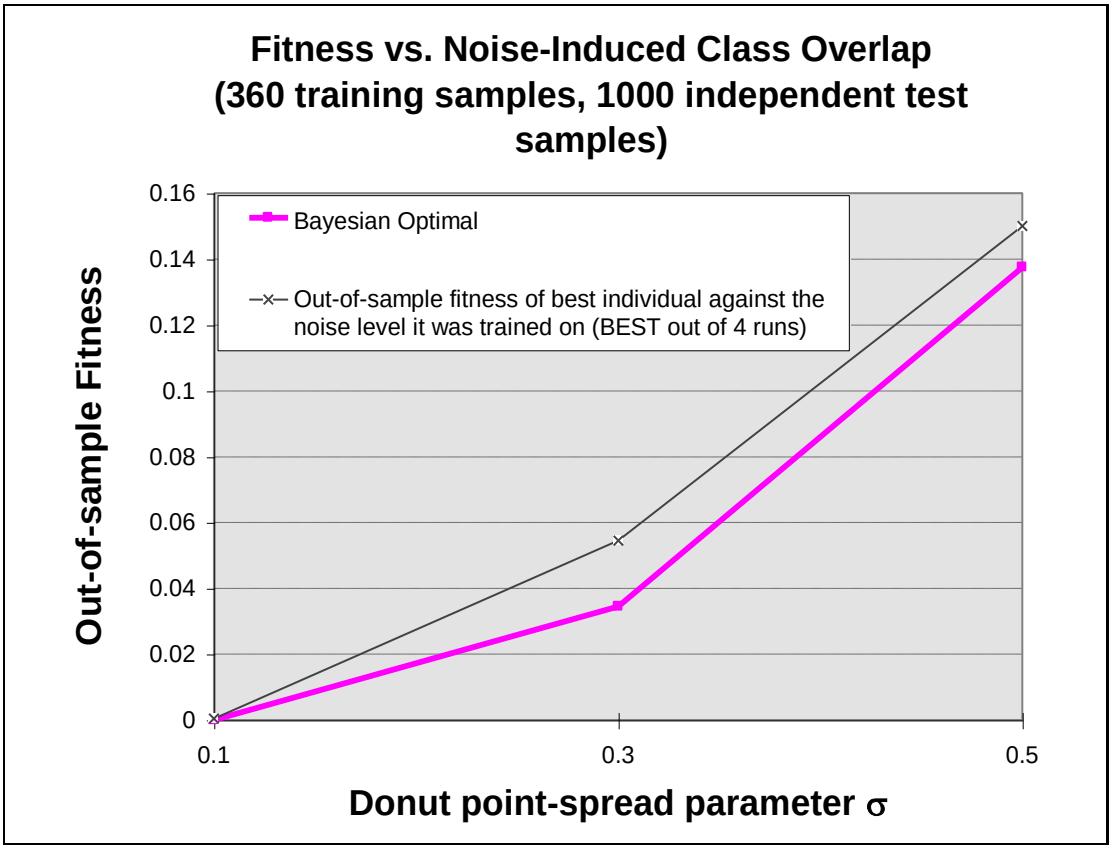


Figure 4.9: Generalization fitness of best individuals for each value of σ vs. optimal fitness.

4.6.1. Scalability With Respect to Class Overlap

In order to judge the effectiveness of genetic programming, it is important to understand whether the search procedure is confused by noise in the form of class overlap. Figures 4.8 and 4.9 indicate that, at least for the Donut problem, it is not. Figure 4.8 shows the generalization fitness averaged over all 14 selection policy configurations and all four initial conditions for the last 10 generations of each run, as a function of the class-overlap parameter σ (plotted with X's). The number of training set elements is fixed at 360.

On the same plot we show the minimum theoretical limit for misclassification as a function of σ (plotted with heavy line). It appears that on average, the fitness of functions generated by GP roughly deviates from the theoretical limit by a constant factor. This is further borne out in Figure 4.9, which depicts the performance of the three single best-test-set-fitness individuals for each value of σ, with training set size fixed at 360 samples.

In plot 4.9 the fitness of the individual at σ=0.1 is not a perfect 0.0, but rather misclassifies only a very few of the 1000 samples from the out-of-sample validation test set.

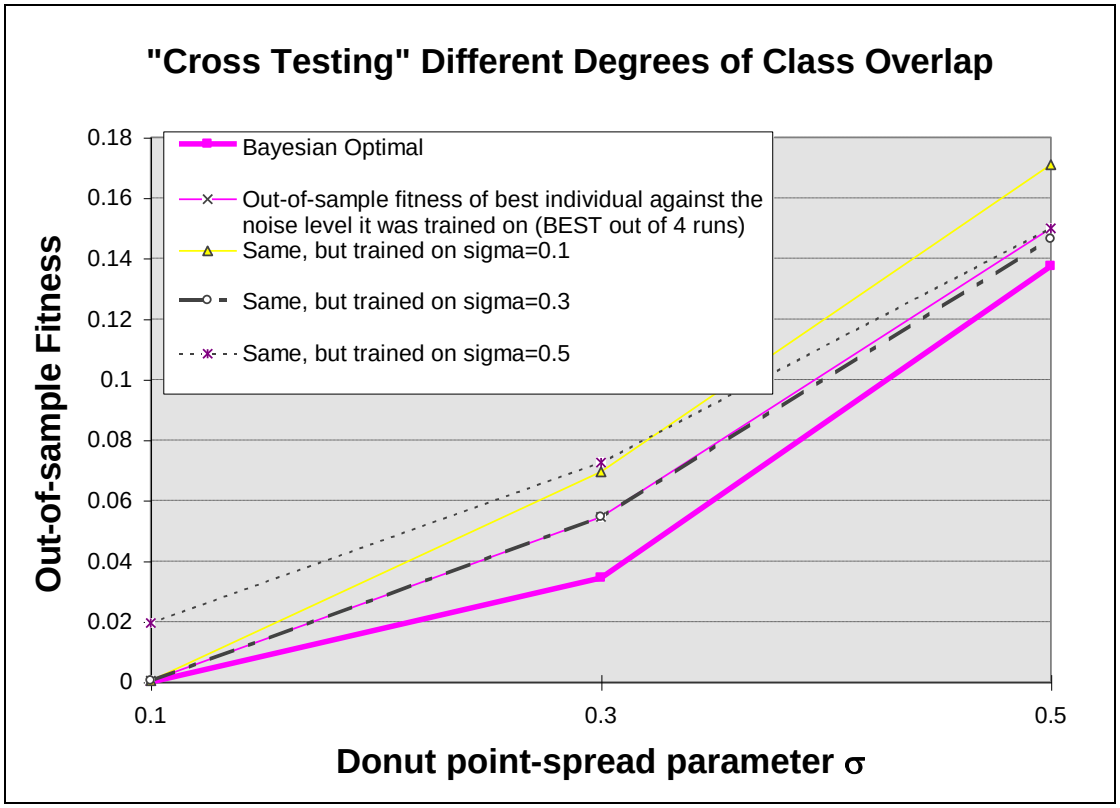


Figure 4.10: Fitness of best individuals trained with three different σvalues and tested with three σvalues.

4.6.2. Generalization With Respect to Class Overlap

In Section 4.2.2 we showed that the Donut problem has a single underlying solution whose difficulty can be parameterized with respect to ambiguity in the form of class overlap. In order to evaluate generalization with respect to the level of overlap and/or noise, it is not necessary to perform additional training, but rather to take the best individuals discussed in Section 4.5.1 and re-test them using each other's validation test set. This is illustrated in Figure 4.10. Four curves

are shown, which depict theoretical minimum misclassification (heavy black circles), and the performance of the best generalization fitness individual trained with (1) σ=0.1 (triangles) (2) σ=0.3 (x's) and (3) σ=0.5 (circles) as a function of the test-set σ. This yields an interesting result, namely that the individual, which was trained with σ=0.3, not only outperforms the other two individuals in classifying the distributions with which they were trained, but also arrives closer to optimal solution with the cross-tested distributions than it does against its own. The unusual point spread for these results may be due to statistical variations, since they test the performance of only three individuals, but again indicate the same constant error trend previously observed in section 4.5.1.

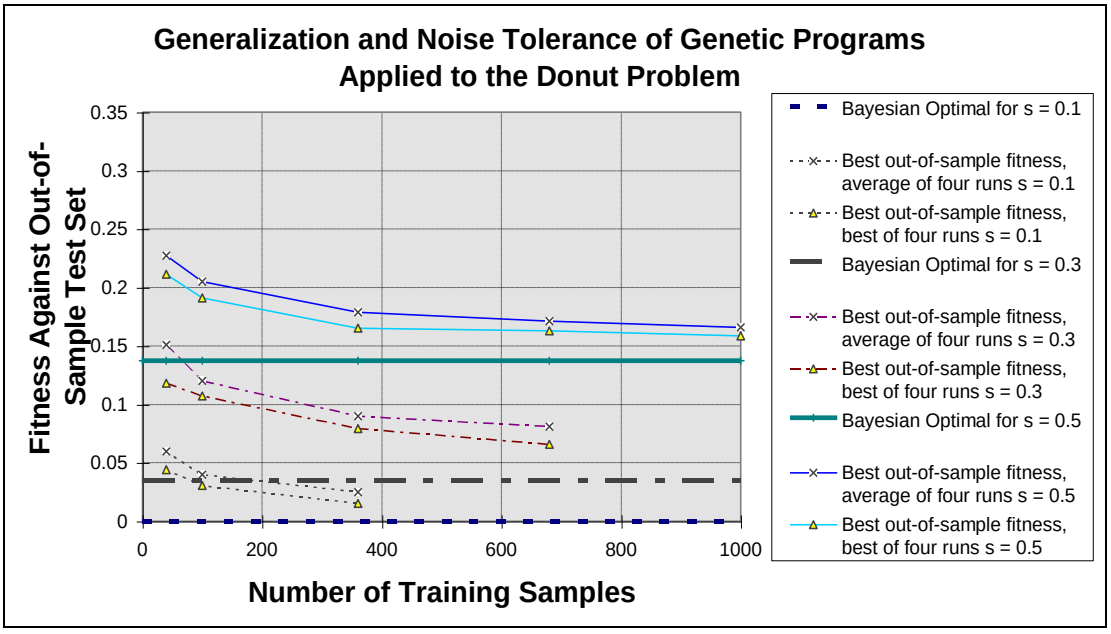


Figure 4.11: Performance vs. training set size for uniformly distributed training samples.

4.6.3. Generalization and Uniform Undersampling of Training Data

Figure 4.11 summarizes performance as a function of training set size. A total of six curves are presented: two each for σ=0.1 (circles), σ=0.3 (diamonds), and σ=0.5 (triangles). All of them depict the generalization fitness performance averaged over the last 10 generations. In each case, the upper curve of the pair depicts this figure averaged over all 14 selection-policy configurations, while the lower curve represents the performance of the single best of the 14. Theoretical minimum misclassification for σ=0.3 (3.45%) and for σ=0.5 (13.75%) are shown as

solid horizontal lines (the minimum for σ=0.1 is approximately 0.0%). To a first approximation, the rate at which fitness converges towards the optimum, as a function of training set size, is roughly the same for all 3 sets of curves. It is notable that on average, the difference between actual fitness and optimal fitness is significantly greater for the σ=0.3 case than for the others.

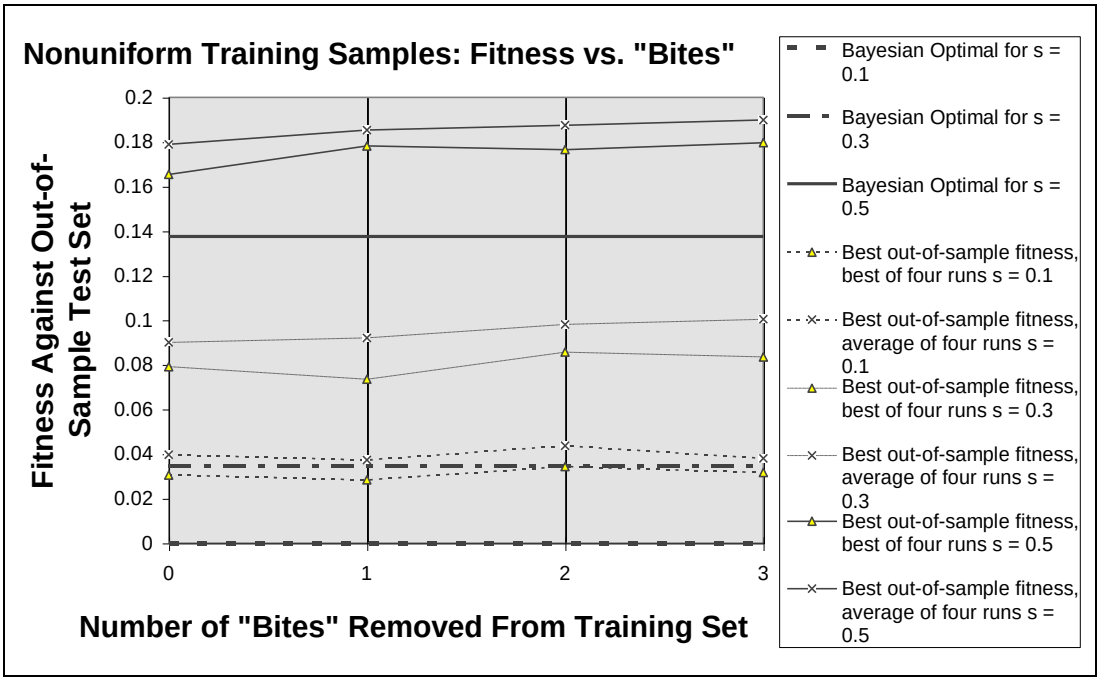


Figure 4.12: Fitness with respect to nonuniformly distributed training samples ("bites" taken from the donuts).

4.6.4. Generalization and Nonuniformly Distributed Training Data

Figure 4.12 shows performance figures in a manner similar to Figure 4.11, except that performance is plotted as a function of the number of "bites" removed from the training set (see also Figure 4.3). Note that unlike Figure 4.11, the number of points in the training set decreases to the right, since each bite removes about 8.3% of the training set samples. The curves for σ=0.3 and σ=0.5 use 360 samples in the full (no bites) training set, while the σ=0.1 set uses 100. Thus for the σ=0.3 and σ=0.5 cases, there are about 270 samples in the three-bites-removed training set. In both of those cases, the removal of the nonuniformly distributed point sets results in an average increase in misclassification of about 1%. This is comparable to the performance degradation that can be interpolated from the corresponding curves of Figure 4.11 for a training set of size 270. Results for the σ=0.1 case appear inconsistent since they fluctuate with

increasing σ, but both average and best results curves display a similar characteristic. We speculate that this behavior is an artifact of the relatively sparse training set used.

4.6.5. Comparative Performance and the Optimal Function Set

In Section 4.2.2, an approximation of the optimal Bayesian solution for the Donut problem was introduced, which involved the use of the square root function. The function set used in experiments up to this point has not included a square root operator. How much performance is being sacrificed in this problem by not including it? Figures 4.13 and 4.14 compare the performance of the standard function set used throughout the problem with one that replaces the IFLTE function with SQRT. The SQRT function that we provide differs from the standard formulation in that it is protected: we take the absolute value of its one argument prior to the square root computation. This is necessary since there is no direct control over the value of arguments, although Darwinian purists might wish to take the approach of letting negative arguments passed to the square root be lethal to the genotype.

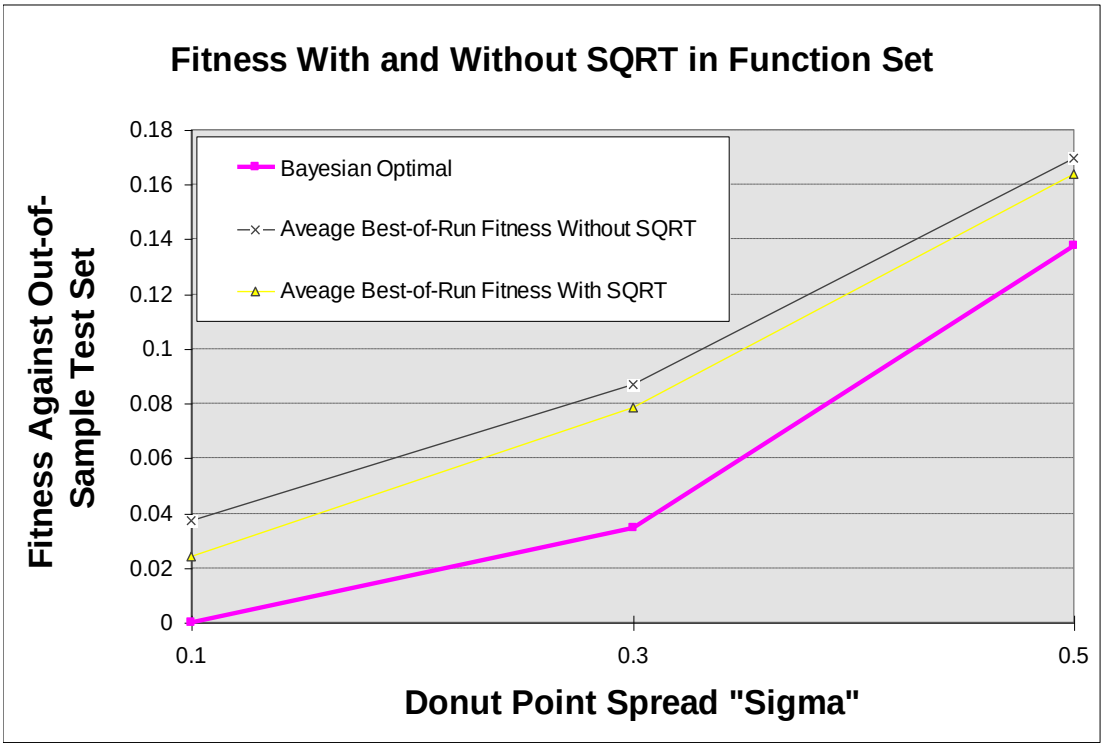


Figure 4.13: Average fitness of individuals with and without square-root included in the function set.

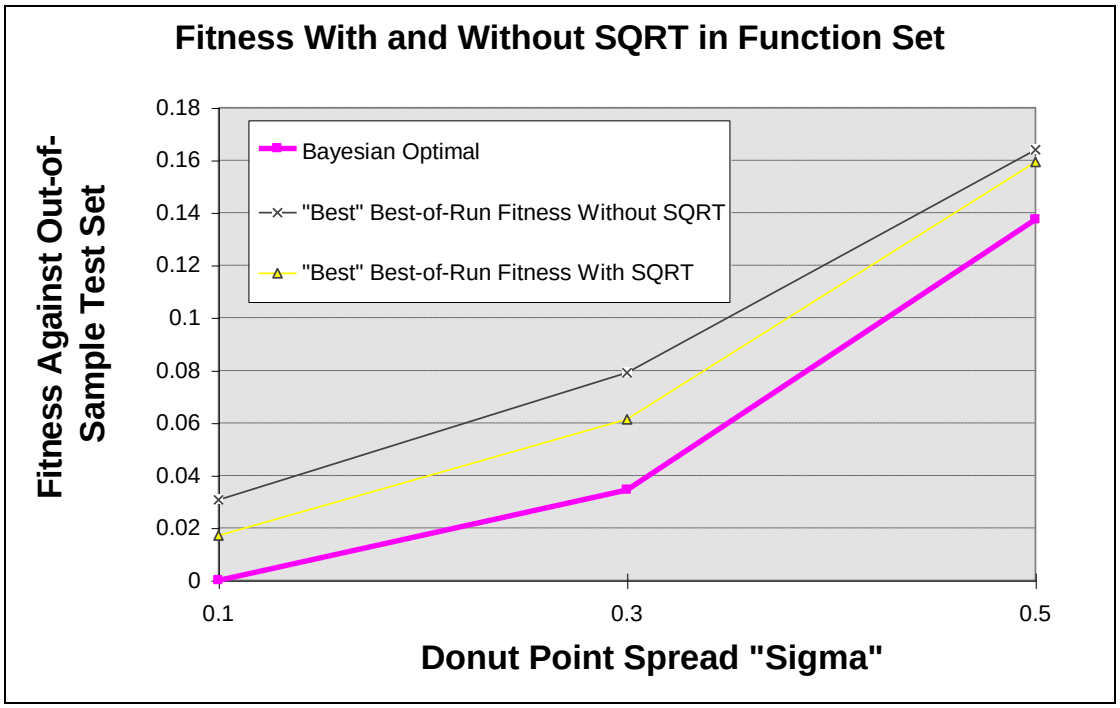


Figure 4.14: Fitness of the best individuals produced with and without the square-root function included in the function set.

Figure 4.13 shows the average of the generalization performance taken over all selection policy configurations and all random initial seeds for the last 10 generations. For this experiment, a training set size was used which increases with σ: 100 samples for σ=0.1, 360 samples for σ=0.3, and 680 samples for σ=0.5. Fitness (classification performance) using IFLTE is plotted with circles, while that using SQRT is plotted using squares. We see that there is indeed a consistent improvement achieved, although the overall error level is still above the optimal. A similar trend can be seen in Figure 4.14, which depicts the best-test-set performance, averaged over all four random seeds and the last 10 generations, for the single selection scheme having the best performance at each combination of σ, training set size, and function set. Again, even though overall performance is improved using SQRT, this single best individual does not achieve the optimal solution.

Table 4.2 The meaning of experiment labels in comparative performance diagrams

Configuration Label	Steady State Elitism	Migration Rate, D	Landscape Shape
Gen Sel, Coast, Migr=.25	No	0.25	128x2 Toroidal Grid
Gen Sel, Square, Migr=.25	No	0.25	16x16 Toroidal Grid

Gen Sel, Coast, Migr=.50	No	0.50	128x2 Toroidal Grid
Gen Sel, Square, Migr=.50	No	0.50	16x16 Toroidal Grid
Gen Sel, Coast, Migr=.75	No	0.75	128x2 Toroidal Grid
Gen Sel, Square, Migr=.75	No	0.75	16x16 Toroidal Grid
Gen Sel, Panmixia	No	N/A	Panmixia (No Demes)
SS Sel, Coast, Migr=.25	Yes	0.25	128x2 Toroidal Grid
SS Sel, Square, Migr=.25	Yes	0.25	16x16 Toroidal Grid
SS Sel, Coast, Migr=.50	Yes	0.50	128x2 Toroidal Grid
SS Sel, Square, Migr=.50	Yes	0.50	16x16 Toroidal Grid
SS Sel, Coast, Migr=.75	Yes	0.75	128x2 Toroidal Grid
SS Sel, Square, Migr=.75	Yes	0.75	16x16 Toroidal Grid
SS Sel, Panmixia	Yes	N/A	Panmixia (No Demes)

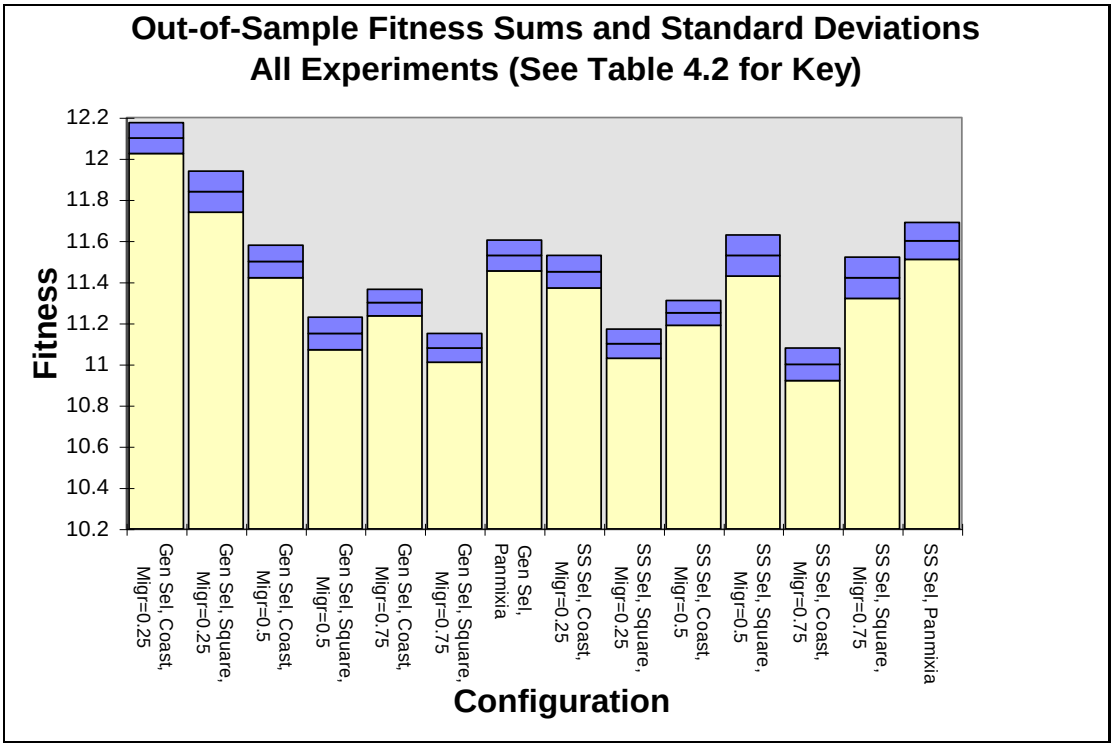


Figure 4.15: Sum of mean fitness, with respect to test set, for all selection policies across all experiments (std deviation shaded).

4.6.6. Comparative Performance of Selection Policies

In order to assess comparative performance between the 14 different selection policies and configurations used in this study, we must first develop a sound statistical basis. This is not as easy as it would seem on the surface: although there were over 2,000 runs performed, many of these had different expected outcomes due to various combinations of training set density, overlap, etc. We cannot simply average the scores together due to their multimodal manner of distribution. Likewise, we require a means by which to assess the variability of any composite performance figure, so that we may assign a reasonable confidence to it: a degree of uncertainty in our conclusions is acceptable as long as that uncertainty can be accurately bounded. Towards the goal of a sound statistical analysis we apply a methodology, which is derived from the analysis of variance (ANOVA), due in large part to the efforts of Ronald A. Fisher. It is an interesting historical note that R. A. Fisher the statistician is one and the same as R. A. Fisher the geneticist, with whom Sewall Wright had a lifelong disagreement over the nature of selection and distributed populations.

Our methodology divides the experiments into different partitions, which may be treated as independent analyses. Each partition consists of a group of experiments with a single common factor, e.g., all experiments with sparse training data. Within the partition, each of the 14 experimental configurations - i.e., various deme layouts and parameters in combination with or without steady state - is kept separate from the others. Perhaps the best analogy for this approach is that we consider each of the 14 configurations as a student in a class: the experiments in a given partition represent a series of tests administered to all students, where each exam may have a different number of points possible, and upon whose sum of scores the grade will be based (the student analogy breaks down a bit when we consider the lower-is-better scoring system, but it does not affect the nature of the analysis). In order to be fair in assigning grades, we associate a confidence, or standard deviation, with the score of each individual. Because of the fact that different experiments may have wildly different average outcomes, we cannot base estimates of mean and standard deviation upon the mean score of all exams. Instead, we base an estimate of variance upon the four runs consisting of different random seeds with otherwise identical experiments. The mean fitness of these four runs is first computed. The sum of the squared differences between this local mean and each of the four individual fitnesses is summed into the global variance estimate for the individual. The square

root of the resulting figure, taken over all experiments in the partition, is the estimate of the standard deviation of the individual's sum of scores from each experiment. It is important to note that the absolute sum of scores is a relatively meaningless measure since it is primarily a function of the number and type of experiments included in the partition. Instead, the true figure of merit is the ratio of the standard deviation for two individuals relative to the difference of their sum of scores. Together these figures tell us the likelihood of making a Type I or Type II error (Larsen & Marx 1981) when stating that one method works better than the other. In the following sections we use the above method to graphically depict the comparative performance for a variety of experiment partitions, with an emphasis on those we consider particularly hard.

