BETHKE, ALBERT DONALLY

GENETIC ALGORITHMS AS FUNCTION OPTIMIZERS

The University of Michigan PH.D. 1980

GENETIC ALGORITHMS AS FUNCTION OPTIMIZERS

by
Albert Donally Bethke

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer and Communication Sciences)
in The University of Michigan
1980

Doctoral Committee:

Professor John H. Holland, Chairman
Assistant Professor Peter J. Benson
Associate Professor Thomas F. Storer
Professor Richard A. Volz

RULES REGARDING THE USE OF

MICROFILMED DISSERTATIONS

Microfilmed or bound copies of doctoral dissertations submitted to The University of Michigan and made available through University Microfilms International or The University of Michigan are open for inspection, but they are to be used only with due regard for the rights of the author. Extensive copying of the dissertation or publication of material in excess of standard copyright limits, whether or not the dissertation has been copyrighted, must have been approved by the author as well as by the Dean of the Graduate School. Proper credit must be given to the author if any material from the dissertation is used in subsequent written or published work.

ACKNOWLEDGEMENTS

I thank John Holland, Peter Benson, Thomas Storer, and Richard Volz for serving as my committee members. I also wish to thank Arnie Rosenthal for serving on my committee. (He was unable to attend my defense, and therefore is not part of my official doctoral committee.) I am especially grateful to John Holland for suggesting the topic of this research and for giving so generously of his time. I have thoroughly enjoyed and benefitted greatly from our many hours of discussion related to this thesis.

This research was partially funded by the National Science Foundation, grant number MCS76-04297, through the Logic of Computers Group. I wish to thank the members of that group for their support and friendship.

Finally, I wish to thank my wife Kathy for her encouragement and patience. She sometimes had more faith in me than I had in myself.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION AND OVERVIEW

This research is concerned with the use of genetic algorithms as function optimizers. We will first describe the function optimization problem. Then we will describe some traditional approaches to solving it. Next we will describe a class of non-traditional optimization methods known as genetic algorithms. Finally, we will give an overview of the remainder of this dissertation.

1.1 The Function Optimization Problem

Function optimization problems arise in such diverse areas as artificial intelligence, on-line control of industrial processes, the design of statistical experiments, wind tunnel testing of aircraft, and so on. The general (unconstrained) function optimization problem has the following form. Given a function f : D --> R, find the global maximum (or minimum) of f. The function to be optimized is called the objective function and its domain is the search space. The range R is nearly always a subset of the real numbers -- either an interval or perhaps a set of in-

tegers. The search space, on the other hand, may be a vector space over the reals, a vector space over a finite field, a cross-product of different types of subspaces, or even a non-numerical space. We will restrict our attention to the problem of optimizing a real-valued function of several real variables[1].

A constrained optimization problem includes relations which must be satisfied by the optimal point. These constraints effectively divide the search space into two parts. The points which satisfy the constraints are called feasible points; those which do not are infeasible points. The distinction between constrained and unconstrained optimization is meaningful only when the search space has a certain representation (as a cross-product of subspaces for example) in which the constraints are not trivially expressed (so that it is not immediately apparent if a given point is a feasible point), or the optimization technique cannot easily avoid infeasible points.

The usual approach to solving a constrained optimization problem is to make it into an unconstrained problem and solve the unconstrained problem. One way of doing this is to incorporate the constraints into the objective function by adding a penalty for violating the constraints. It

---

[1]We will be looking at functions of binary strings when we begin analyzing genetic algorithms as function optimizers. But this is because we are representing real values in a binary form.

is very often the case that finding feasible points is far more difficult than maximizing the original objective function. We will not be concerned with constrained optimization problems.

1.2 Traditional Optimization Methods

Finding the optimum of an arbitrary real-valued function is a difficult problem. The usual approach to such a difficult problem is to make some simplifying assumptions about the problem and hope that the solution obtained can be easily extended to solve the original problem. In particular, if the objective function is assumed to be differentiable or perhaps twice differentiable it is possible to use calculus-based methods to find the optimum.