Table 4.3: Experimental configurations whose results are summed to form the columns of Figure 4.15 and Figure 4.16.

Sigma	Training Set Size	Bites
0.1	40	0, 1, 2, 3
0.1	100	0, 1, 2, 3
0.1	360	0
0.3	40	0
0.3	100	0, 1, 2, 3
0.3	360	0, 1, 2, 3
0.5	100	0
0.5	360	0, 1, 2, 3
0.5	680	0, 1, 2, 3

4.6.7. Performance Across All Experiments

Figure 4.15 depicts a bar chart showing the performance of each configuration taken against all experiments performed. The line running horizontally, through the center of the dark region at the top of each bar, is the sum of best-10-average-test-fitness over all experiments performed for the particular configuration. Best-10-average-test-fitness is defined as the average of the out-of-sample fitness for the 10 best individuals produced during a run. Recall from the definition of generalization fitness that only the individuals which achieved best-of-generation against the training set were eligible for testing. The dark region at the top of each bar depicts the fitness sum plus and minus the estimated standard deviation. Table 4.2 provides a key to the labels in those figures, which are used to describe the experimental configurations in each chart. There are 27 unique experiments in all, each with four random seeds. Thus each bar is the sum

of fitness measures from 108 runs. In viewing these charts, it is important to remember that lower fitness is better.

The estimated standard deviations are small compared to fitness sums for Figure 4.15, indicating that the differences shown are significant in many cases. When steady state population is used, distributed evolution always does as well as or better than panmictic. Without the steady-state model, however, distributed evolution performs significantly better in three cases, and significantly worse in two, with one being about the same. The parameters which produced these results are summarized in Figure 4.15 and in Table 4.3.

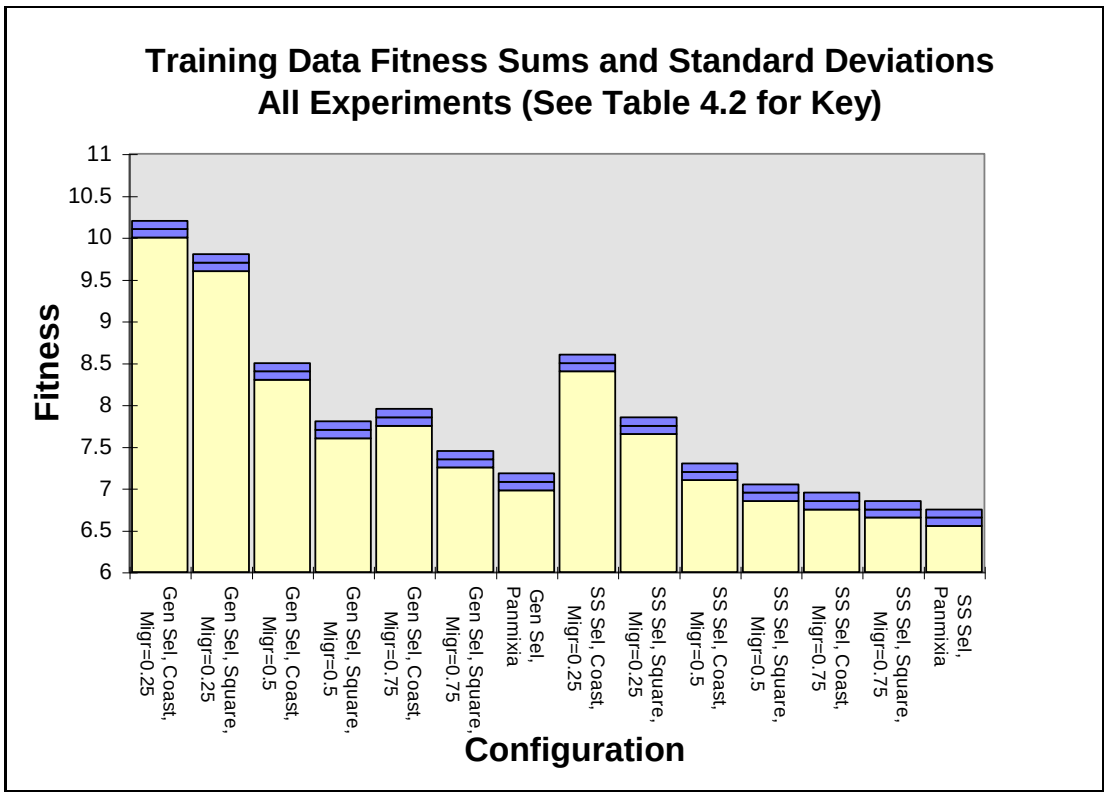


Figure 4.16: Sum of mean fitness for all selection policies across all experiments (std deviation shaded).

Figure 4.16 depicts a similar graph that displays fitness relative to the original training data. We observe that fitness appears to be proportional to mobility, being best for panmictic evolution. We have mentioned in Section 4.4.4 that the training-set fitness measure is common to earlier experiments performed with demes and elitism. This would appear, however, to counter results such as Collins (1992). Why? To see the answer, examine the plots of Figures 4.5, 4.6 and

4.7. The 50-generation cutoff was chosen because it appeared in preliminary experiments that this was sufficient time to allow generalization performance to converge, which it does. Training set performance, however, is still dropping at generation 50. Based on the observation of a great number of charts produced in the manner of Figures 4.5, 4.6, and 4.7, it appears that higher mobility (**D**) leads to more rapid convergence, and hence better training fitness at the time training is cut off.

4.6.8. Performance Using Uniformly Sparse Training Data

Figure 4.17 depicts best-10-average-test-fitness for a set of experiments using sparsely distributed training data. The experiment parameters are summarized in Table 4.4

Table 4.4: Experimental configurations whose results are summed to form the columns of Figure 4.17.

Sigma	Training Set Size	Bites
0.1	40	0
0.3	40	0
0.5	100	0

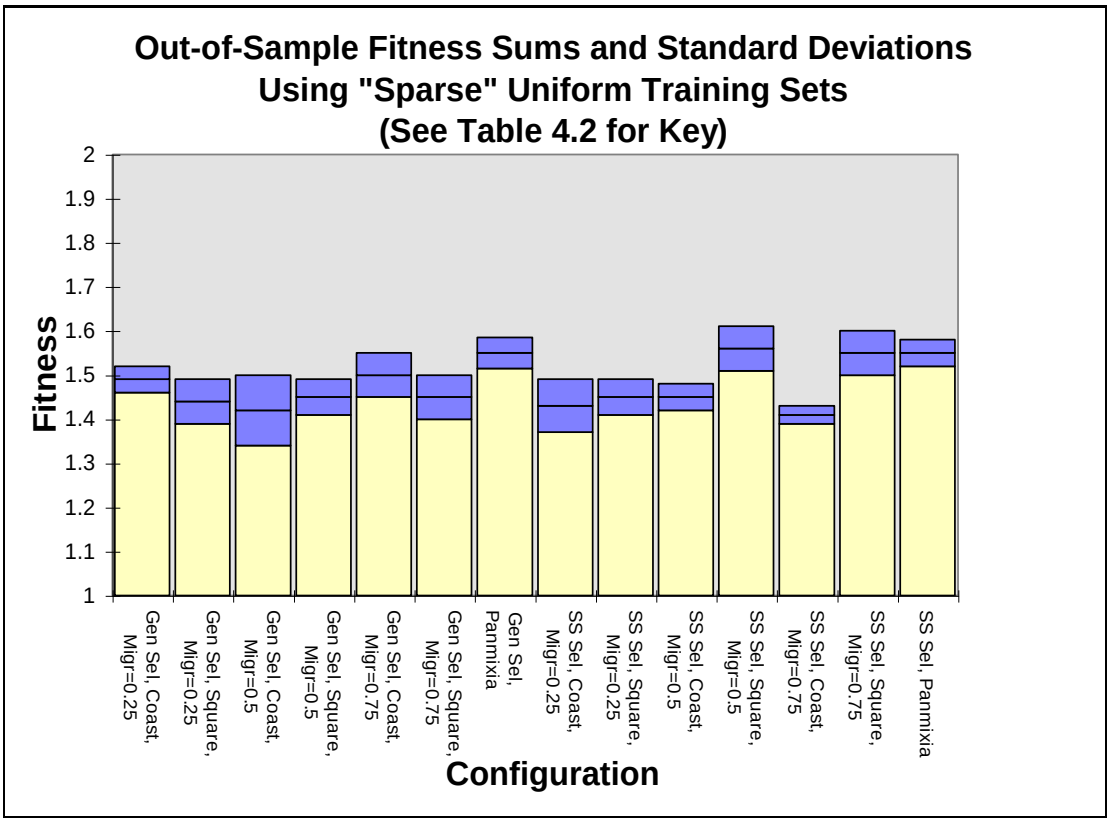


Figure 4.17: Sum of mean fitness for each selection policy for experiments using statistically "sparse" training sets. (std deviation shaded).

The sparse training data case is a particularly hard problem to generalize against. For steady-state-selection, the performance of distributed evolution is consistently the same as or significantly better than panmictic. For generational selection, panmictic fitness is consistently at least a standard deviation above (worse than) that offered by distributed evolution.

4.6.9. Performance Using Nonuniform Training Data

The performance spread achieved with nonuniform training sets (bites), shown in Figure 4.18, appears similar in character to that shown for all data taken together: when using steady-state selection, the distributed scheme always produces the same or significantly better performance. With batch learning, the lowest mobility cases (**D**=0.25) perform significantly worse than

panmictic, as was seen in the test-set performance of Section 4.5.6.1. The experimental parameters are summarized in Table 4.5.

Table 4.5: Experimental configurations whose results are summed to form the columns of Figure 4.18.

Sigma	Training Set Size	Bites
0.1	40	1, 2, 3
0.1	100	1, 2, 3
0.3	100	1, 2, 3
0.3	360	1, 2, 3
0.5	360	1, 2, 3
0.5	680	1, 2, 3

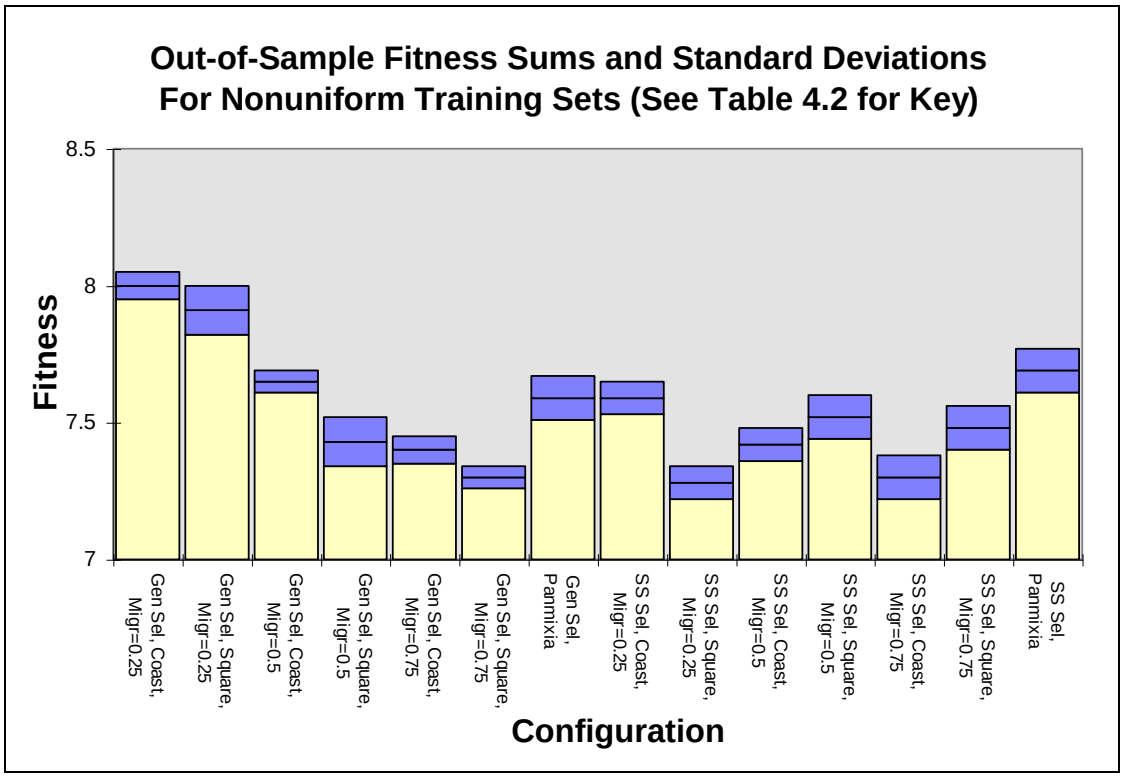


Figure 4.18: Sum of mean fitness for all selection policies using nonuniform training samples (std deviation shaded).

4.6.10. Performance Using Sparse and Nonuniform Training Data

For both steady-state and batch populations, results using distributed evolution are either significantly better, or similar within much less than one standard deviation. For generational selection, however, the trend of the (**D**=0.25) having less favorable fitness properties appears

again, although not nearly as unfavorable as that shown in Figures 4.15 or 4.18. Some explanation of poor performance under D=0.25 is discussed in Section 4.7.1.

Figure 4.19 shows test-set performance for sparse training data combined with bites training data, which is also sparse, as summarized in Table 4.6.

Table 4.6: Experimental configurations whose results are summed to form the columns of Figure 4.19.

Sigma	Training Set Size	Bites
0.1	40	0, 1, 2, 3
0.3	40	0
0.3	40	1, 2, 3
0.5	100	0, 1, 2, 3

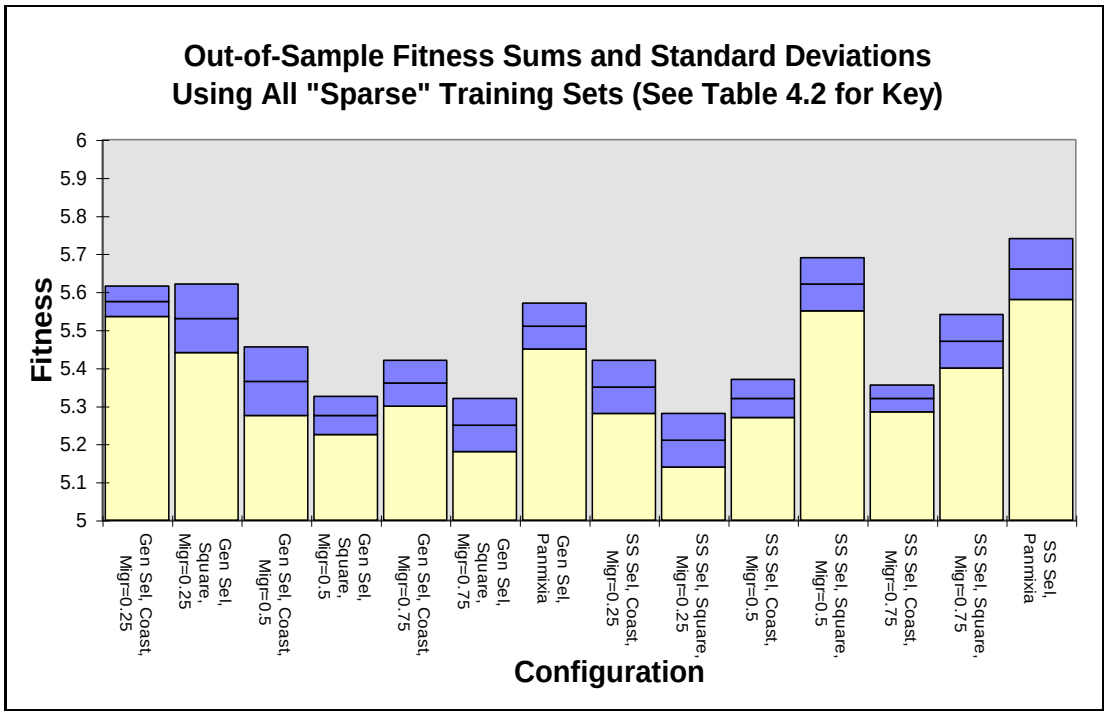


Figure 4.19: Sum of mean fitness, sparse and nonuniform training data (std deviation shaded).

4.6.11. Performance Using Sparse Data With High Degree of Class Overlap

The test-set performance due to interaction of sparse training data with significant class overlap is summarized in Figure 4.20 and in Table 4.7.

Table 4.7: Experimental configurations whose results are summed to form the columns of Figure 4.20.

Sigma	Training Set Size	Bites
0.5	100	0
0.5	180	0
0.5	360	0

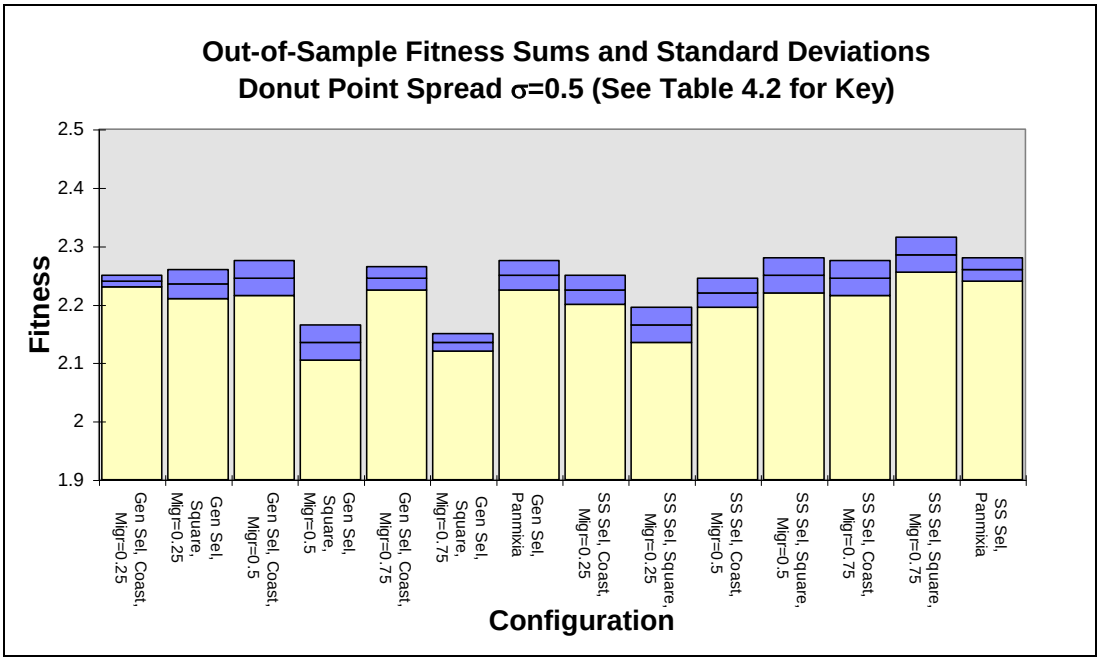


Figure 4.20: Sum of mean fitness for all selection policies for experiments with large class overlap (std deviation shaded).

These runs display a different characteristic previously observed: most methods perform about the same within a reasonable confidence level, but three (distributed populations) display significantly better fitness.

4.6.12. Performance Using Non-Sparse Data Sets

Finally, as a control we do a comparative performance analysis using only densely sampled training data on the theory that these data are easy to classify. The parameters and results are summarized in Figure 4.21 and in Table 4.8.

Table 4.8. Experimental configurations whose results are summed to form the columns of Figure 4.21.

Sigma	Training Set Size	Bites
0.1	100	0
0.1	360	0

0.3	**100**	**0**
0.3	**360**	**0**
0.5	**360**	**0**
0.5	**680**	**0**

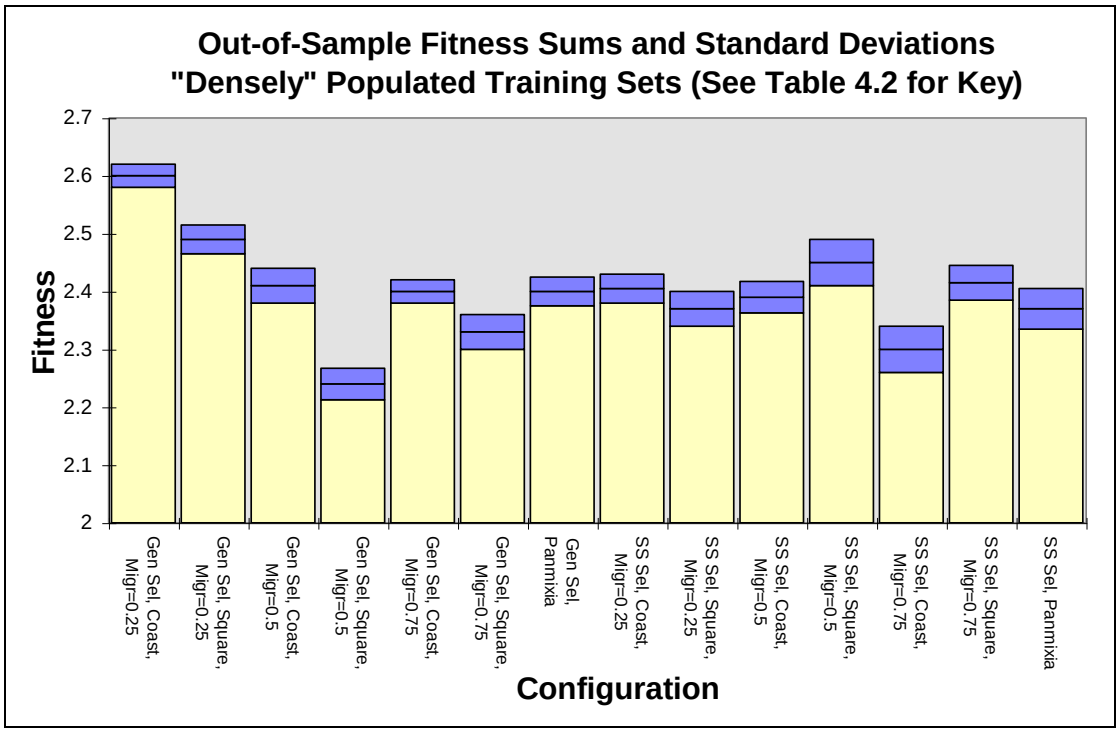


Figure 4.21: Sum of mean fitness for all selection policies for experiments with large numbers of training samples (std deviation shaded).

Here again we see that most configurations, including panmictic, perform about the same, although in the batch-population case, some perform significantly better and worse than the average.

4.7. Discussion

Several results from this study were unexpected. The first and foremost among these is the relationship between training set density and class overlap: in performing up-front experimental design, it was assumed that the spatial density of samples would play a significant role in performance. This is because the volume of a torus is proportional to the square of its cross-section, or minor diameter. For the toroidal distributions of the Donut problem, this diameter should be proportional to the parameter σ. Thus, we might expect that the number of samples

required to cover feature space for σ=0.5 to be about $(0.5/0.3)^2$ = 2.78 times as many to achieve equivalent performance as for σ=0.3. For example, we would expect to see similar quality of results with σ=0.5 and number of training samples = 1,000, as for σ=0.3 and number of training samples = 360. We can readily see that this is not so from Figure 4.11. In fact, we observe that the σ=0.5 case is much closer to its theoretical limit than is the σ=0.3 case for the 360 sample training set, although their rate of convergence (slope) with respect to this limit is roughly the same at that point.

Another unexpected outcome was the lack of any linear increase in classification error, relative to the theoretical limit, with increase in class overlap (σ). It was expected that increased noise, in the form of class overlap, would confound the program induction process. Instead, there was approximately constant difference between theoretical and experimental values. This may be explained in terms of the fact that increasing class overlap does not change the features of the Bayesian solution. Noise may bias the level of minimum error, but the optimal surface itself does not change with σ.

4.7.1. Conclusions About Distributed Evolution

Distributed selection improves performance on the average. Moreover, distributed selection was demonstrated for the case of generalization with respect to out-of-sample performance measures. Previous studies of distributed-selection GA have focused only on in-sample fitness of genetic systems (Collins 1992, Ackley 1993).

So far as specific recommendations for the problem at hand, steady-state selection using a "coastline" (128x2) distributed configuration with **D**=0.75 consistently performed well. For generational selection , 16x16 demes with **D**=0.75 and **D**=0.50 did consistently well. Under that configuration the parameter **D**=0.25 most frequently provided poor results, not only worse than average but worse than panmictic selection in almost all cases. Recall that for the **D**=0.25 case, the actual probability of reaching outside the deme on any given draw is less than 3%. Coupled with the fact that the K-tournament used there picks the best of 6 drawn with replacement from the local population, the best individual of the 4 in the deme will be picked as a parent almost every time. This effectively reduces the size of the global population by a factor of 4.

4.7.2. Conclusions Concerning Steady State Selection

The steady-state selection policy often performed slightly worse on average than did batch population updates. This is most likely due to rapid convergence. On the other hand, rapid convergence means significant speedup relative to other methods. Effectively, this allows more search: by taking less time to achieve convergence, the method can perform a larger number of runs during the same period. Multiple independent runs with different initial conditions are indeed an effective method of maintaining a diverse search of the solution space, so this speed factor may make the steady-state option attractive, even if expected performance on a per-run basis is slightly lower.

4.7.3. Comments on the Procedures Used

For this problem, experimental design was performed up-front, with preconceptions about what problem sizes and parameters should be hard. In general, the GP system dealt much more gracefully with sparseness, nonuniform training data, and class overlap than had been expected, biasing many test runs to explore cases that turned out not to be difficult at all. In doing so, however, this study has set a much tighter bound upon problem parameters most worthy of study.

4.7.4. Big Grids and Other Parameters

Prior studies of distributed evolution have typically used thousands of demes, largely because such a configuration was efficiently mapped onto the SIMD (Single Instruction, Multiple Data) mesh of the computers used in those studies (Collins 1992, Ackley 1993). Those studies, which made a comparison against panmictic evolution found results that were spectacularly better using the distributed scheme (Collins 1992). The research described here has implemented virtual demes in a workstation-based system and maintained a relatively small number of them. Would the results we obtain be similarly improved by scaling up the spatial landscape upon which evolution takes place?

Also, the rate of migration makes an order-of-magnitude jump from **D**=0.25 to **D**=0.5. Especially in light of the fact that degraded performance was often seen at the lower value, it would be beneficial to test a more continuous range of values in between.

4.7.5. Comparative Performance to Neural Networks

The results obtained here reveal properties of generalization and noise-tolerance which are much better than expected. This leaves open the nagging question of whether this success is due to the fact that the Donut problem isn't as hard as we thought it was, despite its intended design. Recent work (Carmi 1994) has shown that (1) the donut problem really is hard, and (2) GP is doing very well in many respects.

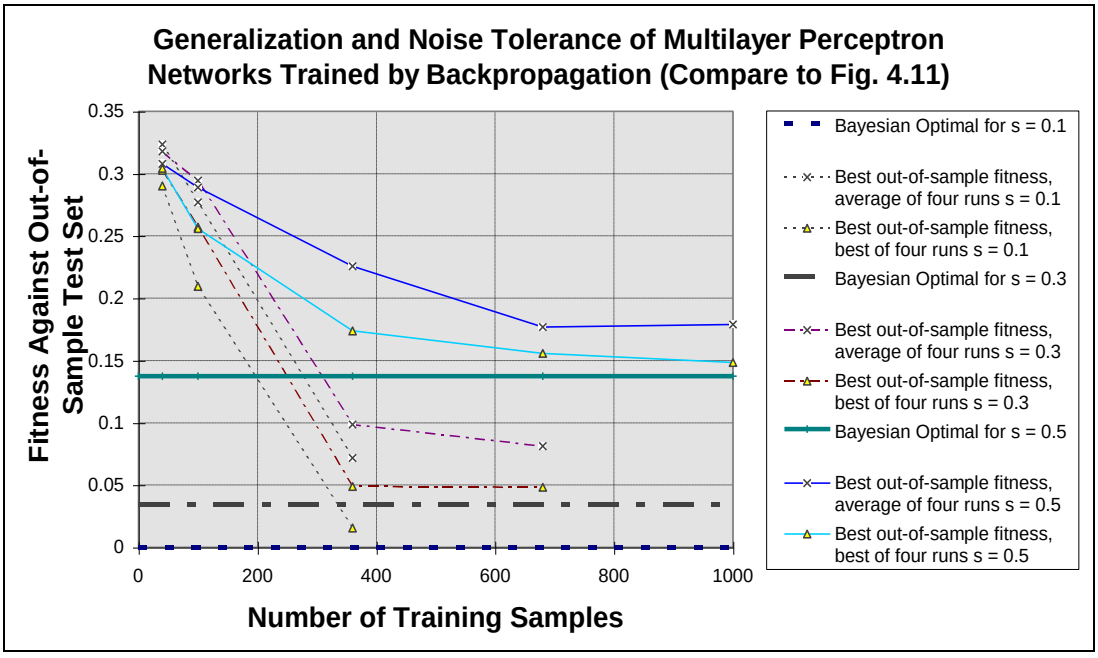


Figure 4.22: Generalization and noise tolerance of neural learning for the Donut Problem. The experiment whose results are shown here is identical to the experiment depicted in Figure 4.11. About 200 hours of labor was devoted to finding the appropriate number of hidden layers and number of units per layer whereas no fine tuning of the GP method was performed. For training sets containing 360 samples or more, the performance of the best neural network is slightly better than the best GP solution, while the average neural performance is slightly worse. For training sets with less than 360 samples the neural method displays much worse performance in all cases, suggesting that some of the power of Genetic Programming lies in the ability of the representation to generalize well. Data courtesy of Aviram Carmi © 1994.

In (Carmi 1994), the author has spent over two hundred hours of trial-and error in optimizing a Multilayer Perceptron / Backpropagation network for use against the Donut problem. This effort includes determining the number of hidden layers and nodes, determining step-size parameters, and even examining several schemes for normalizing the input data. Both

commercial packages and hand-crafted implementations have been used. By contrast, the Genetic Programming system described here used default parameters reported in (Koza 1992) with no optimization specific to this problem. Identical in-sample and out-of-sample data sets were used in both experiments. Some pertinent results are summarized below:

- Generalization of the neural approach was far worse than for the GP approach, with error 4 to 7 times greater for systems trained against small data sets (Compare Figures 4.22 and 4.11).
- Extreme sensitivity to initial conditions. Whereas GP solutions using different initial conditions converge to within 2-3% performance of each other, the typical spread for the neural approach is much greater, often as much as 10%.
- One interesting improvement displayed by the neural approach is that in low noise networks will sometimes achieve a perfect score against the out-of-sample data, i.e., 0 misclassifications out of 2000 samples, despite much higher misclassification on average. Although they are close to 0%, the solutions generated by GP almost always misclassify a small number of samples, e.g., 3 out of 2000.

Chapter 5

An Introduction to the Greedy Recombination Operator

Seeking Locally Optimal Combinations of Parental Traits Reduces Resource Investment and Improves Performance

Many natural organisms overproduce zygotes and subsequently decimate the ranks of offspring at some later stage of development. The basic purpose of this behavior is the reduction of parental resource investment in offspring which are less fit than others according to some metabolically cheap fitness measure. An important insight into this process is that for single-pair matings all offspring are products of the same parental genotypes: the selection taking place therefore seeks the most fit recombination of parental traits. This chapter presents the Brood Recombination operator $R_B(n)$ for Genetic Programming, which performs greedy selection among potential crossover sites of a mating pair. The properties of $R_B(n)$ are described both from a statistical standpoint and in terms of their effect upon search. Results are then presented which show that $R_B(n)$ reduces resource investment with respect to CPU and memory while maintaining high performance relative to standard methods of recombination.

5.1. Introduction: Brood Selection in Natural and Artificial Systems

In nature it is common for organisms, as quoted from (Kozlowski & Stearns, 1989), to "produce many offspring and then neglect, abort, resorb, or eat some of them, or allow them to eat each other." There are many reasons for this behavior, of which the quoted work provides an excellent treatment. This phenomenon is known variously as soft selection, brood selection, spontaneous abortion, and a host of other terms depending upon both semantics and the stage of ontogeny and/or development at which the culling of offspring takes place. The "bottom line" of this behavior in nature is the reduction of parental resource investment in offspring who are potentially less fit than others.

The means of culling offspring are widely varied, ranging from resorbtion of immature seeds by plants (Darwin, 1876) to sibling and parental cannibalism among animals (Stearns, 1987). In some cases the parent uses biochemical measurements to determine which offspring are chosen, while in others such as sibling-competition a more obvious form of tournament selection is at work. The net result is that the resources of the parent are expended only upon those offspring which tend to lie in the upper tail of the fitness distribution induced by the brood culling criteria.

The use of "Soft Brood Selection" in Genetic Programming was first proposed by Altenberg (1993; 1994), who conjectures that it should increase the upper and lower tails of the population fitness distribution. This is due to the effect of "constructional fitness," which describes the variation in fitness of subtrees, or subexpressions, as a function of the loci in which they are inserted under recombination. There are two implications of immediate interest: the first is that an increase in the fitness point spread is effectively an increase in fitness variance, which confers a continued evolvability upon the population according to Fisher's fundamental theorem of natural selection (Fisher 1958). The second is that Altenberg's notion of constructional fitness is based on the idea that a block of code only has a useful net effect when appearing in a specific context: this captures a critical property of computer programs. It is a property that any theoretical foundation for Genetic Programming can ill afford to ignore.

We present the *Brood Recombination Operator* $R_B(n)$ for artificial genetic systems and demonstrate in a limited problem domain that the principle of reduction in parental resource investment indeed carries over from natural systems. It is shown that improved performance can be achieved with significant reduction in CPU and memory requirements relative to standard GP due to reduced population size requirements. The fitness evaluation of brood members is performed with a "culling function" which is a fractional subset of the fitness evaluation function for full-fledged population members. A significant result is that large reductions in the cost of the culling function produce small performance degradation of the population members. We conjecture that random recombination is likely to produce unfit code even when both parents are highly fit: even a simplistic culling function is beneficial since it will tend to discriminate between code which is "in the ballpark" and that which is not.

This Chapter presents a mathematical argument for improved performance due to Brood Recombination and tests the aforementioned conjectures pertaining to it.

5.2. Recombination Operator $R_B(n)$ for Genetic Programming

The term *Brood Recombination* is adopted here in contrast to the term "Brood Selection" used in (Altenberg 1994) and (Tackett and Carmi 1994b). Although a form of selection comprises this operator, it is properly a recombination operator which serves as a substitute for the "standard" Genetic Programming crossover operation defined in (Koza 1992). Thus terminology is chosen so as not to be confused with *selection* operators such as the Steady-State method studied in Chapter 4. Treated as a "black box," the *Brood Recombination Operator* $R_B(n)$ produces two offspring from two parents as does standard crossover. Internally $R_B(n)$ is parameterized by the *brood size factor* n and the *culling function* F_B: whereas standard GP crossover recombines two parents to produce one pair of offspring, $R_B(n)$ produces n pairs of offspring and keeps only the best two pairs. The surviving members of the brood are selected according to fitness values assigned by the culling function F_B which is in general different than the fitness function applied to the population.

We will demonstrate that F_B can be much less costly than the fitness evaluation applied to population members and still retain the effectiveness of $R_B(n)$. This allows many brood members to be evaluated at a cost comparable to the evaluation of one "mature" population member, thereby mimicking the principles of nature. Genetic Programming under $R_B(n)$ can use a smaller population size than is necessary under standard crossover, and the CPU time which would otherwise be used to evaluate extra population members is instead spent on evaluation of large broods. The net result is an improvement in fitness of population members for equivalent CPU resources and reduced memory requirements. The Brood Recombination Operator can be stated in pseudocode form as follows:

Algorithm $R_B(n)$:
1. **Select parents** *P_1 and P_2 from the population per (Koza 1992)*
2. **For** *$i = 1$ to n* **perform crossover** *per (Koza 1992):*
2.1. **Randomly select crossover points** *(subtrees) S_{i1} and S_{i2} from P_1 and P_2*
2.2. **Exchange subtrees** *S_{i1} and S_{i2} to form brood offspring B_{i1} and B_{i2}*
3. **Test the 2n members** *of the brood to determine fitness according to F_B*
4. **Sort the 2n members** *of the brood in order of fitness*
5. **Select offspring** *("Children") C_1 and C_2 which are fittest of the 2n brood members*
6. **Return** *C_1 and C_2 as the offspring produced by $R_B(n)$*

The procedure described above is similar to performing K-tournament selection with K = 2n among all possible successors which can be generated from a pair of individuals by the crossover operation (see Section 5.5). Because all members of the tournament are derived from the same parent genotypes, $R_B(n)$ is selecting among recombinations rather than selecting among individuals,.

5.3. Reduced Investment of CPU and Memory Resources

Step 3 of algorithm $R_B(n)$ evaluates the fitness of the 2n brood members according to F_B. If F_B is the same fitness measure used to evaluate the population at large then the cost of fitness evaluation, typically the bottleneck in most GP experiments, is multiplied by a factor of n. Clearly, this is not nature's model for brood selection: it is the analogy of letting each potential offspring lead a full mature life. So then, how little investment of resources be can expended by F_B? Can F_B be engineered or evolved to minimize the expense of the preselection process? For any complex task the random insertion of a subtree into a program which results from a single GP crossover operation will frequently result in nonsense code even when both parents are highly fit. Therefore it is easy to imagine that for many problems fitness evaluation for population members may be quite costly while a criteria F_B that discriminates between programs which are "in the ballpark" as opposed to being "totally bogus" may be quite cheap. The quantitative approach we take is to parameterize the costs of brood selection in terms of CPU (time) and memory (space) resources relative to standard GP, where resource costs are defined as follows:

C_P = CPU cost of evaluating a full-fledged population member;
C_B = CPU cost of evaluating a brood member using F_B;
M = Average "size" of population or brood member in memory.

Then we can define algorithm resource investment on a per-generation basis:

Time to create new population under standard GP crossover:

5.1 $$T_1 = C_P N_1$$

Space to create new population under standard GP crossover:

5.2 $$S_1 = MN_1$$

Time to create new population under $R_B(n)$:

5.3 $$T_2 = C_P N_2 + nC_B N_2$$

Space to create new population under $R_B(n)$:

5.4 $S_2 = M(N_2 + 2n)$

where N_1 and N_2 are the population sizes required to produce "equivalently good results" between standard GP and $R_B(n)$ respectively, and the cost of evaluating population fitness is included in the computation of T_1 and T_2. The goal of applying $R_B(n)$ is that such results can be achieved with $T_2 < T_1$ and $S_2 < S_1$ due to $N_2 < N_1$ and $C_B < C_P$. Note that the given formulae assume that crossover is the only reproduction operator, although it is easy to adapt them if there are other operators used. Whereas savings of CPU resources is a relatively obvious benefit, note that in the experience of the author, memory resources are often the limiting factor in the size and number of GP runs that can be achieved on a given system. Even under virtual memory machines a large population can cause "thrashing" as members are swapped in and out of real memory during evaluation and breeding. Therefore it may make sense in practice to apply $R_B(n)$ with F_B the same as the population fitness function strictly as a memory-saving device. In that case the first term in Formula (3) can be omitted since the value produced by F_B for each brood member can be re-used as the fitness rating for selection.

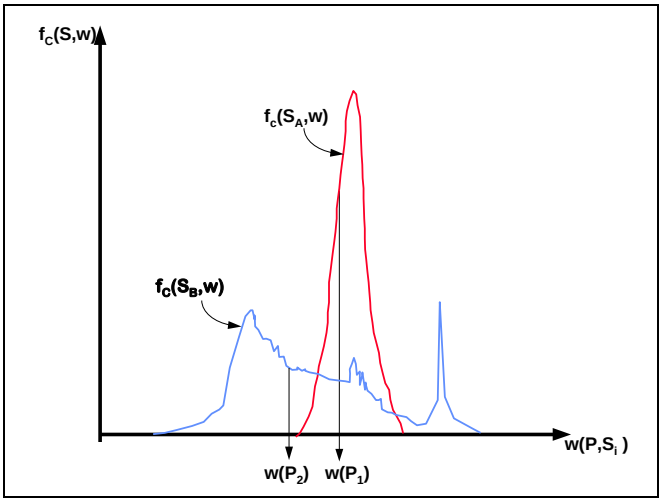


Figure 5.1: Canonical fitness distribution $f_c(S,w)$ of two subexpressions S_A and S_B. Fitness values of parent expressions P_1 and P_2 are points within these distributions.

5.4. Greediness of Brood Recombination

Because Brood Recombination selects among subexpressions and their insertion sites, it is desirable to introduce a representation of the program search space which allows visualization

of the process taking place. Toward this end we introduce the concept of the *canonical fitness* of expressions, which we will denote as $f_c(S,w)$. Canonical fitness refers not to a scalar fitness value, but rather to a probability distribution associated with the expression S. Specifically, $f_c(S,w)$ is the relative frequency with which S occurs within programs that have fitness value w. This frequency is taken over the space of all programs P with size less than some fixed positive integer k. Size may be considered as the count of atomic functions and terminals within a program, or as the nesting depth of expressions (placing limits on size is reasonable from the standpoint that trees of infinite size cannot be evaluated). Over the set of all subexpressions **S**, the ensemble of fitness distributions $f_c(\mathbf{S},w)$ forms a canonical search space. It is dependent only upon the function set, terminal set, and fitness evaluation function: these are exactly the critical design elements in Genetic Programming.

Figure 5.1 depicts the canonical fitness for two expressions, S_A and S_B: the x-axis is $w(\mathbf{P},S_i)_{i=\{A,B\}}$, the fitness of all programs which contain the subexpression S_i. The y-axis is the probability $f_c(S_i,w)$ that S_i occurs in an expression with fitness w. A useful transformation of $f_c(S_i,w)$ is the probability density $F_c(S_i,w)$. It is the probability of finding a program containing S_i which has fitness less than or equal to w:

$$F_c(S_i,w) = \int_{-\infty}^{w} f_c(S_i,v)\, dv \qquad 5.5$$

Figure 5.1 provides some insights to the operation of $R_B(n)$. Consider that S_A is the subexpression chosen from the *crossover point* of parent P_1 and inserted into the crossover point of parent P_2 to form offspring B_{i1}. Likewise, S_B is the subexpression inserted from P_2 to P_1 to form B_{i2}. Thus $w(P_1)$ and $w(B_{i1})$ are both points in $f_c(S_A,w)$, while $w(P_2)$ and $w(B_{i2})$ are both points in $f_c(S_B,w)$: the production of offspring via crossover represents a traversal of the curve described by $f_c(S_i,w)$. Without loss of generality we assume that the same subexpression is not randomly drawn multiple times. Therefore $R_B(n)$ is sampling an ensemble of 2n distributions: n of these distributions are denoted $f_c(S_{i1},w)_{i=\{1,n\}}$ and have $S_{i1} \varepsilon P_1$ in common; the other n distributions sampled are $f_c(S_{i2},w)$ where $S_{i2} \varepsilon P_2$. Price's covariance theorem (Price 1970; Altenberg 1994) shows that *evolvability* of a system, the ability of a population to continually improve, requires correlation between the distributions of parent and

offspring fitness. In light of this, we will consider the performance of $R_B(n)$ in a "worst case scenario." Let us assume that $w(B_{ij})$ is drawn from $f_C(S_{ij},w)$ independently of $w(P_j)$. This implies that *$w(B_{ij})$ is uncorrelated with $w(P_j)$*, which according to Price means that the selection of subtree crossover points, the transmission function, is resulting in *random search*. In this case the probability that $w(B_{ij}) > w(P_j)$ is just 1- $F_C(S_{ij},w)$: qualitatively, this is related to how close $w(P_j)$ is to the "upper tail" of distribution $f_C(S_{ij},w)$. The probability that at least one offspring has $w(B_{ij}) > w(P_j)$ is then just one minus the probability that no offspring has $w(B_{ij}) > w(P_j)$. Under the simplifying assumption that the distributions $f_C(S_{ij},w)$ are independent for each subexpression S_{ij} then the chances of this occurring are:

$$5.6 \qquad p\{ [\mathbf{max}_{i=1,n}(w(B_{ij}))] > w(P_j) \} = 1.0 - \prod_{i=1}^{n} F_c(S_{ij}, w(P_j))$$

Since $0.0 \leq F_C(S_{ij},w(P_j)) \leq 1.0$, the chance of producing a fitter offspring sharing a common subexpression S_{ij} increases exponentially with brood size n. In practice, distributions of parent and offspring fitness are certainly correlated: this quantity can be measured, but the mitigating effects vary on a per-problem basis. Likewise not all distributions $f_C(S_{ij},w)$ will be independent. Thus this analysis can only provide some expectations which must be examined empirically. Some detailed empirical evidence will be discussed in Chapter 6, Section 6.6.

5.5. Greedy Recombination as Search

Chapters 2 and 4 have provided an ongoing analysis of Genetic Program Induction as search on the space of programs, concluding that it is analogous to beam search algorithms often applied in AI. Greedy recombination carries the beam search analogy further by pruning of the search tree based on a "look-ahead" technique. In Equation 2.2 it was shown that an upper bound on the number of successors which can be produced by recombination is the population size times the square of the average expression size, meaning that only a fraction of the possible successors can possibly be visited in one generation. It was argued that the choice of successors from this large set is not uniformly random, but rather is biased towards successors which incorporate partial solutions from states which have received a relatively good fitness, or

heuristic rating. In other words, recombination is driven by selection, which picks a pair of individuals *i* and *j* that are likely to be drawn from near the head of the population queue. Once states *i* and *j* have been picked, the number of possible successors of *i* is given by:

5.7 $$P_{Si} = SIZE_i\ SIZE_j$$

This is of course the same as the number of possible successors for state *j* which can be generated by recombination. Under standard Genetic Programming recombination as described in section 2.1.2.3, one state is chosen randomly from among the possible successors of *i*, and one is chosen from among the possible successors of *j*. By way of analogy, the Harpy speech understanding system (Lowerre and Reddy 1980) uses a "local" form of beam search in which all possible successors of a node are generated and those which fall below some threshold of worth, based on the heuristic value of the best successor, are pruned back. Greedy recombination generates a subset of 2N possible successors from the total set of $2P_{Si}$ and prunes all but the best two. Aside from the fact that all possible successors are not examined there is also a difference between the methods since the heuristic used to evaluate the 2N states is normally different from the heuristic used to order the population queue.

Greedy recombination can effect minor changes to the structure of the search tree relative to the standard "random" method. Each of the two children produced by a single random recombination is the successor to one parent using the convention that parent and child differ only by the inserted subexpression. Using the same convention in $R_B(n)$, both offspring may be successors of a single parent and no successors from the other parent will be retained.

In summary, greedy recombination substitutes a randomized version of greedy search for the otherwise random choice of insertion sites. It does not affect the average branching factor due to recombination, nor does it affect the probability with which states will be selected to participate in recombination. Once the two parent states are selected it does not affect the number of possible successors which can be generated from that pair through different choices of subexpression insertion and donation sites. What it does affect is the probability with which those sites are chosen. Therefore we may expect to see effects of $R_B(n)$ which are somewhat decoupled from methods of selection used. Evidence of this is provided in Chapter 6.

5.6. Method

We have run a series of experiments based on the "Donut Problem" described in Chapter 4. For these experiments the toroidal distributions have a Gaussian cross-section of $\sigma = 0.5$, providing about 13.5% class overlap relative to the optimal surface which minimizes error in the partitioning of the two categories (i.e., the Bayesian solution classifies only 86.5% of the samples correctly). For all experiments described we have used the percent correct classification of 360 "training" samples from each distribution as the fitness function for population members. The use of percent correct classification implies a greater-is-better fitness measure, which is just the opposite of the fitness measure used for the same problem in Chapter 4. This is done to maintain consistency with other problems introduced in Chapters 6 and 7. All performance statistics reported are based on percent correct classification of 1000 independent samples from each distribution which measure the "out-of-sample" performance by the best-of-generation individual. Specifically, for each generation the best individual is determined based on performance against the training set. This individual is then tested against an out-of-sample data set. If that performance figure is better than the performance achieved by any previous best-of-generation individual, then the new best-of-generation individual is recorded as the best-of-run individual. Each point in each plot represents the average best-of-run individual performance averaged over 10 runs using different initial conditions (random seeds).

5.7. Results

Figures 5.2 and 5.3 plot performance, here depicted in terms of percent correct classification, against CPU resource cost in terms of C_p, the amount of computation required to evaluate one population member for fitness. Each plot consists of twenty-five points: five of these points depict performance using standard GP crossover with population size N_1=N, 2N, 3N, 4N, and 5N. The other twenty show performance using five different culling functions combined with four different brood sizes. The runs with different brood sizes apply $R_B(n)$ with N_2=N and n=2, 3, 4, and 5. The five culling functions apply $R_B(n)$ using F_B with decreasing cost. This is achieved by decimating the training set, using only one in *k* training samples to evaluate brood members. These are labeled in terms of culling function cost C_b divided by the fitness evaluation cost C_p. The "base" population size N is 250 in Figure 5.2 and 500 in Figure 5.3. Since fitness is given as percent correct classification, greater fitness values indicate higher

performance, with the possible range from 0.0 to 1.0. The highest performance which is likely to be achieved in theory is about 0.865 and the minimum performance likely to be achieved is about 0.500. In practice the fitness of best-of-generation individuals ranges from about 0.72 to 0.84. This measure is just the inverse of the error-based fitness depicted in Chapter 4 experiments. The CPU cost on the x-axis is computed from equations 5.1 and 5.3 based on the above stated values of population size (N), brood size factor (n), population fitness evaluation cost (C_p), and culling function cost (C_b).

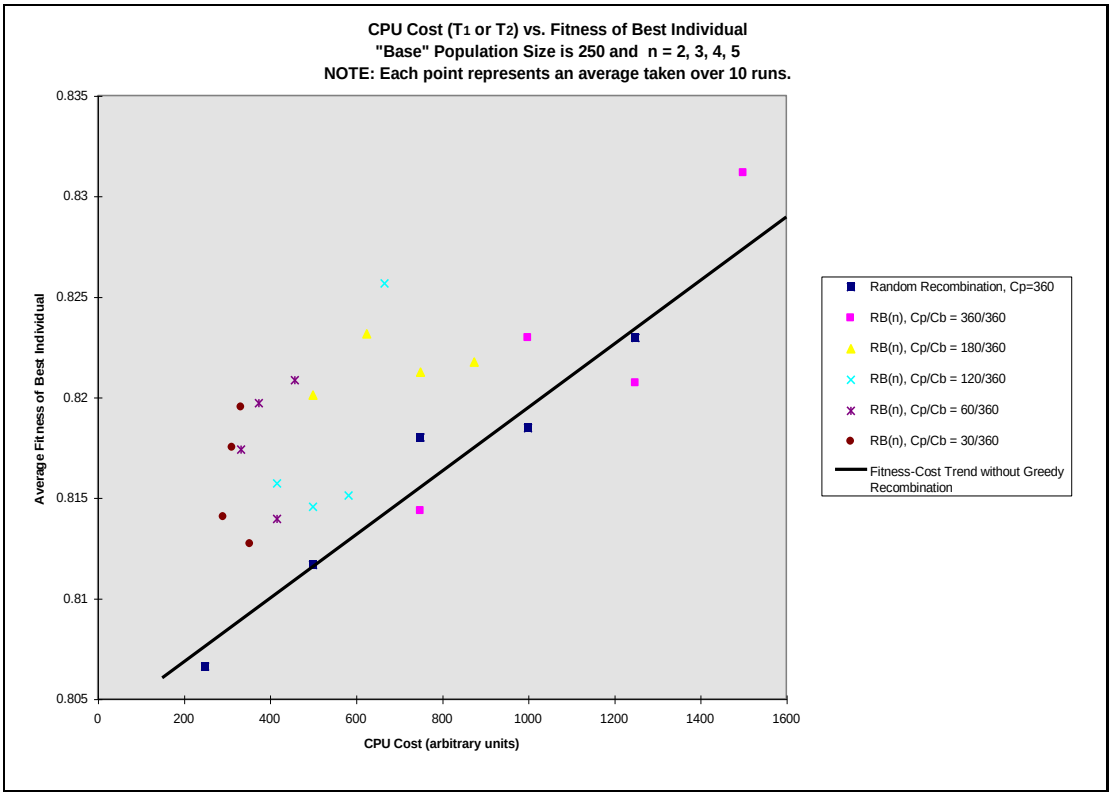


Figure 5.2: Fitness of best-of-run individual vs. CPU resource use for a variety of recombination parameters. A trend line is fitted to five points which depict Genetic Programming with "standard"-i.e. random choice of-recombination insertion sites using population sizes of 250, 500, 750, 1000, and 1250. Also shown are five families consisting of four points each, which use a population size of 250 and culling functions F_B having CPU costs 360/360, 180/360, 120/360, 60/360, and 30/360 relative to the fitness evaluation function which drives selection. The four different points in each family are generated by using different brood size factors n=2, 3, 4, and 5 which pick the best two recombinations out of 4, 6, 8, and 10 trials, respectively.

In each chart a linear trend is fit to the data produced by standard Genetic Programming recombination. In almost all cases the data produced under $R_B(n)$ fall above the trend line, meaning that higher fitness is achieved for a given amount of computation. Average performance increase for the same amount of computation can range from 0.5% to 1.5% using $R_B(n)$. For comparison, examine the trend line for standard recombination shown Figure 5.2. There it is necessary to increase the population size from 250 to 1250 individuals in order to achieve a 1.5% performance increase, and a similar trend can be seen when population is increased from 500 to 2500 in Figure 5.3. This indicates that the artificial genetic system achieves an improved efficiency in terms of resource investment under $R_B(n)$ as do biological systems using "Soft Selection." These charts depict only CPU resources, however. What sort of efficiency is achieved in terms of memory? Examining equations 5.2 and 5.4 we see that memory cost of evaluating a population of 250 expressions using standard recombination is just 250M, 500M for a population of 500, and so on, where M is the average size of (memory occupied by) an expression in the population. Observe that memory cost (in terms of M) given by equation 5.2 takes the same form as CPU cost (in terms of C_p) given by equation 5.1. Therefore, for standard recombination the numeric values and trend line are the same for memory as they are for CPU. Under $R_B(n)$ they are not. Unlike CPU, the memory used under $R_B(n)$ does not depend on the culling function F_B. Applying Equation 5.4 to the parameters of Figure 5.2 we see that memory costs are 254M, 256M, 258M, and 260M respectively for n=2, 3, 4, and 5. If these values were to be plotted in the manner of Figure 5.2 they would appear as a nearly vertical line near x=250. Likewise, memory use for the parameters given in Figure 5.3 is 504M, 506M, 508M, and 510M for n=2, 3, 4, and 5: these would form a nearly vertical line near x=500. Therefore, memory efficiency is far more dramatic under $R_B(n)$ than is CPU efficiency.