One of the oldest and simplest calculus-based methods is Newton's method [9], [14], [15]. Newton's method is based on taking just the first two terms from the Taylor series expansion of a function to get a linear approximation to it. This linear approximation is used to estimate where the function will be zero (see figure 1.2.1):

x_k = current point in search space

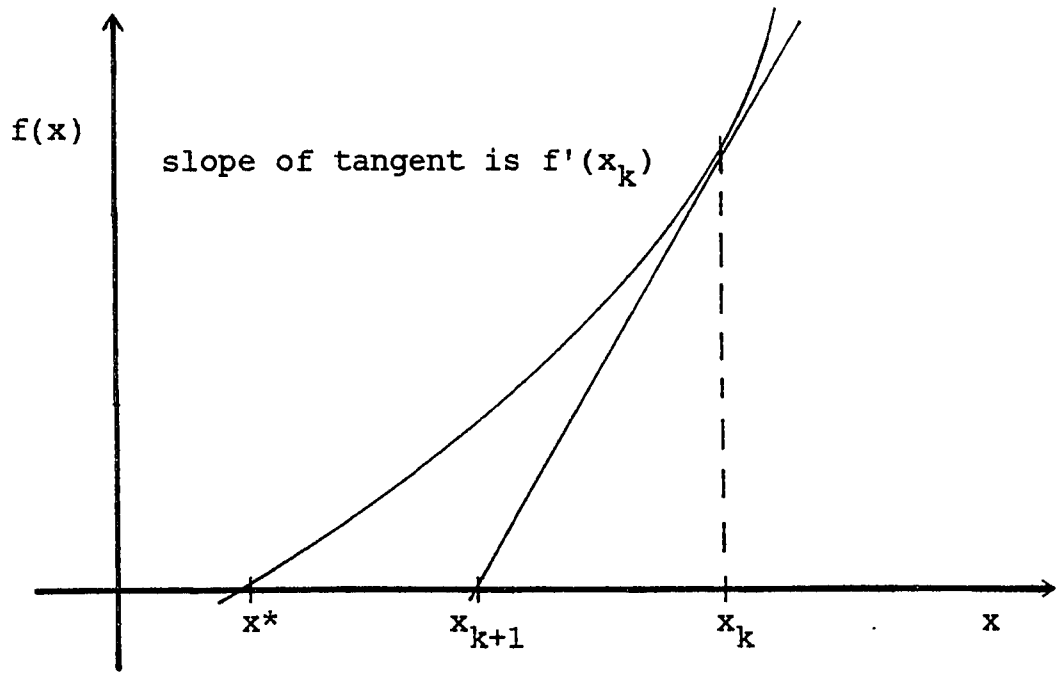
$$f(x) \cong f(x_k) + (x - x_k)\, f'(x_k)$$

Assuming $f(x^*) = 0$ gives us

$$0 \cong f(x_k) + (x^* - x_k)\, f'(x_k)$$

$$x^* \cong x_k - \frac{f(x_k)}{f'(x_k)}$$

NEWTON'S METHOD



The next point is found using a linear approximation to f:

$$0 = f(x^*) \cong f(x_k) + (x^* - x_k)\, f'(x_k)$$

$$x^* \cong x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

FIGURE 1.2.1 Newton's Method

So the sequence of points to be tried is given by:

x_0 = some initial point

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \qquad k \geq 0.$$

As described so far, Newton's method is not an optimization method but a root finder. To use Newton's method as an optimizer, we apply it to the derivative of the objective function. So if f is the objective function, the basic relation is:

x_0 = some initial point

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \qquad k \geq 0.$$

Newton's method can be applied to functions of more than one real variable -- we use the gradient vector in place of the derivative, and the Hessian matrix in place of the second derivative. The basic formulas are:

$\bar{x}_0$ = some initial point

$$\bar{x}_{k+1} = \bar{x}_k - \bar{g}(\bar{x}_k)\, H^{-1}(\bar{x}_k), \quad k \geq 0 \qquad (1.2.2)$$

where $\bar{g}(\bar{x}_k)$ = gradient of f at $\bar{x}_k$,

and $H^{-1}(\bar{x}_k)$ = inverse of Hessian of f at $\bar{x}_k$.