5.8. Discussion

Figures 5.2 and 5.3 demonstrate empirically that for this problem brood selection is a favorable method in terms of performance vs CPU and memory resource investment. One interesting feature of these results is that performance of $R_B(n)$ decreases very gradually as a function of the cost of culling function F_B. For example, using F_B with a cost of 30/360 may only drop performance about 1% relative to using a function of 360/360 which is twelve times as expensive.

5.8.1 *Why* Is Brood Selection an Improvement?

We have shown in Sections 5.4 and 5.5 several reasons why improved performance should be expected. These are based on simple probability arguments coupled with the power of understanding Greedy Recombination in terms of search. These arguments can be summarized by saying that greedy choice of potential recombination sites is preferable to random choice, which is essentially no choice at all. Based on the search model, we can argue that by "looking ahead" at some 2n potential successor states and choosing the two best we are achieving similar effects to those that could be had from a population size n times as large. This is not quite so, since a larger population has a separate effect of promoting increased diversity, embodied in the search model as an increased average branching factor. Do these arguments also comment on why $R_B(n)$ is still very effective using a relatively cheap culling function F_B? They do when there is any correlation between the culling function and the actual fitness function, which is the case in the experiments presented here. To paraphrase the previous argument, making a coarse distinction between the fitness of potential recombinations (by using a cheap F_B) is preferable to making no distinction at all. The culling function will be effective to the degree that it consistently chooses the same two members of the brood as the population fitness function would.

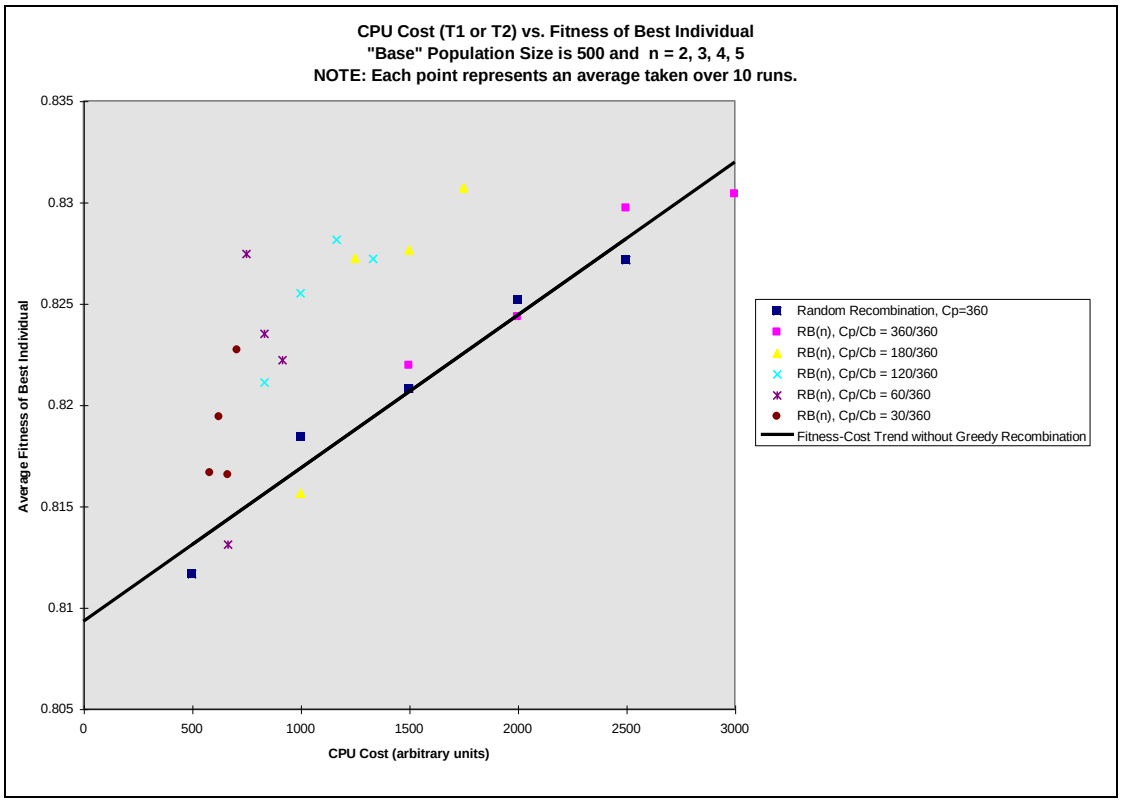


Figure 5.3: Fitness of best-of-run individual vs. CPU resource use for a variety of recombination parameters. A trend line is fitted to five points which depict Genetic Programming with "standard"-i.e. random choice of-recombination insertion sites using population sizes of 500, 1000, 1500, 2000, and 2500. Also shown are five families consisting of four points each, which use a population size of 500 and culling functions F_B having CPU costs 360/360, 180/360, 120/360, 60/360, and 30/360 relative to the fitness evaluation function which drives selection. The four different points in each family are generated by using different brood size factors n=2, 3, 4, and 5 which pick the best two recombinations out of 4, 6, 8, and 10 trials, respectively.

5.8.2 Other Results

The brood selection concept was originally proposed by (Altenberg 1993) with an assumption that F_B is the same as the population fitness function. For that case (Kinnear 1993) has achieved results using the Boolean n-parity problem showing that the performance of brood recombination using a population of size N and brood size factor of n often achieves performance equivalent to using standard recombination and a population size of Nn. Although it does not address the issue of CPU resource investment, that result does reinforce the concept of $R_B(n)$ as memory-saving device.

5.8.3 Brood Selection and Genetic Diversity

There is clearly not a one-to-one correspondence between brood selection $R_B(n)$ and standard crossover with correspondingly larger population (i.e., $N_1 = nN_2$) since the larger population size will have increased diversity of the gene pool. Since $R_B(n)$ is a tournament method which selects crossover points and hence subtrees how greedy is it? Can it act to reduce the diversity of subtrees, eliminating ones which are unfit in the current generation but which might be useful at a later time? Chapter 6 introduces new problems and new tools which can help to directly address these issues.

Chapter 6

Constructional Problems

Problems Which Test the Underlying Mechanisms of Genetic Programming Explain the Power of Certain Operators

We formulate the class of *constructional* problems, which allow precise control over the fitness structure in the space of expressions being searched. The constructional approach is used to create simple GP analogies of the "Royal Road" and "Deceptive" problems common to studies of classical GA. The effects of $R_B(n)$ upon search properties, fitness distribution, and genotypic variation are examined and contrasted with effects of selection and recombination methods.

6.1 Introduction

For purposes of testing we present a class of problems borrowed from binary-string GA which is new to the domain of Genetic Programming. In these "Constructional Problems" fitness is based strictly upon syntactic form of expressions rather than semantic evaluation. A particular target expression is assigned a "perfect" fitness, while those subexpressions resulting from it's hierarchical decomposition comprise intermediate fitness values. If the intermediate fitness values increase monotonically with the complexity of subexpressions in the hierarchy then we call the resulting problem a "Royal Road" after (Mitchell, Forrest, and Holland 1992; Forrest & Mitchell 1993). Alternatively, if some intermediate expressions have lower fitness than subexpressions which they contain the resulting problem is designated "deceptive" (Goldberg 1989). These problems are applied to the continuing analysis of Greedy Recombination begun in Chapter 5, as well as to other methods of selection and recombination which form the basis for comparison. In addition, a variety of measurements are introduced which track fitness moments, probability of success, and diversity of subexpressions occurring in the population.

6.2 Constructional Problems for Genetic Programming

Genetic Programming has traditionally performed semantic evaluation of expressions in order to determine their fitness. Such problems do not lend themselves well to the study of how Genetic Programming may use building blocks, to paraphrase (Goldberg 1989), "[to] construct better and better [trees] from the best partial solutions of past samplings." The Boolean n-multiplexer (Koza 1992) is a scaleable difficult problem for which a solution *may* be recursively composed from partial solutions, but not *necessarily* so. Often, a problem sufficiently difficult to be interesting yields parsimonious yet "unexpected" solutions with no consistent set of intermediate building blocks: (Kinnear 1993a) provides a good example of this. Other difficult problems such as the induction problem described in Chapter 3 may have no known solution whatsoever, and only comparative performance in relation to other methods may be measured. Finally, the difficulty which arises in a problem may not be structural: a subtle change in the atomic functions or terminals may yield easy solutions to a previously intractable problem (Reynolds 1994a; 1994b). In almost all cases, it is possible for Genetic Programming to construct programs with high fitness whose structure cannot be easily understood (Figure 3.3 and Table 7.2): in such instances it is hard to tell which subexpressions of a program actually contribute to fitness and which are "hitchhiking" (Section 7.3). In the context of testing a new operator we wish to devise problems which minimize these sources of ambiguity.

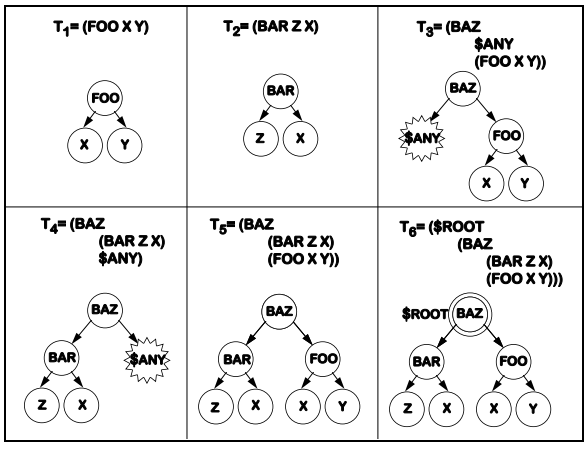


Figure 6.1: Six patterns $T_i\{i=1...6\}$. Each is matched against a program P sequentially, and if a match is made with some subexpression within P, then P is assigned fitness w_i. The pseudofunction $ROOT requires that its argument match the entire expression P. The wildcard $ANY matches any subexpression, including terminals. For the Royal Road the w_i are {0.125,

0.125, 0.25, 0.25, 0.5, 1.0}. For the deceptive problem they are {0.125, 0.125, 0.25, 0.25, 0.125, 1.0}.

Researchers in Genetic Algorithms have long turned to pattern-based problems in order to better understand the genetic search procedure: fitnesses may be assigned to particular bit patterns in order to create false peaks which render them *deceptive* (Goldberg 1989). Likewise, fitnesses may be assigned to building blocks in ascending order of complexity to yield a *Royal Road* problem that "should" be ideally solvable by a Genetic Algorithm (Mitchell, Forrest, and Holland 1992; Forrest and Mitchell 1993). In the Genetic Programming analogy a *constructional problem* is defined here by a set **T** of *target* expressions T_m {m=1...M} to which are assigned fitness values w_m: a program P is searched for expression T_m, and assigned value w_m when a match occurs. In the system implemented by the author a variety of wildcards are allowed, and the option exists to vary fitness value according to the number of times T_m occurs within P. For the purpose of this chapter, however, a relatively minimal problem is implemented. A single set of patterns **T** is used, and the set of fitness values **w** is altered to produce "deceptive" and "Royal Road" problems.

Figure 6.1 depicts the pattern set both in expression and pictorial "tree" form. The goal is to construct an expression solely consisting of the pattern **(BAZ (BAR Z X) (FOO X Y))**. Any expression which matches this pattern exactly receives a fitness of 1.0 (out of a possible 1.0). Expressions which contain **(BAR Z X)** or **(FOO X Y)** (or both) are assigned a fitness of at least 0.125. Those which contain the more complex patterns **(BAZ $ANY (FOO X Y))** or **(BAZ (BAR Z X) $ANY)**, where the wildcard **$ANY** matches any subexpression, are assigned a fitness of at least 0.25. The Royal Road and deceptive problems used here differ only in the fitness assigned when the expression **(BAZ (BAR Z X) (FOO X Y))** is matched to a proper subexpression of P. For the Royal Road a value of 0.5 is assigned, while for the deceptive problem a value of 0.125 is assigned. In the former case intermediate solutions are assigned fitness which increases monotonically, providing "stepping stones" to the solution. In the latter case there is a fitness "valley" between intermediate solutions and the final solution.

In addition to the functions and terminals necessary to construct the fitness-producing target patterns, initial populations includes several "inert" functions: **NOP_1**, **NOP_2**, and **NOP_4**,

have 1, 2, and 4 arguments respectively and do not match any pattern in **T** (except the sub-pattern **$ANY**). Numeric constants resulting from the pseudo-terminal RANFLOAT (see Sections 3.4.2 and 4.2) comprise about 25% of the terminals of the initial population and they likewise do not match any target patterns.

6.3 Methods of Recombination and Selection

Three forms of selection are tested here, each alone and in conjunction with brood recombination for a total of six methods, or "breeding policies." For the three breeding policies that do not include $R_B(n)$ a population size of 500 is used. Those which employ $R_B(n)$ use a population of size 250 with n=5: each mating under $R_B(5)$ results in a total of ten offspring of which the best two "survive." The culling function F_B consists of the patterns $\{T_1 , T_2 , T_3\}$ and their associated fitness values. The combination of population sizes and culling function F_B are chosen a-priori based on experience using $R_B(n)$ on the Donut problem described in Chapter 4. Brood-size factor n is determined empirically so that runs with and without brood recombination run in approximately the same amount of CPU time. No attempt is made to optimize relative performance based on these parameters.

Selection and recombination are the only operators employed: mutation, code-hoisting, and other techniques are eschewed in order to reduce ambiguity of results. Those breeding policies which do not employ $R_B(n)$ instead apply crossover exactly as set forth in (Koza 1992) using the "crossover-at-any-point" method. Individuals in the initial population are limited to a depth of 4, and individuals resulting from crossover are limited to a depth of 6. When individuals are chosen via selection, 50% participate in recombination while the other 50% are cloned (see Section 2.2.3.2). The three selection methods are detailed below. One of them, random selection, is discussed briefly in terms of search. The other selection and recombination methods have already so treated in Chapters 2, 4, and 5.

6.3.1 Random Selection

Section 5.4 presented arguments concerning the performance of $R_B(n)$ in a case where parent and offspring fitness distributions are uncorrelated, resulting in random search. Such statistical properties might be approximately imposed by randomly generating "parents" rather than selecting them from the population. More useful to a study of recombination, however, are

"random walks" that result from performing selection that is not driven by fitness. In the latter method, most schemata survive from generation to generation with distributions affected only by drift and by disruption/creation due to recombination. Using *random selection* with standard crossover there is no bias whatsoever to drive the population to greater fitness. Under $R_B(n)$ random selection allows the isolation of fitness effects due to greedy recombination.

6.3.1.1 Random Selection as Search

The replacement of fitness-based selection with uniform random selection alters search characteristics at a fundamental level. In particular, Chapter 2 modeled selection via the formation of an "interim queue" from which successor states of a search are generated. Parents occur in the interim queue with a frequency proportional to their rank in the population. We argued there that the transformation is similar to the transformation undergone by the priority queue in a beam search algorithm allowed to run over several iterations. This does not hold for random selection. Under random selection the population does not take on statistical properties that are associated with a queue structure since elements are equally likely to be selected from any position regardless of rank. Therefore the interim queue will be expected to contain elements in roughly the same frequencies as the population queue. The formulae for branching factors due to recombination are the same as those given in Chapter 2, but recombination is no longer biased by selection to insert subexpressions drawn from fitter members of the population.

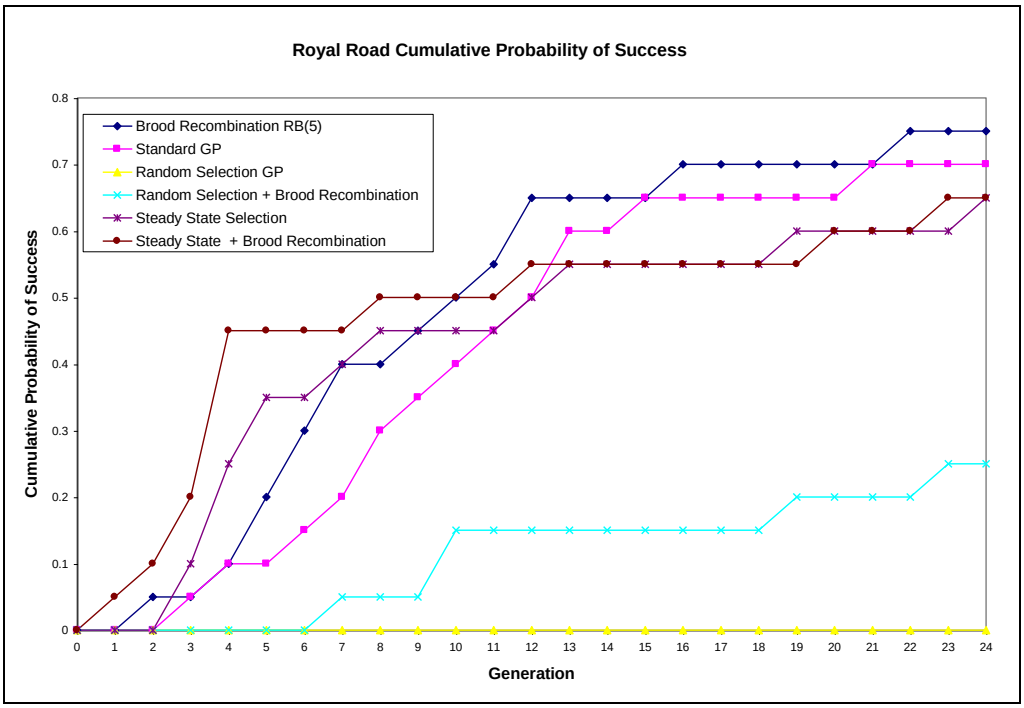


Figure 6.2: Cumulative Probability of success for the Royal Road Problem.

6.3..2 Generational K-Tournament Selection

Generational K-tournament selection is chosen here as the "standard" method. It has been treated in detail in Chapters 2 and 4, but is briefly recapitulated here for completeness. The tournament method is recognized as having improved properties relative to fitness-proportionate selection (Whitley 1989; Goldberg and Deb 1991). It proceeds by selecting K individuals at random from the population and choosing the fittest one. Individuals resulting from cloning and recombination are generated sequentially and stored in a list with no effect upon the population from which parents are drawn. When the list of new individuals is equal to the population size it replaces the old population. Tournament size is set to K=6; this value has been recommended as a good choice over a large range of conditions (James Rice, personal communication 1992), and corroborated by the experiments documented in Chapters 3, 4, and 5.

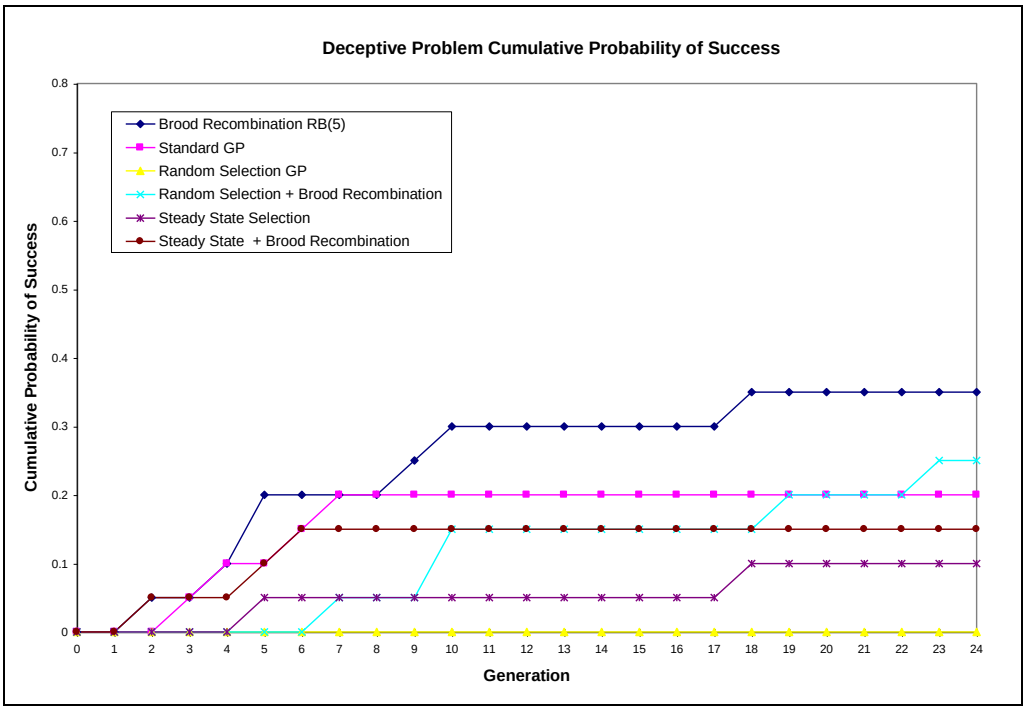


Figure 6.3: Cumulative probability of success for Deceptive Problem.

63.3 Steady-State K-Tournament Selection

Steady-State selection, which was studied in detail in Chapter 4, is used here to represent greedy selection procedures. In Genetic Programming it has been used to advantage, for example, in (Kinnear 1993b). Rather than creating a new population "batch" as described above, individuals are created and tested continuously. The key performance feature of this method arises from the fact that when new individuals are created they replace some existing member of the population: the "building blocks" comprising the replaced individual are lost in favor of those belonging to the offspring of some fitter (in probability) individuals. Performance depends critically on how individuals are selected for replacement (DeJong 1993). In particular, overly greedy replacement methods such as choosing the worst member of the population are typically detrimental to performance and should be avoided (Syswerda, personal communication, ICGA 93). The implementation performed here uses K-tournaments (K=6) to select "winners" for duplication and recombination, and to likewise select the "losers" which will be replaced.

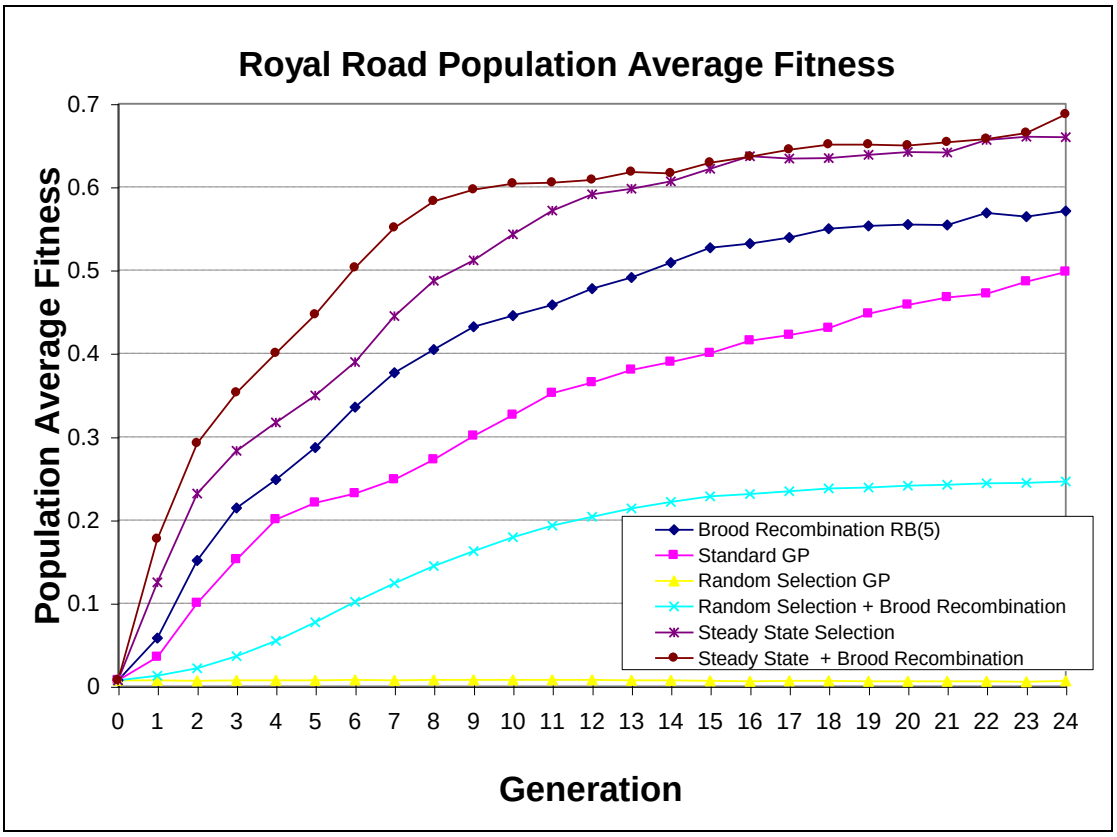


Figure 6.4(a): Mean Fitness of 6 selection and recombination policies tested against the "Royal Road" problem.

6.4 Results

Each of the six breeding policies is run twenty times with the same set of twenty different random seeds. This results in the same twenty initial populations for all breeding policies with the exception that runs using $R_B(n)$ have half the initial population size of the non-$R_B(n)$ runs, and so are a proper subset. Each curve in the comparative performance plots shown here is a composite taken over the 20 runs.

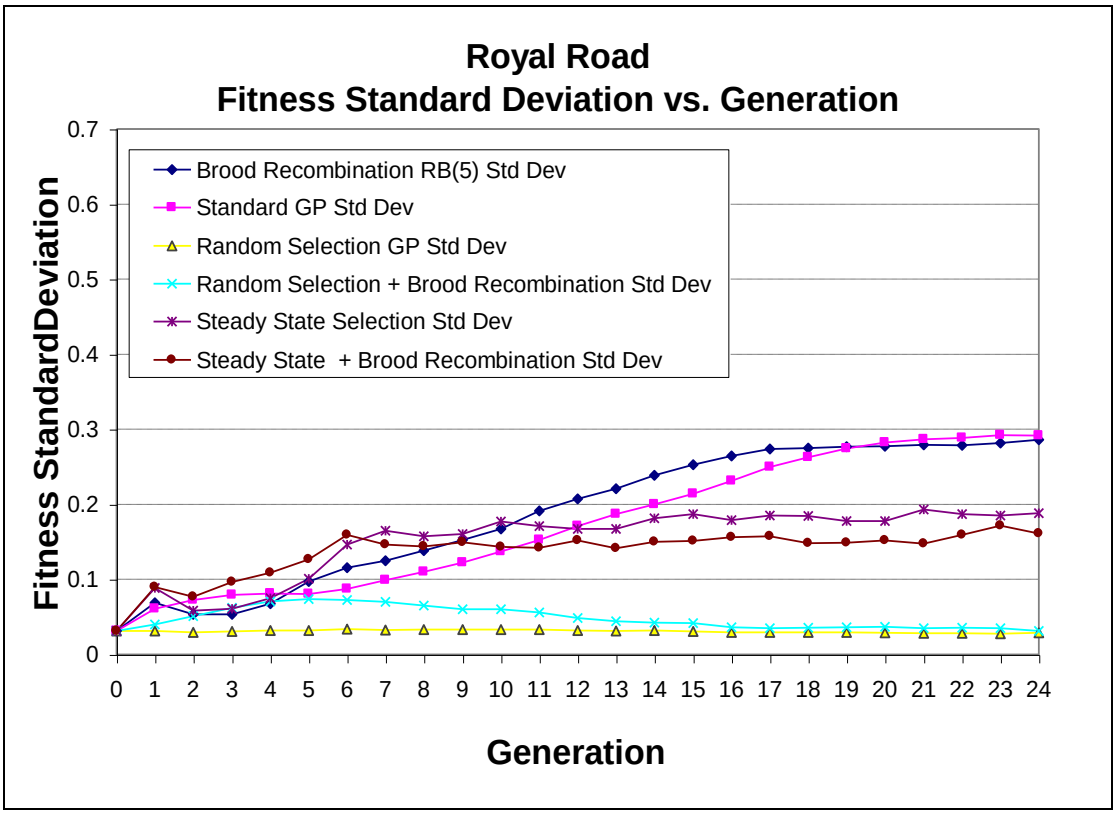


Figure 6.4(a): Fitness Standard Deviation of 6 selection and recombination policies tested against the "Royal Road" problem.

6.4.1 Cumulative Probability of Success

This measure was popularized in (Koza 1992) and is used as a standard measure of relative difficulty for problems which possess an absolute criteria for "success," i.e. a known optimal fitness value. Cumulative probability of success (CPS) is defined for a given generation as the fraction of runs performed which found an individual with optimal fitness on or before that generation (in this case for example, CPS must increment in steps which are multiples of 0.05 since there are 20 runs). CPS vs. Generation for each breeding policy against the Royal Road problem is shown in Figure 6.2, while CPS for the Deceptive Problem is shown in Figure 6.3. In all cases, $R_B(n)$ appears to speed convergence by a small amount, but for the case of the Royal Road all methods using non-random selection achieve roughly equal performance. The deceptive problem provides a different story: $R_B(n)$ with generational selection finds solutions more frequently than other methods. An interesting illustration of the exploratory power of $R_B(n)$ is provided by looking at the curves for random selection: whereas random selection by

itself never finds a perfect solution, random selection under $R_B(n)$ does. As a matter of fact, in generation 24 of the Deceptive problem it has the second-best performance! Also note that the random-selection curves are the same for both problems: fitness values do not affect random selection, and F_B is the same for both. Further clues about the exploratory powers of $R_B(n)$ can be gained by examining moment statistics in the next section.

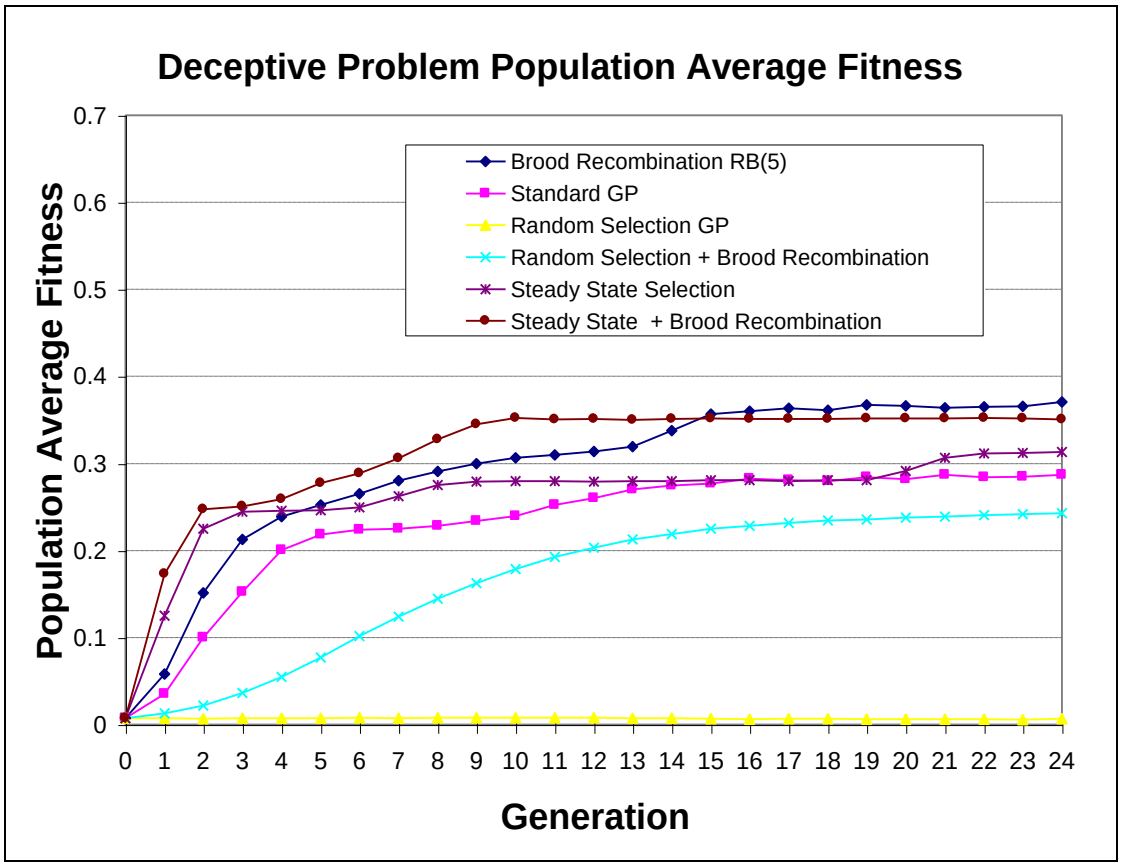


Figure 6.5(a): Mean Fitness of 6 selection and recombination policies tested against the "Deceptive" problem.

6.4.2 Moment Statistics of Population Fitness Distributions

According to Fisher's fundamental theory of natural selection (Fisher 1930; 1958) the ability of a population to increase in fitness is proportional to the variance in fitness of the population members. In studying exploration and evolvability properties due to $R_B(n)$ and selection operators it is of interest to examine the related statistics, particularly the mean and standard deviation of population fitness over time.

Figures 6.4(a) and 6.4(b) show population mean fitness and standard deviation respectively for the Royal Road problem. Curves for standard deviation vary strongly with selection method for this problem, and $R_B(n)$ appears to have a small but usually positive effect. Steady-state breeding policies show the highest mean fitness, closely tracking the corresponding CPS curves: this indicates that once successful individuals have been constructed they rapidly dominate the population. Correspondingly, standard deviation is lower than for generational selection.

Figures 6.5(a) and 6.5(b) show population mean fitness and standard deviation respectively for the Deceptive problem. Standard deviation is still most strongly dependent upon method of selection. The highest mean fitness and the greatest fitness variation are both achieved under $R_B(n)$.

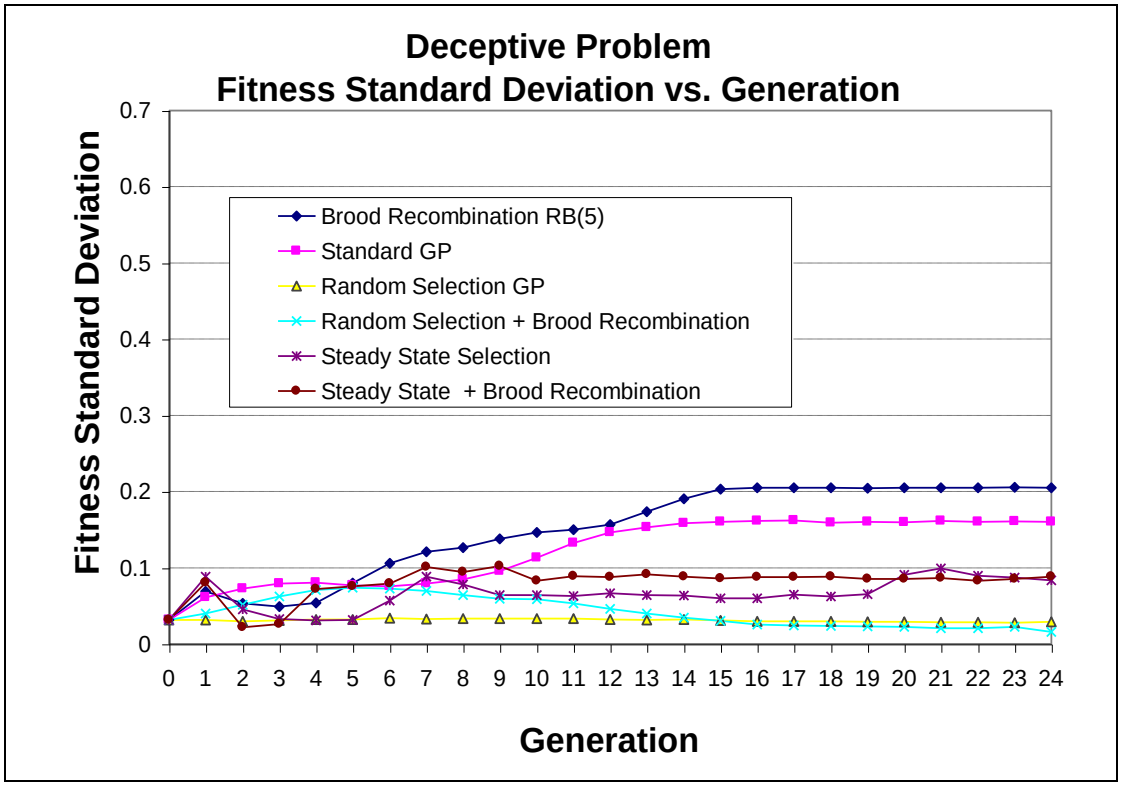


Figure 6.5(b): Fitness Standard Deviation of 6 selection and recombination policies tested against the "Deceptive" problem.

6.4.3 Structural Diversity

The final measurement made here is *structural diversity*. This describes the frequency with which new unique building blocks are formed as evolution proceeds. A list is made of all trees

and subtrees ("schemata") occurring in the population based on an exhaustive search. This list is maintained from generation to generation so that those schemata which are extinct (i.e. those which occur in the previous generation but not in the current generation) are removed from the list. A count is made of those schemata which are new to the list: they may be either entirely new or rediscovered after a previous extinction. Structural diversity is obtained by dividing the number of new schemata produced each generation by the population size. The result is the average number of new schemata per population member, which is an effective measure of the rate at which a particular breeding policy explores the space of program structures.

Figure 6.6 shows structural diversity for the Royal Road. Under random selection, $R_B(n)$ appears to produce new schemata at an increasing rate. Under random selection without $R_B(n)$ diversity slowly decays, presumably due to the effects of genetic drift. For other methods diversity continually decays with time as "perfect" individuals are formed and proliferate throughout the population. This is due to the fact that target pattern T_6 requires that no other schemata be present in the matching individual, and so its proliferation inhibits diversity. The intermediate targets T_1 through T_5 may appear within any greater structure and do not penalize structural diversity. Note that all methods have an artificially high number of new schemata in generation 0 since *all* are new at that time.

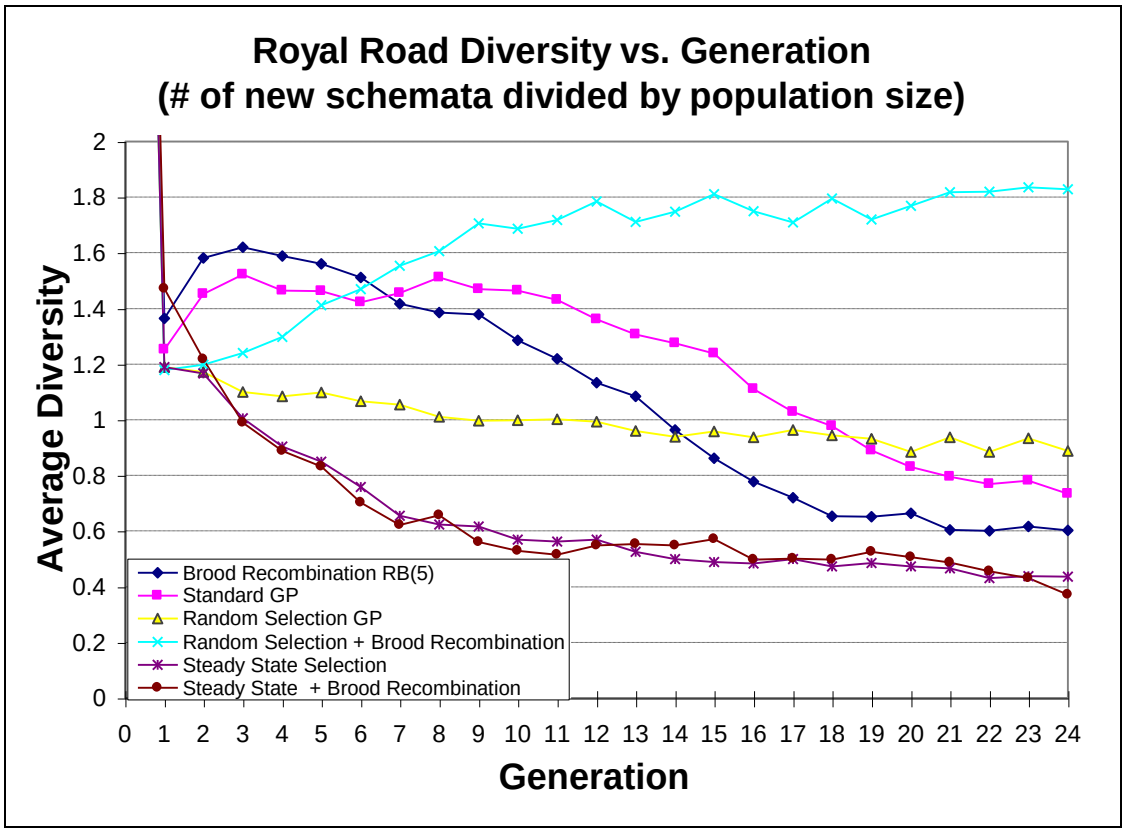


Figure 6.6: Structural Diversity for the Royal Road problem.

Figure 6.7 shows structural diversity for the deceptive problem. The curves are again the same for random selection. Generational selection has greater diversity than steady-state methods as might be expected, and all non-random selection methods have higher diversity for the deceptive problem than for the Royal Road. This is most likely due the fact that far fewer perfect individuals are encountered, therefore minimizing selection against diversity.

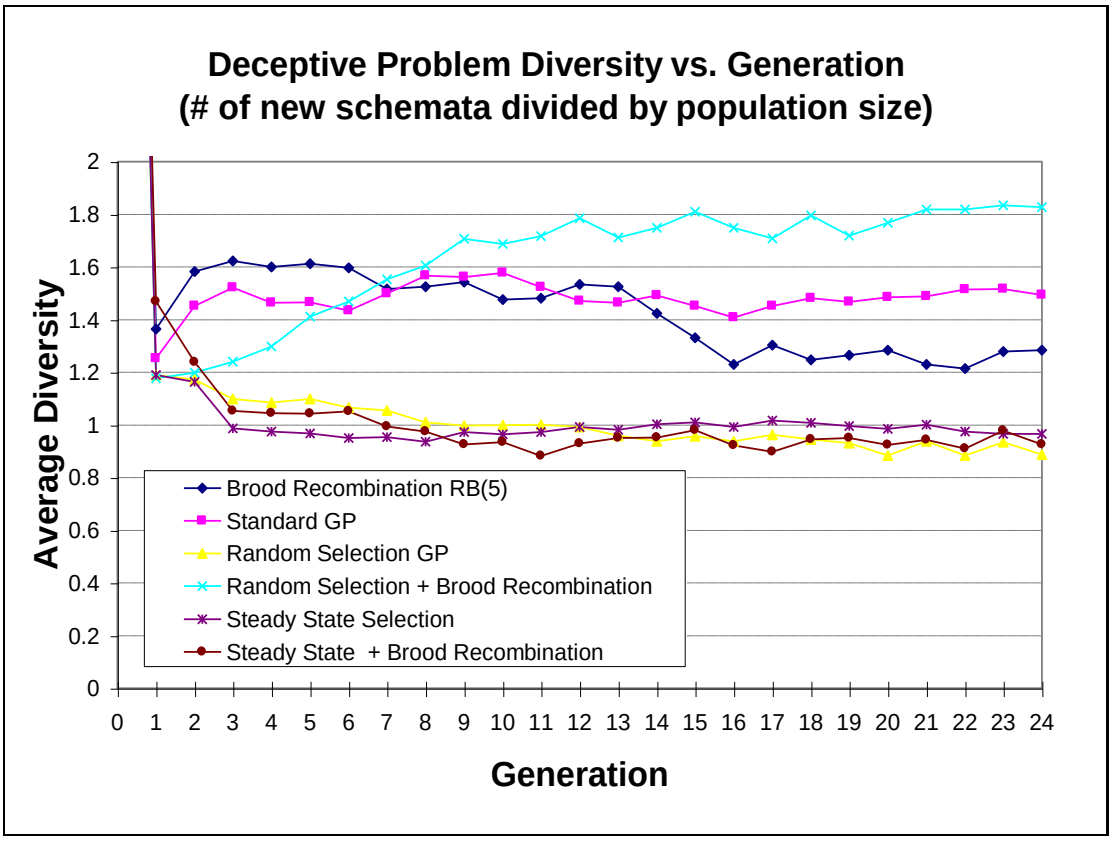


Figure 6.7: Structural Diversity for the Deceptive Problem.

6.6 Discussion

Although the problems tested are limited in many ways, the general class of constructional problems offer a new and more systematic means of exploring the genetic construction of programs than does semantic evaluation. The fact that the tests run here clearly illustrate properties of steady-state selection already observed both in Chapter 4 and by others (Goldberg 1991; DeJong 1993) provides some optimism that the results obtained for $R_B(n)$ may be general as well.

Altenberg's conjecture concerning increase in upper and lower tails of the fitness distribution gains validation, since in most cases the standard deviation indeed is increased under $R_B(n)$. Likewise, the mathematical prediction made in Section 5.4 concerning improved performance when parent and offspring distributions are uncorrelated holds for the correlated case. This is well illustrated by the cumulative probability of success (CPS) results obtained for $R_B(n)$ under random selection (Figures 6.2 and 6.3). That experiment isolated the fitness effects dues to

$R_B(n)$, thereby demonstrating that it is much more likely to produce fit offspring. Comparing greedy recombination with greedy selection we see that $R_B(n)$ under generational selection retains greater structural diversity than do steady-state, i.e. greedy selection, methods.

The results obtained here strongly indicate that $R_B(n)$ is an *exploratory* operator. This may be initially inferred on the evidence of very large structural diversity under $R_B(n)$ and random selection. However, there is something more subtle and significant happening in the structural diversity plots of Figure 6.6 and Figure 6.7, namely a nonlinear interaction between selection and recombination. In both problems, generational selection under $R_B(n)$ provides greater structural diversity than generational selection without $R_B(n)$ for the first eight generations or so. This is an *exploratory phase* in which highly fit expressions and subexpressions - building blocks - are discovered. The exploratory phase is followed by an exploitative phase, where selection takes advantage of these building blocks, allowing them to proliferate. During this phase, structural diversity of generational selection under $R_B(n)$ drops below that without $R_B(n)$. Throughout this process, the fitness under $R_B(n)$ is greater.

Chapter 7

Gene Banking and Trait Mining

Does the Symbolic Nature of Genetically Induced Programs Allow the Knowledge They Acquire to be Understood by Humans?

The work presented so far has focused on understanding and improving the power of genetic program induction. In this final chapter we take a different turn, investigating ways in which the symbolic representation of programs can be exploited. The goal is that knowledge learned by the artificial system can be understood a posteriori by the user. Specifically, we explore ways in which programs constructed under genetic search can be mined for traits in the form of subexpressions which illustrate salient problem elements and the relationships between them. A prerequisite is that all subexpressions generated over time must be kept track of in a "gene bank" which records their emergence, extinction, and distinguishing properties. We demonstrate the use of "gene banking" and "trait mining" in extracting critical subexpressions from the Royal Road problem introduced in Chapter 6 and the Donut induction problem of Chapter 4.

7.1. Introduction

A criticism of connectionist learning by the AI community is that neural methods are opaque with respect to explaining what they have learned about a problem. Alternatively, at least one approach to machine learning has been developed which produces parsimonious symbolic rules of induction through hill climbing (Quinlan 1990). Related work has shown how such learning may be readily integrated with preexisting expert knowledge (Pazzani and Kibler, 1990). Genetic Programming is likewise a symbolic method of induction, and so has potential to feed symbolic knowledge about what it has learned back into the user environment. Among the potential advantages are that genetic search may be more powerful than other methods applied

in symbolic learning to date. An important challenge which is illustrated in Chapter 3 (see Figure 3.3) and in other studies of Genetic Programming such as (Angeline and Pollack 1994) is that genetically induced programs do not yield readily to inspection for many problems.

Section 3.5.2 presented a solution generated by Genetic Programming which "discovered" contrast measures as an important feature for image discrimination. It was also pointed out, however, that (1) the solution under analysis was atypically parsimonious and (2) the recognition of these features as important was made by a human observer with prior knowledge of image processing techniques. The goal of automating the extraction of knowledge requires that information can be extracted from programs regardless of their size or complexity, and that an estimate of the importance of features can be made without prior human knowledge. The "Gene Banking" approach taken here records all subexpressions in the population of each generation along with salient statistics such as frequency of occurrence, fitness mean and variance, time of first creation, and time of extinction. By applying a carefully tailored scheme of hashing and indexing the time and space for doing this is held to a manageable level. After the last generation is complete these traits and their statistics can be evaluated, or "mined," and reported in order of their saliency.

One previous approach to gene banking is provided by the Tierra system (Ray 1991). There, the author creates hash codes which are based on the sequence of assembly language instructions comprising a program. Since Tierra is primarily an ecological model populated by self-replicating programs, there is no explicit measure of fitness. The primary measures of interest are frequency of occurrence for a given type of program, the program size, time of creation, and time of extinction. Programs are not subdivided into traits in the manner described in this chapter. Another interesting precursor is Angeline's GLIB (Genetic LIBrary) system (Angeline 1993; Angeline and Pollack 1994). GLIB is an extension of Genetic Programming which addresses the problem of automatic subroutine generation. It proceeds by randomly extracting subexpressions from programs in the population and reformatting them into parameterized functions, called *modules* by the author, which are stored in a library. The modules extend the function set from which programs are generated and may also be passed on to ancestors of the programs from which they were originally extracted. The frequency with which they occur in the population is tracked, and they may be removed when no longer in use

by any population member. By contrast, the gene banking approach described here stores all complete subexpressions without parameterization, and does not affect the function set or exert any other external influence over the course of evolution.

7.2. The Size Problem

A phenomenon commonly observed by practitioners of genetic programming is that the average size of expressions in the population grows with time. Many theories have been proposed to explain this phenomenon, which is commonly referred to as *bloating*, or *the size problem.* The "defense against crossover" hypothesis (Singleton 1993) suggests that it is evolutionarily advantageous for programs to insert "introns" in order to protect critical sections of code. That theory implies that Greedy Recombination should reduce the average size of programs, since recombinations that are disruptive to important code sections would tend not to survive among the brood and would reduce the pressure to create "decoy" code. On the opposite end of the spectrum, (Altenberg 1994b) conjectures that bloating may be a neutral phenomenon with respect to selection pressure, caused by asymmetries of the recombination operator. This in turn implies that bloating should occur in the absence of selection pressure. Although both of those theories seem plausible, Figure 7.1 shows evidence to the contrary. Specifically, the six combinations of selection and recombination which were studied in Chapter 6 demonstrate average growth in size which is proportional to selection pressure. In examining Figure 7.1, it is best to consider the first few generations only: growth is artificially forced to taper off beyond that point first by hard limits on program size (specified as an input parameter) and also by the fact that the optimal solution has a fixed size and does not admit extraneous code sections. The defense against crossover would imply that the growth curves for methods employing greedy recombination would be less steep than for those which do not. During those first few generations, where the bloating phenomenon is the only force affecting size, the exact opposite occurs. Figure 7.1 also shows that the only time bloating does not occur is when random selection is coupled with random recombination: in other words, when fitness is completely ignored. Therefore, bloating is not selection-neutral. We suggest that, based on evidence to be presented later on, bloating may be tied to a phenomenon known as "hitchhiking," and is the product of recombination acting in concert with selection. A subexpression S may be considered *atomic*, so that by removing or changing any of the subexpressions which comprise it

will result in lower fitness of S. The subexpression S may, however, be embedded in any of an infinite family of greater expressions which are *inert*: they contain S, but perform no actions nor return any values which differ from the results due to S. Therefore, recombination events which transmit S intact are more likely than not to result in the transmission of extra material.

Regardless of the source of bloating, it is clearly the primary challenge to the trait-mining process. The methods explored in this chapter therefore seek to decompose the programs of the population into their constituent subexpressions, and to detect salient traits by discovery of their distinguishing features. This is a formidable task of accounting, requiring careful design in order not to exhaust computational resources.

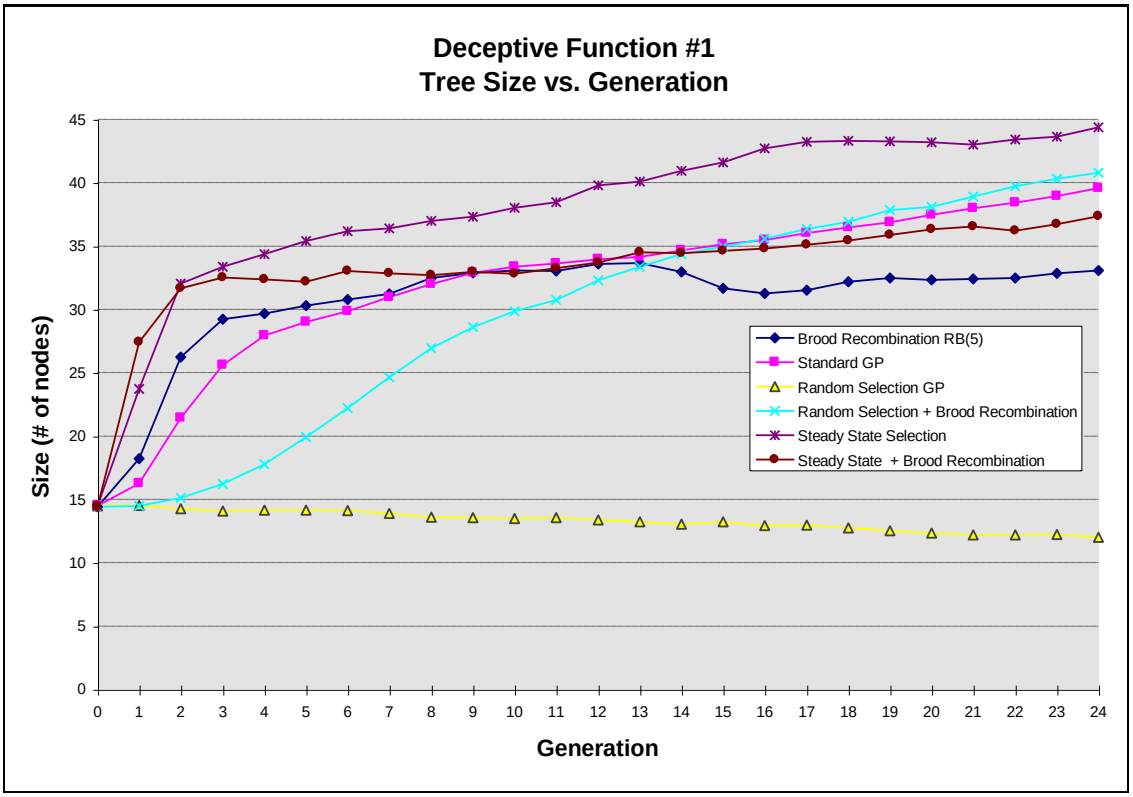


Figure 7.1: The size problem illustrated for the methods tested in Chapter 6. The average size of expressions grows at a rate proportional to selection pressure for the first two or three generations. Growth tapers off somewhat in later generations only because of depth limits and because the optimal solution discourages extraneous code.

7.3. The Gene Banker's Algorithm

Chapter 6 introduced a measurement of structural diversity on the population which was defined as "the frequency with which new unique building blocks are formed as evolution proceeds...based on an exhaustive search." Although it was not discussed in detail at the time, the exhaustive search of all subexpressions in the population can be a formidable task. Consider that the number of whole subexpressions in an expression is equal to the total number of function nodes and terminal nodes which it contains. Then the number of subexpressions which must be counted and processed is:

7.1 $$\#\text{ of Nodes} = \sum_{j=1}^{N} SIZE_j$$

In the problems studied here the population size N can range from hundreds to thousands, while the count of nodes in an individual given by SIZEj can range from tens to hundreds. Also, each expression must be compared to all other expressions in order to determine uniqueness and tabulate statistics over multiple occurrences. In the worst case it must be matched against all subexpressions in the population except itself. If, on average, some fraction *A* of the nodes in each expression must be compared against those of other expressions in order to determine a match, then the number of comparisons required is:

7.2 $$\#\text{ of comparisons} = A\sum_{i=1}^{N} SIZE_i \sum_{j=1}^{N} SIZE_j$$

In a typical situation the number of nodes in a population can number in the tens of thousands, and so the number of comparisons can run into hundreds of millions. The Gene Banker's Algorithm hashes all expressions and subexpressions occurring in the population using time linearly proportional to the number nodes. This is done by recursively assigning a hash code to a complete subexpression based on the hash codes of its root node and its immediate children. Doing this requires traversing each expression only once, although in practice many elaborate statistics are more easily gathered in a two-pass process. The kernel of the population hashing process is the function hash_one_tree(), which recursively computes hash codes for subexpressions and inserts information into a hash table. It can be stated in pseudocode form as follows:

```
integer function hash_one_tree(nodetype Node, integer generation, integer size, integer depth,
real fitness)
integer                 i
integer                 h
integer                 mysize, mydepth

begin
mysize = mydepth = h = 0
if (type_of(Node) == FUNCTION)              /* then compute hash codes of children first.... */
        for i=1 to arity(Node)              /* iterate over children... */
                /* Hash code is weighted sum of hash codes of children.  Hashprimes is an array of
large
                   prime numbers.  If we do not weight the child nodes then the hash code is
                   commutative. This way (+ A B) has a different code from (+ B A).  Note that
                   child_of(Node, i) means the i'th child of Node. */
                h = h + (hashprimes[i] * hash_one_tree(child_of(Node, i), generation, size, depth,
fitness)))
                h = h mod 277218551   /* prevent overflow of hash code */
                mysize = mysize + size   /* size is cumulative */
                mydepth = max(depth, mydepth);   /* depth is based on deepest child branch */
        end for
        size = 1 + mysize          /* Size of subexpression rooted at Node includes itself. */
        depth = 1 + mydepth     /* Depth of subexpression rooted at Node includes itself. */
        /* Hash code is adjusted to account for the operator represented by node.  This way (+ A B)
has a
           different code from (* A B). */
        h = (h + identifier_of(Node) * 21523) mod 277218551
else if (type_of(Node) == TERMINAL)
        size = 1
        depth = 0
        /* generate a unique hash code for each constant */
        if (identifier_of(Node) == CONSTANT)
                h = (constant_value_of(Node)*21929.0) mod 277218551
        else   /* Node is a terminal but not a constant- it is a variable */
                h = identifier_of(Node)
        endif
endif
insert_hash_table_entry(h, generation, size, depth, fitness)
return(h)
end /* hash_one_tree */
```

The hash_one_tree() algorithm recursively traverses down a tree, assigns hash codes to the leaves, and then computes the hash codes of parent expressions on the way back up. The hash code is a sum of prime numbers weighted by the hash codes of children and by the numeric

identifier associated with the node. Numeric identifiers are unique ordinal numbers assigned to the functions and terminals: for example, the variable 'A' might have the identifier 1 while the variable 'B' has the identifier 2 and the function '+' has the identifier 6. Once hash codes are computed, the function insert_hash_table_entry() updates statistics. Statistics are maintained in a list, or "bucket" of the hash table. The index of the bucket within the hash table is just the hash code modulo the size of the hash table. By using a large number of buckets and choosing the prime multipliers so as to uniformly distribute entries within the table, the list maintained in each bucket may be kept relatively short. The call to insert_hash_table_entry() searches the bucket to see if any pre-existing hash codes match the hash code being inserted. If so, then the size and depth of the new entry and existing entry are compared. If these match, then the expressions themselves are matched for final verification. When a match is found, the statistics associated with the subtree are updated. If a match is not found then a new entry is created and the statistics are initialized. Note that although only generation, size, depth, and fitness are listed in the pseudocode above, many more attributes are recorded in practice. The subexpression itself must be recorded as well, but this is not shown in order to eschew extensive details mainly pertinent to memory savings. It is sufficient to say that hash table entries for rooted expressions contain a "master" copy of the expression and the entries for subexpressions which it contains reference that master copy. Another important method of storage management is garbage collection: after all expressions in a population have been hashed, the table can be searched for entries which were not updated in the current generation. Those which were not have gone extinct and may be removed.

As the run proceeds a report on subexpressions and their pertinent statistics is generated for post hoc processing. For this purpose only the subexpressions which are not "subsumed" will be printed out. We define subsumption of an expression to mean that it occurs only as part of a single larger expression. For example, if the expression (+ A B) occurs only within the larger expression (* C (+ A B)) then only the larger expression is reported. If (+ A B) is also contained in one or more other expressions, or occurs by itself, then it is not subsumed and will be reported as a separate entity.

7.4. Methods for Ranking Saliency of Traits

In Chapter 5 the concept of canonical fitness was introduced. An idea key to that analysis was that some subexpressions may confer greater fitness to the expressions which contain them than do others. In ranking traits to determine their contributions to performance we can begin by asking, "What is the expected fitness of an expression P given that it contains subexpression S_i?" The conditional expected fitness is stated mathematically by the equation:

(7.3) $$w_S(S_i) = \mathbf{E}\{w(P) \mid S_i \in P\} = \mathbf{E}\{w(P, S_i)\}$$

where w is the fitness of programs, **E** is the *expected value* operator, and $w_S(S_i)$ is the average fitness of all members of the population which contain subexpression S_i. This figure is an interpretation of Holland's "Schema Fitness" (Holland 1975, 1992) applied to Genetic Programming. According to Holland, those traits which confer high fitness can be expected to propagate exponentially throughout the population as generations progress. Traits which confer greater-than-average fitness will cause the expressions which contain them to reproduce at a greater-than-average rate, and thus the frequency of a fit trait is compounded over generations. The exponential growth rate will be limited in practice since the average fitness of the population increases and dilutes the relative advantage of the trait. In general it can be expected that traits which contribute to fitness of expressions will have a high schema fitness and high frequency of occurrence in the population. Saliency measures based on schema fitness and frequency can be confounded, however, since the converse is not necessarily true: although salient traits should have high schema fitness and high frequency, high fitness and frequency may not necessarily imply salience. It is possible that a useless section of code H in a subexpression S_i can be replicated and passed on to offspring along with highly fit code, so that H occurs uniquely in highly fit individuals. The computed conditional fitness of H will be high even though it actually makes no contribution. This phenomenon, known as hitchhiking, has been observed in studies of bit-string genetic algorithms (Schraudolph & Belew 1990; Das & Whitley 1991; Forrest & Mitchell 1993). An important goal of the work presented here is to determine the degree to which hitchhiking prevails in Genetic Programming.

A hitchhiker may initially achieve high conditional fitness, but as time goes on it may be expected to diffuse among individuals which are not fit. When this happens, the conditional

fitness for the hitchhiker should decay. As a general principle we consider that rather than make static measurements, a great deal of information may be derived from examining time-dependent statistics. Although promising, such measurements can be difficult to automate since they require identifying and classifying the time-varying waveforms associated with fitness.[1] Some simpler examples of time-based statistics are the time of creation for a trait and its longevity, which is just the number of generations over which it survives. These can be expressed as scalar quantities which are readily interpretable: for example, highly fit traits that occur the earliest are less likely to be hitchhikers simply because there are no pre-existing highly fit traits for them to "hitch a ride" with.

In turn, the time-of-creation measure suggests another general category of saliency measures, namely those derived by commonsense reasoning. For example, when ranking traits there are considerations such as subexpression size or "parsimony": given a group of traits which are equal in fitness and frequency, those which are simpler may be more useful and easy to comprehend.

We have identified three categories of saliency measures which can be labeled as statistical, time-dependent, and commonsense (or *heuristic*). These classes are not mutually exclusive, and the examples discussed only hint at the large number of possibilities. More detailed measures could be made, for example, by computing information measures which examine how individual traits affect partitioning of test data as suggested by (Quinlan 1990). Other possibilities include elaborate heuristics that accurately capture commonsense reasoning about saliency. Since the goal of this chapter is to better understand the problems involved, the approach taken is a relatively simple one. The experiments will focus on statistical measures of conditional fitness and frequency of occurrence, and provide some graphical evidence concerning the time-dependent properties of those statistics. Saliency is determined by sorting on scalar quantities. Subexpressions are primarily sorted based on the peak fitness which they achieve during the run. Ties are broken by sorting on frequency as a secondary criteria. Ties in the first two criteria are broken by earliest time of creation, and finally by expression size.

[1] This does, however, raise the interesting possibility of using Genetic Programming for induction on fitness waveforms to determine whether they are salient features or hitchhikers, i.e. "Genetic Programming Applied to Genetic Programming."

7.5. Experiment 1: The Royal Road

Chapter 6 introduced the class of "constructional problems." In those problems the credit for partial solutions is precisely controlled in order to control problem complexity. Since the true worth of each subexpression is known, the constructional problems are also ideal for studying the principles of saliency extraction. This experiment will examine the Royal Road problem for Genetic Programming illustrated in Figure 6.1. In that problem there are exactly five expressions which receive credit, and one of those five can receive two different possible scores depending on the context in which it appears. All runs examined were performed using the standard selection method and greedy recombination.

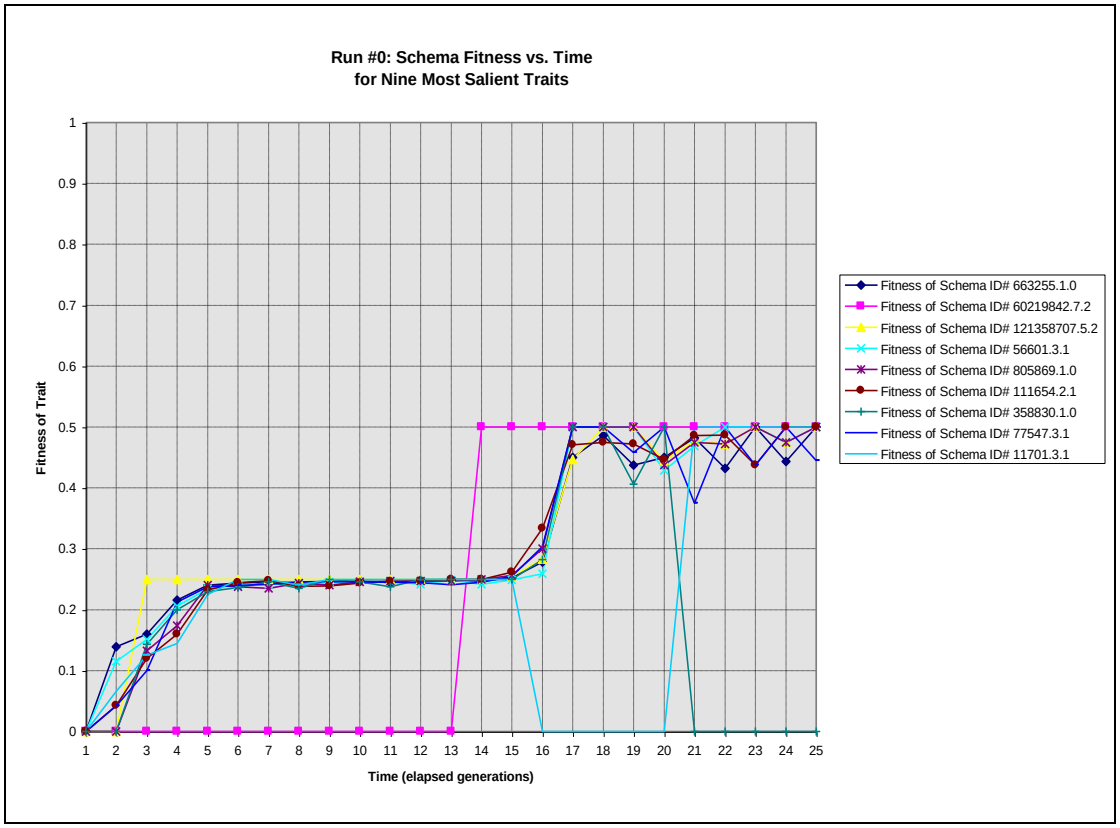


Figure 7.2: Fitness vs. Time for nine most "salient" traits of Royal Road run #0. This run never found the optimal solution in 25 generations, and so there are no traits which have a fitness of 1.0.

7.5.1. Royal Road Run #0

The first of these runs, which we will designate run #0, never achieved the optimal solution. Figure 7.1 illustrates fitness of the nine most salient subexpressions vs. time for that run, while

Figure 7.2 shows the corresponding measures of Frequency of Occurrence vs. Time (note that Frequency of Occurrence is abbreviated as FoO in the diagrams for brevity). The identifiers for each individual are given in the form <*hash code*>.<*size*>.<*depth*>.

Generation 0 produced 1791 unique subexpressions from a population size of 250. Subsequent generations produced an average of about 500 new (unique) subexpressions per generation. Out of these, a total of 1555 passed the criteria for consideration as salient expressions. Table 7.1 lists the nine most salient traits in their printed form along with their hash codes.

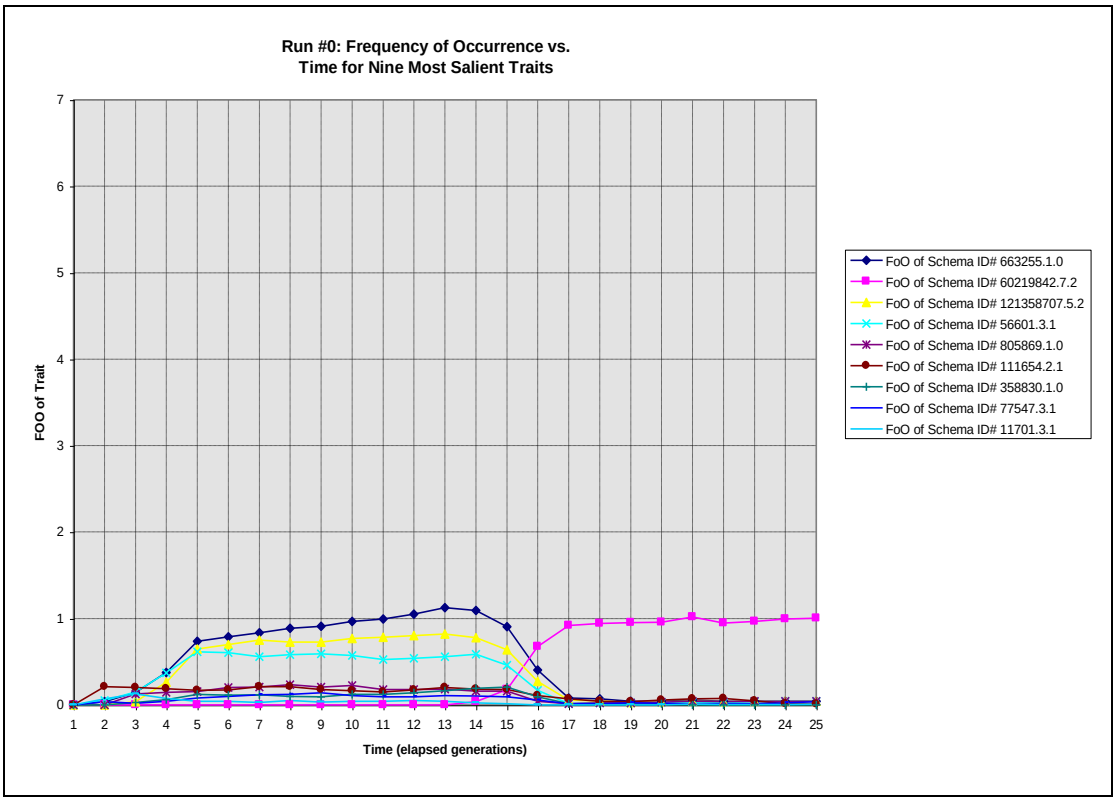


Figure 7.3: Frequency of Occurrence vs. Time for the nine most salient traits of run #0. The optimal solution was not found during this run.

Table 7.1: The nine most salient features estimated for Royal Road run #0.

1) Schema ID# 663255.1.0 peak_fitness= 0.500000 MaxFoO= 1.124000 3.360636

2) Schema ID# 60219842.7.2 peak_fitness= 0.500000 MaxFoO= 1.016000

```
(BAZ
        (BAR
                Z
                X)
        (FOO
                X
```

```
                Y))

3) Schema ID# 121358707.5.2 peak_fitness= 0.500000 MaxFoO= 0.820000
(BAZ
        3.360636
        (FOO
                X
                Y))

4) Schema ID# 56601.3.1 peak_fitness= 0.500000 MaxFoO= 0.612000
(BAZ
        Z
        Y)

5) Schema ID# 805869.1.0 peak_fitness= 0.500000 MaxFoO= 0.236000
4.083253

6) Schema ID# 111654.2.1 peak_fitness= 0.500000 MaxFoO= 0.212000
(NOP_1
        Y)

7) Schema ID# 358830.1.0 peak_fitness= 0.500000 MaxFoO= 0.204000
-1.818142

8) Schema ID# 77547.3.1 peak_fitness= 0.500000 MaxFoO= 0.140000
(BAR
        Y
        Y)

9) Schema ID# 11701.3.1 peak_fitness= 0.500000 MaxFoO= 0.128000
(FOO
        Y
        X)
```

According to these statistics, hitchhiking is rampant. Out of the nine most salient traits as measured by our simple criteria, only two - numbers (2) and (3) in the ranking - had any true worth at all, and they were ranked lower in saliency that one hitchhiker. There are temporal clues, however, which point out that (2) (id# 60219842.7.2) and (3) (id# 121358707.5.2) are salient. Examining Figure 7.1, subexpression (3) is the first trait to have a fitness of 0.25, which it achieves as soon as it appears. Other traits eventually achieve the same fitness, but with some lag, as they slowly find their way into expressions which contain truly fit subexpressions. The same is true for subexpression (2), which achieves a schema fitness of 0.5 three generations before any other. Recall that these temporal clues were predicted in Section 7.3. Another important temporal cue concerns correlation between fitness and frequency of occurrence. The rankings here were made using peak values of fitness and frequency for each trait without any correlation between the two measures. Comparison of Figures 7.1 and 7.2 reveals that for hitchhikers these two peaks must have occurred at different times. Specifically, peak fitnesses of hitchhikers appear to have been achieved by a very small number of instances

at a time late in the run when the hitchhiking trait was almost extinct. The only trait showing strong correlation between fitness and frequency curves is (2), which is the most truly salient. Clearly these simple means of ranking traits are imperfect and require some refinements, such as automating the analysis of the temporal features described above. To understand how well they are doing, it is helpful to examine the "raw" form in which fit traits occur. Table 7.2 shows the first, and probably the least complex, individual in the population which contains the salient trait (2). The complexity of this expression serves to illustrate that even the methods described here are doing a reasonably good "first cut" at information filtering.

Table 7.2: A typical highly fit expression is much more complex than the list of ranked traits.

```
(NOP_1
        (NOP_4
                (NOP_1
                        Z)
                (BAZ
                        (BAR
                                Z
                                X)
                        (FOO
                                X
                                Y))
                (NOP_4
                        (NOP_1
                                X)
                        (BAR
                                -2.280125
                                Y)
                        (BAR
                                -1.351968
                                X)
                        (BAR
                                X
                                X))
                (NOP_4
                        0.727543
                        (FOO
                                X
                                Y)
                        (FOO
                                (FOO
                                        (FOO
                                                2.003905
                                                (NOP_4
                                                        (FOO
                                                                -1.543578
                                                                X)
                                                        (FOO
                                                                -0.657315
                                                                Z)
                                                        (FOO
                                                                0.727543
                                                                Y)
                                                        Z))
                                        X)
                                X)
                        Z)))
```

7.5.2. Royal Road Run #1

The next run for the Royal Road located the optimal solution. This creates a situation where the most salient traits should be easy to find, since the optimal solution requires the exact match of a rooted expression and therefore disallows hitchhikers. Although such a situation is unlikely to be encountered in a real problem, it serves to illustrate how well the simple ranking method can be expected to perform in an "ideal" situation.

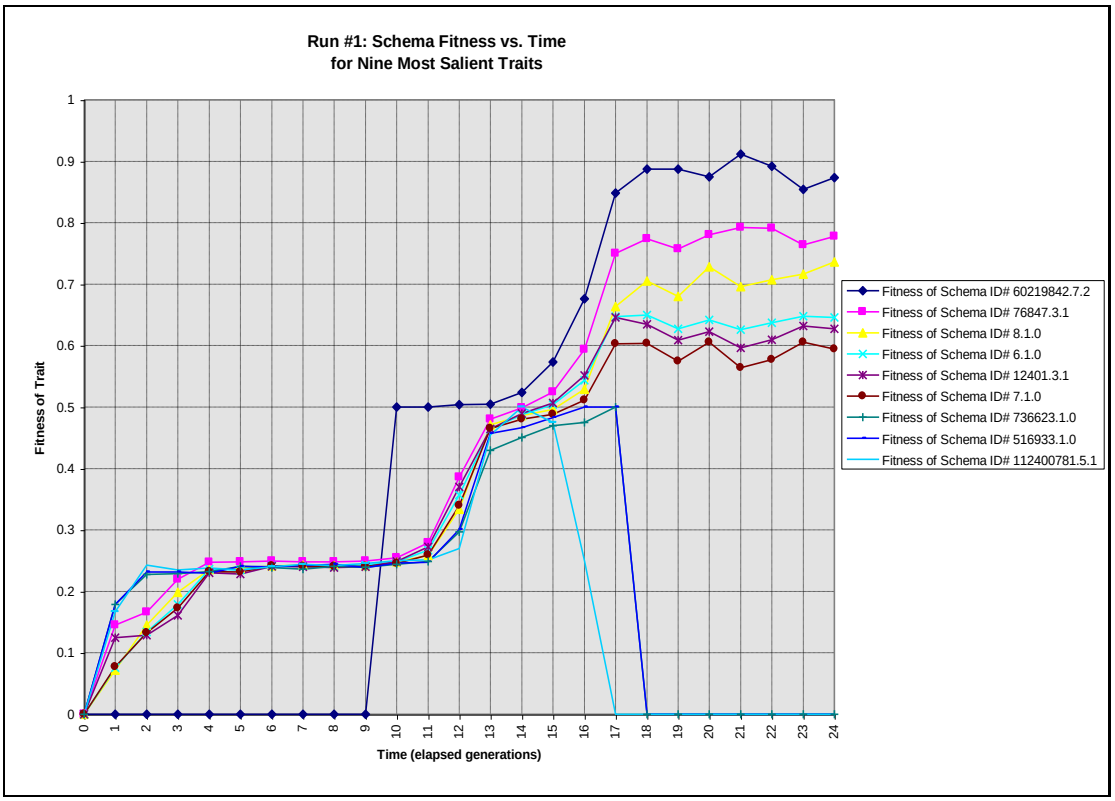


Figure 7.4: Schema fitness vs. time for Royal Road run #1. The six top-ranked traits belong are subexpressions of the optimal solution, with the top-ranked trait being the optimal solution itself.

Generation 0 produced 1660 unique subexpressions from a population size of 250. Subsequent generations produced an average of about 400 new (unique) subexpressions per generation until the optimal expression was found in generation 12. After that event, the number of new subexpressions decreased each generation, reaching 0 in generation 18. Out of the total number produced during the run, a total of 1038 passed the criteria for consideration as salient

expressions. The top nine are shown in Table 7.3 ranked by saliency, while Figures 7.3 and 7.4 show their time-varying measures of fitness and frequency. This time, the optimal solution is top-ranked, and the next five ranked subexpressions are ones that occur in it. Nonetheless, this illustrates another form of hitchhiking: the expressions consisting of the single terminals X, Y, and Z actually have no inherent fitness by themselves, yet receive a large amount of credit. This points out a sort of gray area in the ranking problem, since X, Y, and Z are critical elements of the solution even though they are worth nothing by themselves. Similarly, the expressions (BAR Z X) and (FOO X Y) have estimated fitness values much higher than their true worth of 0.125 each. The reason for this is clear: since the optimal individual is twice as fit as the next fittest individual possible, it can be expected to rapidly dominate the population, so that most members of the population will take the optimal form and have a fitness of 1.0. According to the schema fitness measure given in Equation 7.3, all subexpressions which occur within that form should have an estimated fitness close to 1.0. The expressions ranked 7 through 9 are true hitchhikers like those encountered in run #0. Note that they have fitness values of 0.5, as did highly ranked hitchhikers in run #0.

Examining the time varying properties, it can be seen that the optimal solution is the first to achieve high fitness values, with the fitness statistics of other subexpressions lagging. Also, the hitchhiking expressions can be seen to go extinct between generations 16 and 18 as they are crowded out by optimally fit individuals.

As stated before, this run is less realistic than run #0, since an exact solution which does not allow garbage code will not frequently be encountered. The next experiment adds a stronger measure of reality, by analyzing a problem in which there are inherently no exact solutions to be found.

Table 7.3: The nine most salient features estimated for Royal Road run #1.

```
1)Schema ID# 60219842.7.2 peak_fitness= 0.911565 MaxFoO= 1.004000
(BAZ
        (BAR
                Z
                X)
        (FOO
                X
                Y))

2) Schema ID# 76847.3.1 peak_fitness= 0.791892 MaxFoO= 2.016000
(BAR
        Z
        X)
```

3) Schema ID# 8.1.0 peak_fitness= 0.735931 MaxFoO= 6.988000
Z

4) Schema ID# 6.1.0 peak_fitness= 0.649909 MaxFoO= 5.364000
X

5) Schema ID# 12401.3.1 peak_fitness= 0.645761 MaxFoO= 1.704000
(FOO
X
Y)

6) Schema ID# 7.1.0 peak_fitness= 0.605707 MaxFoO= 5.644000
Y

7) Schema ID# 736623.1.0 peak_fitness= 0.500000 MaxFoO= 0.704000
3.732384

8) Schema ID# 516933.1.0 peak_fitness= 0.500000 MaxFoO= 0.672000
-2.619268

9) Schema ID# 112400781.5.1 peak_fitness= 0.500000 MaxFoO= 0.604000
(NOP_4
Y
Z
-3.067644
Z)

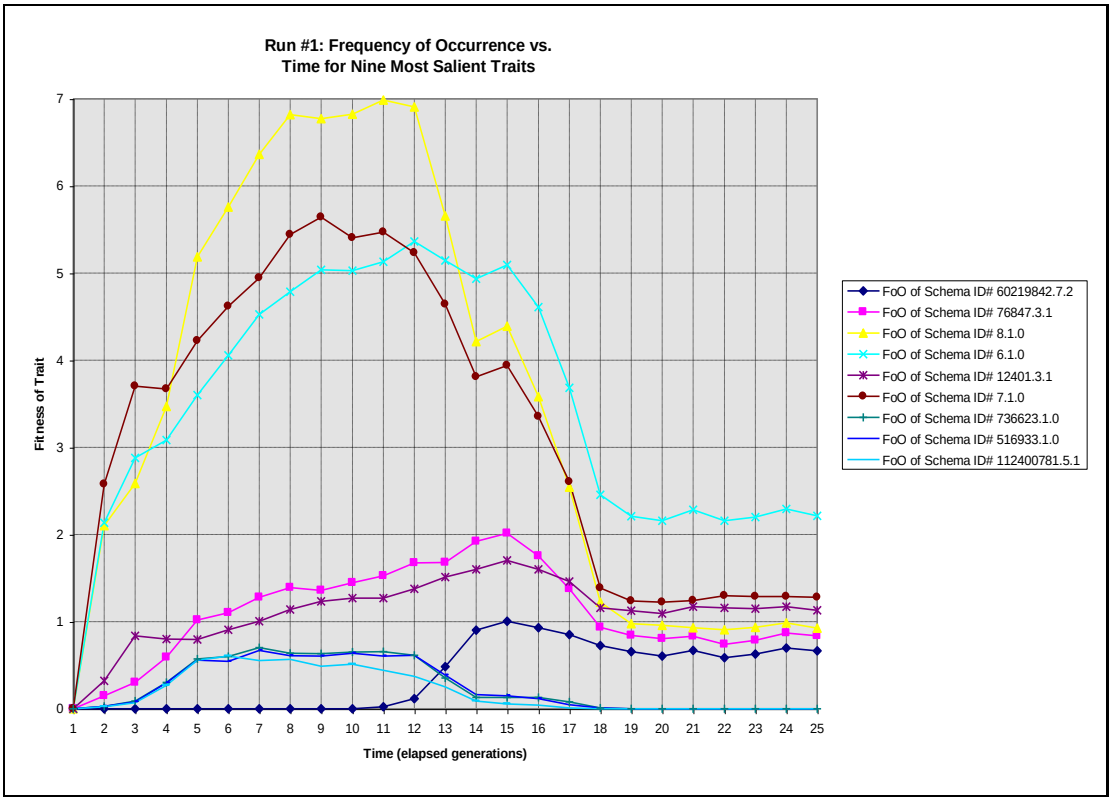


Figure 7.5: Frequency of occurrence vs. time for Royal Road run #1. During this run the optimal solution was found, and the six top-ranked traits are subexpression of that solution.

7.6. Experiment 2: Donut Mining

The "Donut Problem" was introduced in Chapter 4 in order to study selection and generalization, and was revisited in Chapter 5 to study recombination. In formulating that artificial problem several "imperfections" were introduced in order to make it more accurately reflect the dilemmas encountered in real-world induction problems. Among those imperfections were a fitness measure that tended to cause over-fitting to the training data, and a function set which did not include the functions necessary to express the optimal mathematical solution with an expression of finite length. Therefore, the solutions created to the donut problem are at best approximations, and there are likely to be many families of approximate solutions which do equally well but take entirely different forms from each other. Unlike the constructional problems of the previous section, there are no prior known solutions which we can expect to find. This experiment will examine fitness and frequency in order to guess, without verification, which traits might be salient according to the criteria observed in the Section 7.4. The two runs to be examined here use the same parameters as described in Chapter 5, with the exception that only 100 training samples per category are used to determine fitness. A population size of 250 individuals was used, with brood sizes of 10 offspring.

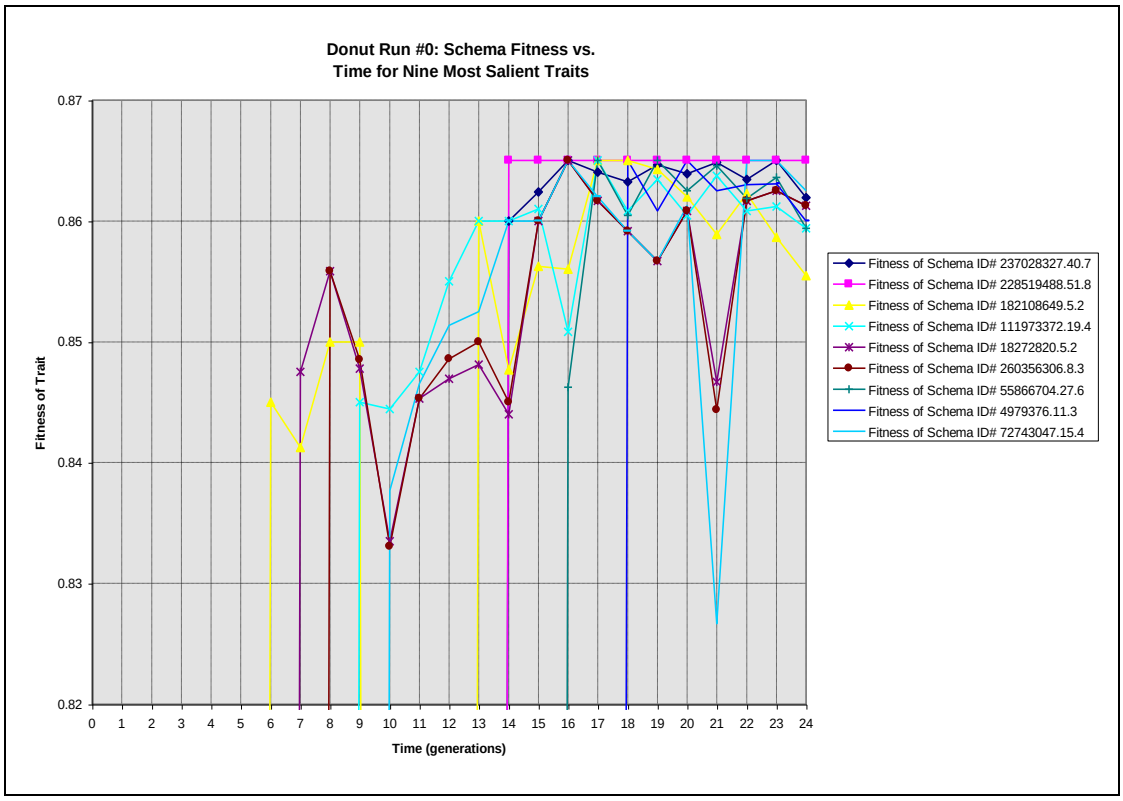


Figure 7.6: Schema fitness for run #0 of the Donut induction problem.

7.6.1. Donut Run #0

Generation 0 produced 1791 unique subexpressions from a population size of 250. Subsequent generations produced an average of about 300 new (unique) subexpressions per generation. Out of these, a total of 874 passed the criteria for consideration as salient expressions. The top nine are reported on here. A careful examination of Table 7.4 reveals a process of "decomposition" is at work, much as with run #1 of the Royal Road. Of the nine traits shown, only three are not contained entirely within some other expression in the list. Specifically, trait (8) is a subexpression of (4), which is a subexpression of (7) which is a subexpression of (1) which is a subexpression of (2). In a second lineage, (6) is a subexpression of (5), which is a subexpression of (9). Only the expression (3) stands alone.

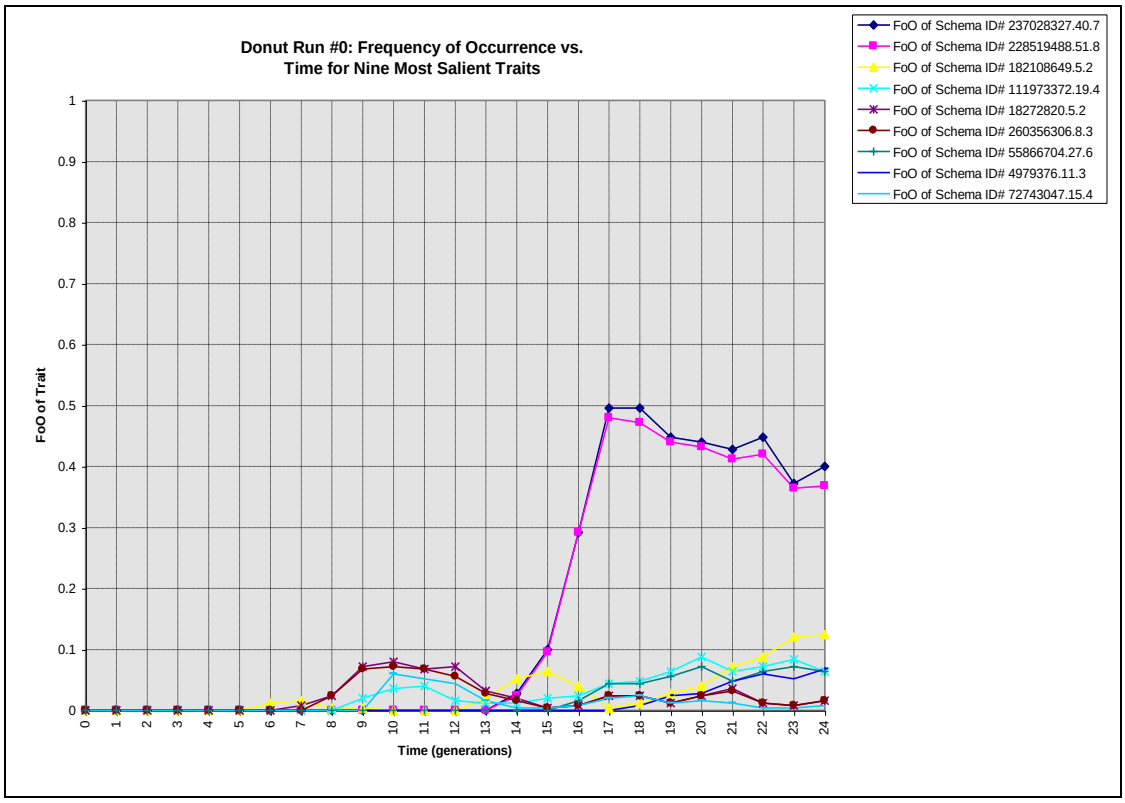


Figure 7.7: Frequency of occurrence for run #0 of the Donut induction problem.

An examination of the population members (not shown here) reveals the fact that fitness increases rapidly in the first couple of generations, and thereafter the fitness of individuals in the population are distinguished by only a few percentage points at most: the competition between individuals is a "close race." This behavior is illustrated in Chapter 4 (the quick rise in fitness is well illustrated in Figure 4.7, for example), [2] and also in Chapter 5, where doubling or tripling the population size results in an average performance increase of 1% or so. This is addressed quite easily by adjusting the scale at which schema fitness values are compared.

The schema fitness curves for expressions (1) and (2) are identical. Those expressions appear in generation 14 with a fitness of 0.865, and remain at that level for the duration of the run. Other expressions, including those which are subexpressions of (1) and (2), display the "lagging" behavior in their fitness curves. Also, there is a strong correlation between the shapes of the fitness curves for (1) and (2) and their frequency-of-occurrence curves depicted in Figure 7.6. According to the observations made in Section 7.4, we should conclude that (1) and (2) are

[2] Note that Chapter 4 figures use a "lower-is-better" fitness criteria so that higher fitness implies values closer to 0.

"truly" salient, and carry the information about which problem elements and relations are required to solve the problem. Examining Figure 7.6, it can also be seen that subexpression (1) occurs more frequently than expression (2) starting in generation 17. This indicates that it is appearing in expressions other than (2). Since its schema fitness does not decrease, this means that those other expressions which contain subexpression (1) are also achieving the maximal fitness value of 0.865, and that subexpression (1) is likely to be the most critical element to the discrimination task.

Table 7.4: Nine most salient subexpressions for Donut problem run #0.

```
1) Schema ID# 237028327.40.7 peak_fitness= 0.865000 MaxFoO= 0.496000
(+
        (-
                (NOP
                        X)
                (%
                        (IFLTE
                                -0.579501
                                Z
                                Z
                                Y)
                        (+
                                Y
                                Z)))
        (-
                (+
                        X
                        -3.268226)
                (IFLTE
                        -0.579501
                        (+
                                (-
                                        (NOP
                                                X)
                                        (%
                                                (IFLTE
                                                        -0.579501
                                                        Z
                                                        Z
                                                        Y)
                                                (+
                                                        Y
                                                        Z)))
                                (-
                                        (+
                                                X
                                                -3.268226)
                                        (NOP
                                                Z)))
                        Z
                        Y)))

2) Schema ID# 228519488.51.8 peak_fitness= 0.865000 MaxFoO= 0.480000
(IFLTE
        (*
                -0.126466
                -4.383145)
        (NOP
                (%
```

```
                    Y
                    (+
                        Y
                        Z)))
        (+
            (-
                (NOP
                    X)
                (%
                    (IFLTE
                        -0.579501
                        Z
                        Z
                        Y)
                    (+
                        Y
                        Z)))
            (-
                (+
                    X
                    -3.268226)
                (IFLTE
                    -0.579501
                    (+
                        (-
                            (NOP
                                X)
                            (%
                                (IFLTE
                                    -0.579501
                                    Z
                                    Z
                                    Y)
                                (+
                                    Y
                                    Z)))
                        (-
                            (+
                                X
                                -3.268226)
                            (NOP
                                Z)))
                    Z
                    Y)))
        X)
```

3) Schema ID# 182108649.5.2 peak_fitness= 0.865000 MaxFoO= 0.124000

```
(-
        (+
            X
            -3.268226)
        Z)
```

4) Schema ID# 111973372.19.4 peak_fitness= 0.865000 MaxFoO= 0.088000

```
(+
        (-
            (NOP
                X)
            (%
                (IFLTE
                    -0.579501
                    Z
                    Z
                    Z)
                (+
                    Y
                    Z)))
        (-
            (+
```

```
                X
                -3.268226)
        (NOP
                Z)))

5) Schema ID# 18272820.5.2 peak_fitness= 0.865000 MaxFoO= 0.080000
(%
        Z
        (+
                Y
                Z))

6) Schema ID# 260356306.8.3 peak_fitness= 0.865000 MaxFoO= 0.072000
(-
        (NOP
                X)
        (%
                Z
                (+
                        Y
                        Z)))

7) Schema ID# 55866704.27.6 peak_fitness= 0.865000 MaxFoO= 0.072000
(-
        (+
                X
                -3.268226)
        (IFLTE
                -0.579501
                (+
                        (-
                                (NOP
                                        X)
                                (%
                                        (IFLTE
                                                -0.579501
                                                Z
                                                Z
                                                Z)
                                        (+
                                                Y
                                                Z)))
                        (-
                                (+
                                        X
                                        -3.268226)
                                (NOP
                                        Z)))
                Z
                Y))

8) Schema ID# 4979376.11.3 peak_fitness= 0.865000 MaxFoO= 0.068000
(-
        X
        (%
                (IFLTE
                        -0.579501
                        Z
                        Z
                        Y)
                (+
                        Y
                        Z)))

9) Schema ID# 72743047.15.4 peak_fitness= 0.865000 MaxFoO= 0.060000
(+
        (-
                (NOP
                        X)
```

```
                (%
                        Z
                        (+
                                Y
                                Z)))
        (-
                (+
                        X
                        -3.268226)
                (NOP
                        Z)))
```

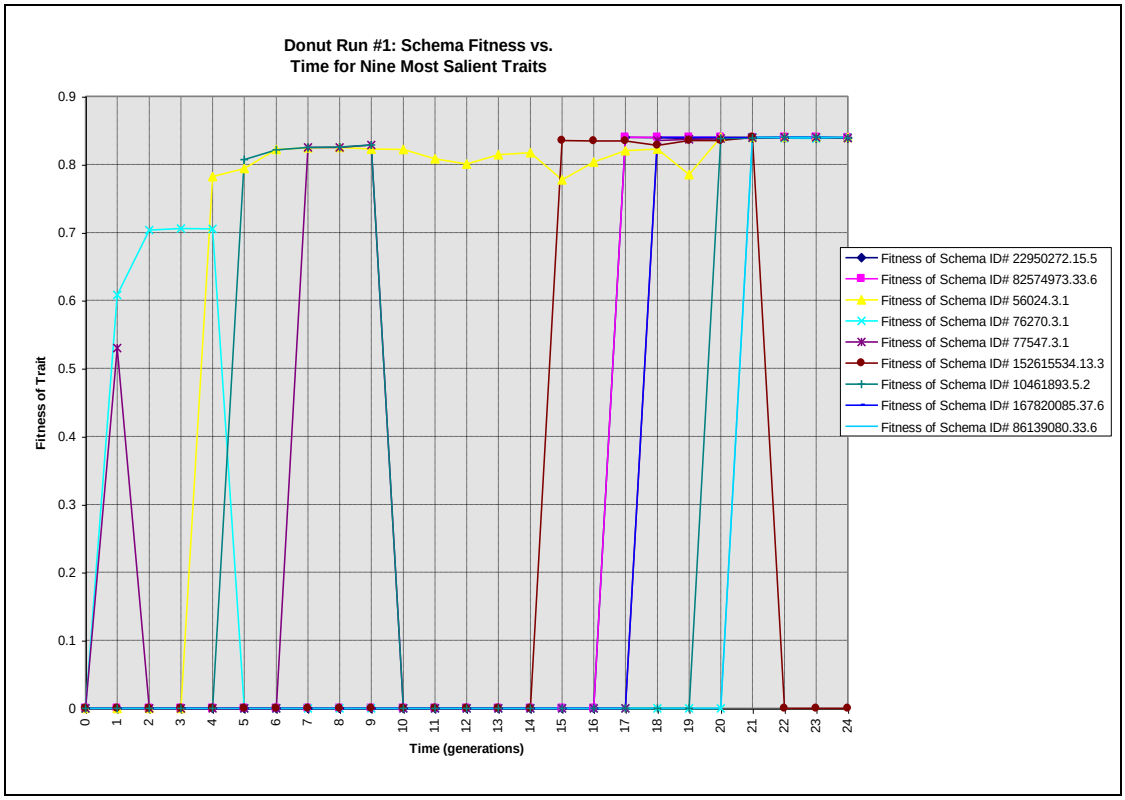


Figure 7.8: Fitness vs. Time for run #1 of the Donut induction problem.

7.6.2. Donut Run #1

Generation 0 produced 1660 unique subexpressions from a population size of 250. Subsequent generations produced an average of about 250 new (unique) subexpressions per generation. Out of these, a total of 845 passed the criteria for consideration as salient expressions. The top nine are reported on here. Fitness values achieved in this run are significantly lower than in run #0, considered in the context of the "close race" characteristic of fitness scores for this problem. They reach a maximum of 0.84. The relationship between the top ranked expressions

is not so hierarchical as for run #0. The subexpression (1) is contained within (2), and (5) within (7), while all the others are similar without being exact components of each other.

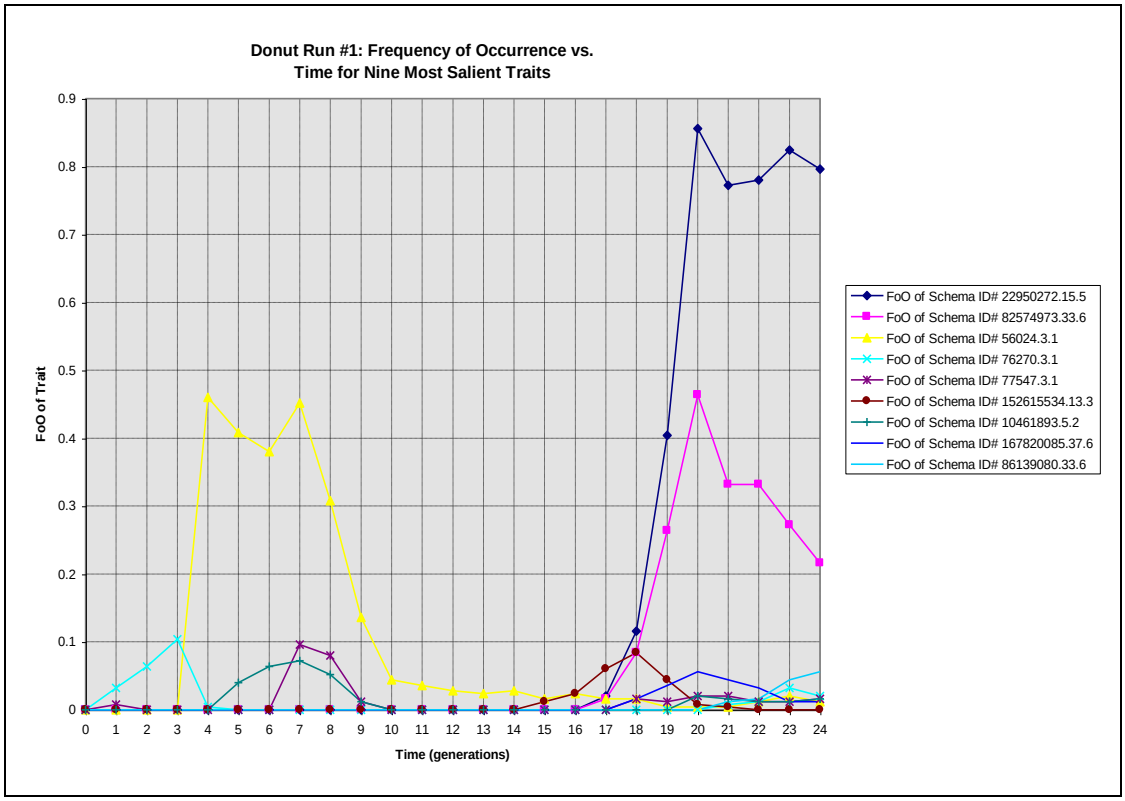


Figure 7.9: Frequency of occurrence vs. Time for run #1 of the Donut induction problem.

The expressions ranked (1) and (2) behave in much the same way as the two top-ranked expressions in run #0. They appear on the scene with a high fitness which is retained throughout the remainder of the run, and the shape of that fitness curve is highly correlated with that of the frequency-of-occurrence curves. Subexpression (1) appears with greater frequency than (2), indicating that it can appears in other contexts while retaining high fitness.

Again, it appears likely that subexpression (1) is an important salient element, although it is not clear by inspection what the important relations are. It can be observed, however, that this subexpression is a second degree polynomial in Y. This is of some interest since the term Y^2 figures prominently in the optimal Bayesian decision surface illustrated in Figure 4.4 and Equations 4.1-4.3.

Table 7.5: Nine most fit subexpressions for run #1 of the Donut induction problem.

```
1) Schema ID# 22950272.15.5 peak_fitness= 0.840000 MaxFoO= 0.856000
(-
        X
        (*
                (+
                        (+
                                Y
                                (+
                                        Y
                                        Y))
                        (+
                                Y
                                Y))
                (+
                        Y
                        Y)))

2) Schema ID# 82574973.33.6 peak_fitness= 0.840000 MaxFoO= 0.464000
(IFLTE
        Y
        (IFLTE
                -1.894904
                Y
                (*
                        (+
                                Y
                                Y)
                        Y)
                X)
        (-
                X
                (*
                        (+
                                (+
                                        Y
                                        (+
                                                Y
                                                Y))
                                (+
                                        Y
                                        Y))
                        (+
                                Y
                                Y)))
        (%
                Y
                (%
                        (*
                                Y
                                X)
                        Y)))

3) Schema ID# 56024.3.1 peak_fitness= 0.840000 MaxFoO= 0.460000
(*
        Y
        Y)

4) Schema ID# 76270.3.1 peak_fitness= 0.840000 MaxFoO= 0.104000
(%
        Y
        X)

5) Schema ID# 77547.3.1 peak_fitness= 0.840000 MaxFoO= 0.096000
(%
        Y
        Y)

6) Schema ID# 152615534.13.3 peak_fitness= 0.840000 MaxFoO= 0.084000
```

```
(IFLTE
        -1.894904
        (+
                Y
                (+
                        Y
                        Y))
        (*
                (+
                        Y
                        Y)
                Y)
        X)

7) Schema ID# 10461893.5.2 peak_fitness= 0.840000 MaxFoO= 0.072000
(%
        4.090448
        (%
                Y
                Y))

8) Schema ID# 167820085.37.6 peak_fitness= 0.840000 MaxFoO= 0.056000
(IFLTE
        Y
        (IFLTE
                -1.894904
                Y
                (*
                        (+
                                Y
                                Y)
                        Y)
                X)
        (-
                X
                (*
                        (+
                                (+
                                        Y
                                        (+
                                                Y
                                                Y))
                                (+
                                        Y
                                        Y))
                        (+
                                Y
                                Y)))
        (%
                (*
                        (+
                                Y
                                Y)
                        Y)
                (%
                        (*
                                Y
                                X)
                        Y)))

9) Schema ID# 86139080.33.6 peak_fitness= 0.840000 MaxFoO= 0.056000
(IFLTE
        Y
        (IFLTE
                -1.894904
                Y
                (*
                        (+
                                Y
```

```
                                        Y)
                                    Y)
                                Y)
(-
                                X
                                (*
                                    (+
                                        (+
                                            Y
                                            (+
                                                Y
                                                Y))
                                        (+
                                            Y
                                            Y))
                                    (+
                                        Y
                                        Y)))
(%
                                Y
                                (%
                                    (*
                                        Y
                                        X)
                                    Y)))
```

7.7. Discussion

The experiments presented here provide a wide range of difficulty in interpretation. In run #1 of the Royal Road problem there is a solution which should clearly stand out when ordered by fitness, and indeed it does. Run #0, on the other hand, presents a situation where the solution is buried among a clutter of "hitchhiking" subexpressions. There, ordering alone does not provide enough information since an obvious hitchhiker receives equal estimated fitness with a higher frequency of occurrence than the most truly fit expression. There are distinguishing features in the temporal properties of fitness and frequency, however, by which the true solution may be separated out.

Moving on to the Donut problem, we have observed up front that looking for salient features is inherently a "blind" experiment, since the optimal Bayesian discriminant cannot be constructed with the function set provided. Any inferences about saliency must be made using the cues which were discovered when working with constructional problems. It was observed that for both types of problems, subexpressions are found that have a high initial fitness which does not decay, accompanied by a rapid rise in frequency of occurrence. Furthermore, we know that subexpressions showing those characteristics in the Royal Road problems were indeed salient. We therefore form a hypothesis that the traits which display the same temporal fitness and

frequency characteristics in the Donut problem are salient as well. Presuming that this hypothesis is correct, the conclusions we reach for the Donut induction reveal a sort of good news/bad news situation. The good news is that there are traits which provide some evidence of saliency, and these traits have a top ranking among all which were considered. The bad news is that although clearly important, those traits are complex and difficult to comprehend. The analysis therefore shows *which* components of the solution are important, but does not explain *why*. The research performed here is only a simple demonstration of principles. A variety of suggestions for improving the sophistication of trait mining will be discussed in Chapter 8.

Chapter 8

Conclusions and Directions for Future Research

8.1 Conclusions

The work described here has answered many questions about Genetic Programming. At the outset, one important question was "what is Genetic Programming, and how does it differ from what has gone before?" The conclusion reached in Chapter 2 is that Genetic Programming is closely akin to heuristic search, specifically beam search. Population and selection effectively implement a fixed queue of states, and at any given time those states which have the best heuristic value will be "opened" in order to generate successors. One difference from conventional beam search is that the state of the queue is updated as a batch process through stochastic sampling, but this is unlikely to have any great effect on the behavior of the algorithm. The most novel aspect of the genetic method is probably the concept of recombination, in which parts of successful strategies are combined to form successors. Also unusual among search methods is that all successors of a given state cannot be enumerated easily. This, however, is largely due to the domain being searched rather than the search method. Most operators which can produce significant variations of program structures over time have the power to produce a large number of perturbed versions of a single program. Understanding the GP method in the language of AI search fosters communication between the respective communities. Equally importantly, it opens up a "short cut" to theoretical understandings of GP by way of analogy.

A second question was central to Chapter 3, namely: "Can Genetic Programming be applied to a real-world problem, and if so, how does it compare to other methods?" The answer was surprising, in the sense that the GP method was favorable in not just one but a variety of ways. First of all, genetically induced solutions achieved better error rates than the neural or decision-tree methods. Second, the formulae induced by GP required an order of magnitude less computational cost than the neural methods. Finally, and perhaps most important to many

researchers, the GP method required no "finessing" of algorithm parameters, whereas the other methods required a great deal of trial and error to determine system configuration. Also, some human-understandable solution elements created by GP in those experiments inspired the trait-mining concept which forms the basis for Chapter 7. One disadvantage of the GP method was computational expense of the learning process itself, requiring approximately an order of magnitude greater time than the neural method. It is interesting to note the inverse relationship between learning time and post-learning computational cost of solutions. With regard to contributions, the work described in Chapter 3 and in (Tackett 1993) was the first large-scale test of Genetic Programming against real data.

The work of Chapter 3 left open the question as to what properties of Genetic Programming bestow its apparent performance advantages for induction problems. Toward this end a new induction problem was introduced in Chapter 4 in order to parametrically measure GP performance as a function of noise and training sample density. The original intent of the experiment was to demonstrate the conditions under which the method "breaks down." In this regard, the experiments were disappointing since the method did not break down, but rather degraded gracefully. For a given training set size, the classification performance as a function of noise variance was within approximately a constant amount of the Bayesian limit. Reasonable solutions were obtained using as few as 40 training samples. These results represent the first parametric study of GP for induction problems, and suggest that Genetic Programming derives some performance advantages through an ability to generalize. Although the original work intended to only examine Genetic Programming, an interesting contrast is provided in (Carmi 1994), which repeats the experiments designed here using a Multilayer Perceptron trained by Backpropagation. The results there show that for all noise levels the neural method achieves worse performance on average than the GP method, and requires in excess of 300 training samples to converge to any reasonable solutions.

A second track pursued in Chapter 4 was to examine the differences in performance due to selection methods, namely spatially distributed selection and steady-state selection. Some differences between methods were statistically significant, but it is questionable if they were of practical significance. This stands in contrast to results obtained by researchers in the field of conventional genetic algorithms (GA), who claim that both methods confer significant

performance advantages. One likely explanation is that this is the first such study which deals with generalization: it measures performance only against data which were not used in training the system. Prior studies of selection have focused exclusively upon optimization problems, in which training and test data were the same. Overall, the various selection methods produced only minor differences in performance, leading us to consider instead improvements to recombination.

Studies of selection in Chapter 4 sought to determine if Genetic Programming performance could be improved. This investigation was continued in Chapter 5, but through a different aspect of the GP algorithm. Recombination was identified there as an area where a great deal of improvement is possible. In standard GP recombination, offspring are chosen uniformly at random from among the large number possible from a single pairing. The Greedy Recombination operator samples from among these potential offspring and chooses the best from that sample, thereby substituting a greedy choice for a random choice. Arguments were presented based both upon probability and upon search, predicting that Greedy Recombination should produce fitter offspring, on average. An important result obtained empirically was that a different fitness evaluation may be used for selecting among potential offspring. In particular, that evaluation can have greatly reduced cost relative to the fitness evaluation of "mature" population members. The result was that Greedy Recombination consistently produced improved performance at reduced cost relative to the standard method of recombination. This is the first published presentation of the algorithm and results for Greedy Recombination.

Although the results of Chapter 5 are encouraging they are not general, being applied only to a single problem. To rectify this, Chapter 6 introduced a way of constructing problems which allow insights into the search properties of the genetic method. The Royal Road and Deceptive problems are performed separately, but they are interpreted together: the only difference between them is the "tuning" of the search landscape through the change in a single parameter. It is the difference in results between the two methods which is most pertinent to performance insights. This methodology of analysis is new to Genetic Programming. A total of six methods of selection/recombination were examined, including random and steady-state selection. Their properties were clearly differentiated by the problem suite. One particularly encouraging insight was that many predicted properties of the steady-state method, well-scrutinized in the GA

world, were directly observable. Those properties include "allele loss" (DeJong 1993) and convergence to local minima. They were not as clearly illustrated by the experiments of Chapter 4 due to less precise control over problem structure. Likewise, the exploratory properties of Greedy Recombination predicted in Chapter 5 were observable. Therefore, Chapter 6 not only provided a new analytic methodology to Genetic Programming, but also demonstrated in a general and apparently reliable way that Greedy Recombination provides improved performance on problems where the fitness landscape is prone to misleading local optima.

Chapter 7 explored a completely novel aspect of Genetic Programming, namely the extraction of symbolic information from the population. In order to show why this is not a trivial task, we demonstrated the principles of bloating ("the size problem")and hitchhiking, proposing that they are related phenomena. Whereas the pathology of hitchhiking is well-understood from the literature of GA, the size problem (increase in average size of expressions with time) is unique to Genetic Programming and has not been understood at all until now. Prior theories of the size problem have often supposed that it is a selection-neutral property of the recombination operator. Results in Chapter 7 demonstrated that the size problem is directly dependent upon selection pressure for a relatively generic problem. More conclusively, those same results demonstrate that under random selection there is no size problem, even though recombination is taking place. This is a very general result, since random selection must have the same effects for all fitness functions, terminal sets, and function sets (or at least those function sets which have similar branching factors). Once the presence of these phenomena were established, the question of how to "mine" salient traits from large amounts of inert code was examined. It was shown that static measures of fitness and frequency of occurrence could be confounded by hitchhiking. A result that is new and significant among the literature of both GA and GP was achieved through the demonstration that the time signatures of fitness and frequency of occurrence distinguish salient features from hitchhikers. This principle was demonstrated first using constructional problems, and then was shown to carry over to semantically evaluated functions. Further implications resulting from Chapter 7 will be discussed in the next section.

8.2 Directions for Future Research

Earlier chapters have demonstrated the tremendous promise of Genetic Programming in the field of induction through demonstrations against some very hard problems. They have provided new analyses and theory as well as extensions to the method. Meanwhile, other researchers are not idle. John Koza of Stanford continues research into the automatic generation of subprograms and into the use of specialized iteration constructs for syntactic pattern recognition in structures such as protein sequences (Koza 1994). Lee Altenberg of Duke is providing new and important theoretical results concerning the open-ended development of the genome structure in genetically induced programs (Altenberg 1994). Kim Kinnear of Adaptive Computing is exploring the ruggedness of the fitness landscape, in an attempt to discriminate between how much difficulty is introduced by a problem in comparison to the difficulty inherent in searching program space (Kinnear 1994). Astro Teller of CMU has provided methods of Genetic Programming which incorporate a read-write store, proved that they are Turing complete, and created practical problems which demonstrate many interesting properties of self-organized memory (Teller 1994a; 1994b). There are of course many others.

The work here described through Chapter 6 and the work by others mentioned above are focused on improving the *power* of Genetic Programming to solve problems. It is important to keep sight of the fact that Genetic Programming is a powerful method because of *representation*: it learns procedural information rather than parameters. That alone is enough to justify its use. But this same representation offers the potential for the learning process to be a "two way street," allowing users to learn what the machine has learned. Many representations cannot offer that property. The work shown in Chapter 7 represents the first fundamental step in knowledge extraction from genetically induced programs. The following suggestions therefore seek to extend new directions for genetic program induction and trait-mining:

- Gene banking and trait mining were tested in the context of the Royal Road and Donut induction primarily because those problems and their properties were thoroughly examined throughout previous chapters. Those problems are, however, abstract with respect to the "meaning" of their elements and relationships. Recall, for example, how Chapter 3 provided the first indication that knowledge might be extracted when it was shown that the concept

of "contrast" was synthesized from primitives representing intensity. Although the vehicle discrimination problem is probably too abstract to reliably provide such meaningful discoveries, it suggests a design for problems specifically intended to test trait mining. Therefore, one suggestion is to revisit the analysis of Chapter 7 using problems specifically designed to have meaningful elements, with some expectation of what relationships should be found.

- In the discussion at the end of Chapter 7 it was pointed out that our simplistic trait mining scheme indicates which problem elements are important but does not indicate why. One approach to explaining why traits are important is to apply domain-dependent analysis. In the case of the Donut problem, one might apply graphics and analytic geometry to better understand the characteristics of the discriminant surface. Domain dependent adjuncts to the method can be costly since they require a good deal of work for each new specific problem, but those costs are acceptable for long term projects. This is especially true in domains where the problem elements are fixed but the system must be constantly retrained in order to adapt to shifting characteristics of incoming data. In such a problem where new learning is continuously being performed, new relationships must be continuously interpreted to explain changing conditions. Prediction and induction on financial markets provide a good example of an application which bears those properties. We suggest that research in domain dependent trait mining be explored in the context of such a large scale real world problem.
- The genetic method proceeds with a broad brush, condensing information about performance against all training data into a single heuristic measure of fitness which affects the search process primarily through selection. One interesting alternative to this approach is offered in (Quinlan 1990), which suggests "pruning" expressions by examining how the deletion of a subexpression affects performance against training data. The scheme is easily implemented in that work since the expression, or "clause," being pruned consists of a linear list of literals: the pruning algorithm simply tries deleting each literal one at a time. Deletion of subexpressions in a tree structure is somewhat more tricky. One approach might be to successively replace a function node with each of its branches. Beginning at the root of an expression and working downward would serve to remove the largest pieces first. This

would massively increase the cost of search if applied to the whole population, however, since each member would need to be tested against the training set anew for each proposed deletion, which would multiply the number of evaluations by a factor proportional to the average expression size. Pruning of a few select individuals would not be too expensive however, and so pruning might best be applied as a trait-mining tool. Also, some lessons from Greedy Recombination may be applied by using a reduced-size set of training data to drive the pruning process.

- Related to the analysis described above, it can also be useful to examine which components of an expression affect which aspects of fitness. There is an analogy to this in lesion studies conducted by neuroscience researchers, where functionality of cortical areas can be inferred from the results of damage to those areas. The idea in the context of induction is to observe which elements of the training set are categorized incorrectly in response to pruning a particular program branch. Similar studies of "mutation analysis" are carried out by geneticists as well.
- One other approach that can have a high likelihood of identifying important problem elements is to perform cross-comparison of important subexpressions developed by separate runs. Aside from difficulties inherent in performing the large number of runs to support this, comparisons can be difficult to perform at the syntactic level since different subexpressions can produce identical behaviors. Some of these can be arrived at by simplification, but simplification rules are virtually impossible to generalize. To do so would be the equivalent of producing a compiler which maps all semantically equivalent programs into a single object code. The obvious alternative is to classify subexpressions according to their response to a set of test cases. In that scheme the "phenotype" of algorithm behavior is characterized by a vector which has one element for each case in the training data set. That element is just the response of the expression or subexpression to a single test case. Similarity can then be characterized by computing cross-correlation and covariance between response of subexpressions.
- The analyses made here are not context-sensitive: measures have been made of the conditional fitness of expressions according to the subexpressions they contain, but those measures do not speak of where subexpressions appear. Any analysis of context is

confounded by the fact that programs are generally different from each other in size and "shape." Therefore a context - a location within some program - only has meaning with respect to that program. One beginning of an approach might be to analyze the dual of subexpressions, which we will dub here as "superexpressions." A superexpression is just a program which has a "wildcard" in place of one of the subexpressions it contains. Just like there are a number of subexpressions in a program equal to the number of nodes it contains, likewise there is an equal number of superexpressions.

- Some performance improvement was achieved in Chapter 5 by altering the recombination operator, generating successors of a pair of parents by a method more intelligent than blind random choice. One application of the gene banking technique which has not been addressed in this research is the construction of successor operators which are gene-bank based. The availability of descriptive statistics for each subexpression suggests that recombination can be replaced with "genetic engineering," in which both the insertion site and subexpression to be inserted in an individual are chosen based on genebank information. Code already exists to do this quickly and efficiently, since each node in the population can be tagged with the hash code for the subexpression of which it is the root. The criteria by which subexpressions are picked can shift as well, adjusting to conditions and compensating for loss of diversity ("allele loss") and decrease in population fitness variance.
- Reinforcement and credit assignment are intimately tied to the issues of trait mining. Whereas the effect of selection upon the propagation of traits has been clearly illustrated as a "weak credit assignment method" (Angeline 1993) that acts during the on-line learning process, the trait mining scheme represents an a posteriori credit assignment scheme. The concepts of reinforcement learning (Kaebling 1990) deal explicitly with identification of which aspects of a learned strategy are affecting the performance of that strategy. There is also a strong relationship between reinforcement learning and classifier systems (Holland 1986, Goldberg 1989), which are a variant of machine learning that incorporates the Genetic Algorithm. Much as the analysis in earlier chapters benefited from an understanding of Genetic Programming as AI search, so could trait mining benefit from being studied in the context of known methods of credit assignment and reinforcement.

- *The size problem in Genetic Programming*. A common observation among practitioners of Genetic Programming is that the average size of programs in the population becomes greater with time. Researchers have postulated a variety of theories to account for this, many of which assume that this growth is an artifact of recombination independent of fitness and selection. A result that runs counter to these theories was obtained when working with the "GP Royal Road" functions described in Chapter 7 of my dissertation. Those experiments used six different combinations of selection and recombination methods, covering a wide range of selection pressure. Those experiments clearly demonstrate that the rate of increase in the average size of expressions is directly proportional to selection pressure. This analysis indicates that the size problem is strongly driven by fitness and selection, with some indications that "hitchhiking," well known in GA literature, may be the culprit. A preliminary hypothesis concerning the pathology of the size problem was proposed in Chapter 7, and further experiments can be designed in order to test that hypothesis and provide an analytic characterization of bloating. This is useful to the extraction of salient traits because it may point a way to removing hitchhikers and generally pruning GP solutions.
-

Bibliography

Ackley, David H. (1994) A case for distributed Lamarckian evolution. in *Artificial Life III*. In *Artificial Life III*. C.G. Langton (ed). Santa Fe Institute Studies in the Sciences of Complexity, Proc. Vol. XVII. Redwood City, CA: Addison Wesley.

Adams, Douglas (1982) *The Hitchhikers Guide to the Galaxy*. Pocket Books.

Altenberg, Lee, 1993. Personal communication.

Altenberg, Lee, 1994. The evolution of evolvability in genetic programming. In *Advances in Genetic Programming*, Chapter 3. Kenneth E. Kinnear, Jr., Ed., Cambridge, MA: MIT Press.

Altenberg, Lee, 1994b. Emergent phenomena in genetic programming. In Proceedings of the <???> Conference on Evolutionary Programming. San Diego, CA, February.

Angeline, P. J. (1993) *Evolutionary Algorithms and Emergent Intelligence*. Doctoral Dissertation, The Ohio State University, Laboratory for Artificial Intelligence Research (LAIR), December.

Angeline, P. and Pollack, J. (1994) Co-evolving high-level representations. In *Artificial Life III*. C.G. Langton (ed). Santa Fe Institute Studies in the Sciences of Complexity, Proc. Vol. XVII. Redwood City, CA: Addison Wesley.

Back, T., and Schwefel, H. (1993) An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation* v1n1, pp 1-24.

Bisiani, R. (1987). Beam search. In: Shapiro, S. (Ed.) Encyclopedia of Artificial Intelligence, Vol. 1. New York, NY: Wiley.

Brindle, A. (1981) *Genetic Algorithms for Function Optimization*. Unpublished Ph.D. dissertation, U. of Alberta, Edmonton.

Carmi, Aviram, (1994). Research to appear as Master's Thesis. Dept. of Computer Science, California State University at Northridge.

Collins, R.J., and Jefferson, D. (1991) Selection in massively parallel genetic algorithms. In Belew, R, and Booker, L. (Eds.) *Proceedings of the Fourth International Conference on Genetic Algorithms*, pp 249-256. San Mateo: Morgan-Kaufmann.

Collins, R.J. (1992) *Studies in Artificial Evolution*. Ph.D. Dissertation, University of California at Los Angeles. Available by public ftp to cognet.ucla.edu.

Daniell, C.E., Kemsley, D.H., Lincoln, W.P., Tackett, W.A., Baraghimian, G.A. (1992) Artificial neural networks for automatic target recognition. *Optical Engineering*, v31 n12, Dec. 1992, pp. 2521-2531.

Darwin, Charles (1876) *The Effect of Cross and Self Fertilisation in the Vegetable Kingdom*. Murray, London, UK.

Dawkins, Richard. (1987) The Blind Watchmaker. New York, NY: Norton.

De Jong, K. A. (1975) An analysis of the behavior of a class of genetic adaptive systems. (Doctoral dissertation, University of Michigan). *Dissertation Abstracts International 36(10)*, 5140B. University Microfilms No. 76-9381.

De Jong, Kenneth A. (1993) "Generation Gaps Revisited." in Whitley, L. D. (ed) (1993) *Foundations of Genetic Algorithms 2*. Morgan Kaufmann.

Fisher, R.A. (1930; 1958) *The Genetical Theory of Natural Selection*,*(2ed)*. Dover Press, New York.

Forrest, Stephanie, and Mitchell, Melanie (1993) Relative building-block fitness and the building-block hypothesis. In: Whitley, L. D. (ed) (1993) *Foundations of Genetic Algorithms 2*. Morgan Kaufmann.

Fukushima, K. (1982) Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position. *Pattern Recognition*, Vol. 15, No. 6, pp. 445-469.

Fukushima, K. (1989) Analysis of the process of visual pattern recognition by the neocognitron," *Neural Networks*, Vol. 2, pp. 413-420.

Gilbert, W. (1987) The exon theory of genes. *1987 Cold Harbor Symposia on Quantitative Biology, Volume LII*.

Gilmour, J., and Gregor, J. (1939) Demes: A suggested new terminology. *Nature*, 144:333.

Goldberg, David E. (1983) Computer-Aided Gas Pipeline Operation Using Genetic Algorithms and Rule-Learning. Ph.D. Dissertation, University of Michigan. *Dissertation Abstracts International*, 44(10), 3174B. (University Microfilms No. 8402282).

Goldberg, David E. (1989)*Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading MA: Addison Wesley.

Goldberg, David E. (1989b) Zen and the art of genetic algorithms. In: Schaffer, J.D. (Ed.) *Proc. of the Third International Conference on Genetic Algorithm*s. San Mateo, CA: Morgan Kaufmann.

Goldberg, D.E., and Deb, K. (1991) A comparative analysis of selection schemes used in genetic algorithms. In Rawlins, J.E. (ed) *Foundations of Genetic Algorithms*. Morgan Kaufmann.

Grossberg, S. (1988) *Neural Networks and Natural Intelligence*. Cambridge: MIT Press.

Hertz, J., Krogh, A., Palmer, RG (1991) *Introduction to the Theory of Neural Computation*. Redwood City, CA: Addison Wesley.

Hillis, W. Daniel (1991) Co-evolving parasites improve simulated evolution as an optimization procedure. In: *Artificial Life II*. Redwood City, CA: Addison Wesley.

Holland, John H. (1975; 1992) *Adaptation in Natural and Artificial Systems*. Cambridge, MA: MIT Press.

Holland, John H. (1986) Escaping brittleness: the possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In: Michalski, R., Carbonell, J., and Mitchell, T. (Eds.) *Machine Learning, an Artificial Intelligence Approach. Volume II.* Los Altos, CA: Morgan Kaufmann..

Hu, T.C. (1982) *Combinatorial Algorithms*. Menlo Park, CA: Addison-Wesley.

Hughes Aircraft Co. (author unnamed), (1990) Bandwidth Reduction Intelligent Target-Tracking / Multi-function Target Acquisition Processor (BRITT / MTAP): Final Technical Report, CDRL B0001. El Segundo, CA.

Ingber, L., and Rosen, B. (1992) Genetic algorithms and very fast simulated reannealing: A comparison. *Mathematical Computer Modelling*, 16(11):87-100, 1992.

Kanal, LN (1979), "Problem-solving models and search strategies for pattern recognition," IEEE Trans. Pattern Anal. Machine Intell., vol. PAMI-1, pp. 194-201, Apr.

Kinnear, K. (1993a) Generality and difficulty in genetic programming: evolving a sort. In Forrest, S. (ed) *Proceedings of the Fifth International Conference on Genetic Algorithms,* Urbana-Champaign IL. Morgan Kaufmann.

Kinnear, Kenneth E. (1993b) "Evolving a Sort: Lessons in Genetic Programming." In *Proceedings of the 1993 International Conference on Neural Networks*. New York, NY: IEEE Press.

Kinnear, Kenneth E., 1993. Personal Communication.

Kenneth E. Kinnear, Jr. (Ed) (1994) *Advances in Genetic Programming*. Cambridge, MA: MIT Press.

Kohonen, T. (1987) *Self Organization and Associative Memory, 2ed*. Springer-Verlag.

Korf, R.E. (1987). Search. In: Shapiro, S. (Ed.) Encyclopedia of Artificial Intelligence, Vol. 2. New York, NY: Wiley.

Koza, John R. (1989) Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence, v1*. Morgan Kaufmann.

Koza, John R. (1992)*Genetic Programming*. Cambridge, MA: MIT Press.

Koza, John R. (1994)*Genetic Programming II*. Cambridge, MA: MIT Press.

Kozlowski, J., and Stearns, S.C., 1989. "Hypothesis for the Production of Excess Zygotes: Models of Bet-Hedging and Selective Abortion." *Evolution*. 43(7):1369-1377.

Larsen & Marx (1981) *An Introduction to Mathematical Statistics*. Prentice-Hall.

Lowerre, B.T., and Reddy, R.D. (1980) The Harpy speech understanding system. In: Lea, W.A. (Ed.) *Trends in Speech Recognition*. Englewood Cliffs, NJ: Prentice-Hall.

Maynard-Smith, J. (1989) *Evolutionary Genetics*. New York: Oxford University Press.

Miller, John H. (1989) *The Co-Evolution of Automata in the Repeated Prisoners Dilemma*. Santa Fe Institute Report 89-003, Santa Fe, NM.

Minsky, M., and Papert, S. (1969) *Perceptrons*. Cambridge MA: MIT Press.

Mitchell, M. Forrest, S., and Holland, J. (1992) The royal road for genetic algorithms: Fitness landscapes and GA performance. In Varela, F. and Bourgine P. (Eds.) *Proceedings of the 1st European Conference on Artificial Life*. Cambridge: MIT Press.

Mitchell, M. and Holland, J. (1993) *When Will a Genetic Algorithm Outperform Hill-Climbing?* Santa Fe Institute Report 93-06-037, Santa Fe, NM.

Pagan, F. (1981) *Formal Specification of Programming Languages*. Prentice-Hall.

Papoulis, A. (1965; 1984) Probability, Random Variables, and Stochastic Processes. New York, NY: McGraw-Hill.

Pazzani, M. and Kibler, D. (1990) The utility of knowledge in inductive learning. Technical Report 90-18, University of California at Irvine.

Poggio, T. and Girosi, F. (1989) *A Theory of Networks for Approximation and Learning*. MIT AI Laboratory, A.I. Memo #1140, July.

Provine, W. (1986) *Sewall Wright and Evolutionary Biology*. University of Chicago Press.

Quinlan, J. Ross. (1983) Learning efficient classification procedures and their application to chess end games. Chapter 15 in: Michalski, R., Carbonell, J., and Mitchell, T., (Eds.) *Machine Learning*. Palo Alto, CA: Tioga.

Quinlan, J. Ross. (1990) Learning logical definitions from relations. *Machine Learning*, 5, 239-266.

Rawlins, J.E. (ed) (1991) *Foundations of Genetic Algorithms*. Morgan Kaufmann.

Ray, T. S. (1991) An approach to the synthesis of life. In *Artificial Life II*, edited by C.G. Langton et al. Santa Fe Institute Studies in the Sciences of Complexity, Proc. Vol. X. Redwood City, CA: Addison Wesley.

Reynolds, Craig (1992) Personal Communication.

Reynolds, Craig W. (1994a) "Evolution of Obstacle Avoidance Behavior: Using Noise to Promote Robust Solutions." In *Advances in Genetic Programming*. Kinnear, K. E. (ed), Cambridge, MA: MIT Press.

Reynolds, Craig W. (1994b) The difficulty of roving eyes. To appear in: Z. Michaelewicz (ed) *Proceedings of the 1994 IEEE Conference on Evolutionary Computation, Orlando FL*. New York: IEEE Press.

Rosenblatt, F. (1962) *Principles of Neurodynamics*. Washington DC: Spartan Books.

Rosenbloom, P. (1987). Best first search. In: Shapiro, S. (Ed.) Encyclopedia of Artificial Intelligence, Vol. 2. New York, NY: Wiley.

Rummelhart, D.E., and McClelland, J. (1986) *Parallel Distributed Processing vol. 1*. Cambridge: MIT Press.

Shannon, Claude E. (1948) A mathematical theory of communications. Bell System Technical Journal. Part I: pp. 379-423, July. Part II: pp. 623-656, October.

Singleton, Andrew (1993) Personal Communication.

Stearns, Stephen C. (1987) "The Selection Arena Hypothesis." In Stearns, S.C. (ed), The Evolution of Sex and Its Consequences. Burkhauser, Basel, Switzerland.

Sutherland, S. (1986) Patterns of fruit-set: What controls fruit-flower ratios in plants? *Evolution* 40:117-128.

Syswerda, G. (1989) Uniform crossover in genetic algorithms. In Schaffer, J. (ed) *Proceedings of the Third International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann.

Tackett, Walter A. (1993) "Genetic programming for feature discovery and image discrimination. In Forrest, S. (ed) *Proceedings of the Fifth International Conference on Genetic Algorithms,* Urbana-Champaign IL. Morgan Kaufmann.

Tackett, Walter A. (1994) Greedy recombination and the genetic construction of computer programs. Submitted to: *Foundations of Genetic Algorithms 3*. Estes Park, CO.

Tackett, Walter A., and Gaudiot, J-L. (1992) Adaptation of self-replicating digital organisms. Proc. of the International Joint Conference on Neural Networks 1992, Beijing, China. New York, NY: IEEE Press.

Tackett, Walter A., and Carmi, A. (1994a) The donut problem: Scalability, generalization, and breeding policy in the genetic programming, Chapter 7 of: Kinnear, K. (ed) *Evolution of Genetic Programming*. MIT Press.

Tackett, Walter A., and Carmi, A. (1994b) The unique implications of brood selection for genetic programming. To appear in: Z. Michaelewicz (ed) *Proceedings of the 1994 IEEE Conference on Evolutionary Computation, Orlando FL*. New York, NY: IEEE Press.

Teller, Astro (1994a) The evolution of mental models. In: Kinnear, K. (ed) *Evolution of Genetic Programming*. MIT Press.

Teller, Astro (1994b) Turing completeness in the language of genetic programming with indexed memory. To appear in: Z. Michaelewicz (ed) *Proceedings of the 1994 IEEE Conference on Evolutionary Computation, Orlando FL*. New York, NY: IEEE Press.

Wetzel, A. (1979). Reported as Personal Communication in (Brindle 1981).

Whitley, D. (1989) The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In Schaffer, J. (ed) *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann.

Whitley, D. (ed) (1993) *Foundations of Genetic Algorithms 2*. Morgan Kaufmann.

Wright, Sewall, (1931) Evolution in Mendelian populations. *Genetics* 16:97-159.

Wright, Sewall, (1932) The roles of mutation, inbreeding, crossbreeding, and selection in evolution. In *Proceedings of the Sixth International Congress of Genetics, v1*, pp356-366.

Wright, Sewall, (1978) *Evolution and the Genetics of Populations. Volume 4: Variability Within and Among Natural Populations*. University of Chicago Press.

(this page unintentionally left blank)