Another simple optimization method is called the method of steepest ascent (or descent if minimizing). The gradient vector points in the direction of the most rapid increase in function value. So we may choose our next point by making a step in that direction:

$\bar{x}_0$ = some initial point

$$\bar{x}_{k+1} = \bar{x}_k + a_k \bar{g}(\bar{x}_k), \quad k \geq 0 \qquad (1.2.3)$$

where a_k is some suitably chosen value, not fully specified here.

Most of the traditional optimization techniques are based on equations 1.2.2 or 1.2.3 or similar relations derived by assuming the objective function is differentiable or twice differentiable, is approximately quadratic (at least near the optimum), and has a single critical point. (A critical point is any point at which the gradient is zero. This may be a local maximum, a local minimum, or a saddle point.) When the objective function is unimodal and twice differentiable, and the search space is not high dimensional, these techniques work very well. In fact, the traditional methods work reasonably well on most differentiable, unimodal objective functions. But when the objective function is not differentiable, or is not unimodal, or the dimensionality of the search space is very large, such techniques either fail to converge or converge to badly suboptimal points.

1.3 A Non-Traditional Approach

Instead of basing our search strategy on the geometry of quasi-quadratic functions, we will make the simplifying assumption that the search space contains regions of good objective function values and regions of poor values. Naturally, we wish to concentrate our exploratory efforts

in the good regions of the search space. One class of simple algorithms which does this is genetic algorithms. Genetic algorithms will be discussed more completely in the next chapter. Here we present only a very sketchy outline of what a genetic algorithm is.

In order to use a genetic algorithm, the points in the search space must be represented as finite length strings. We discuss various representations in chapter 4. A genetic algorithm maintains a collection (population) of strings and explores the search space by generating successive populations which are distributed differently in the search space. Hopefully, the distribution changes so as to cluster about the optimum as the algorithm proceeds.

The overall algorithm consists of choosing a random initial population, and then generating successive populations using the following basic cycle:

1. Each string is replicated in proportion to its function value to generate an intermediate population.
2. Strings from the intermediate population are chosen randomly (each string is equally likely to be chosen) to be modified by one or more "genetic" operators and placed into the new population.

The net effect of this process is to concentrate most of the new strings in the neighborhoods of the better old strings, which is what we said we wanted to do.

1.4 Overview

The ideal optimization package would take an optimization problem defined by the user, choose an appropriate method for solving the problem[2], and then solve it. To define an optimization problem, the user must specify an objective function, a set of constraints, and a cost function. The optimization package would attempt to optimize the objective function subject to the given constraints using the method which costs the least.

The objective function may be specified by a subroutine which computes the value of the function. The minimum information required by the optimization package would be a specification of the domain of the objective function (the search space) -- the number of independent variables and range for each variable. In order to facilitate the choice of a suitable optimization technique, it would be desirable to allow the user to give additional information about the objective function, such as how to compute the gradient or whether the function is continuous or unimodal or noisy, whenever such information is available.

---

[2]For some problems, it may be wise to use two or more optimization techniques in combination -- especially when no single technique has a high probability of success. Deciding how to solve the problem posed by the user is itself an optimization problem. Hopefully a small amount of effort would usually be sufficient to choose a good strategy.

In choosing an appropriate method two factors must be balanced -- the cost should be minimized while the probability of finding the true optimum should be maximized. Consider optimizing a function of many variables which is known to have continuous partial derivatives but which is not known to be unimodal. Using a hill-climbing method would be relatively cheap, but not certain to find the true optimum. Enumerative search, on the other hand, would ensure that the optimum would be found, but at very great expense. Somewhere in between, genetic optimization would be less expensive than enumerative search with a greater chance of finding the true optimum than hill-climbing.

It is therefore desirable to be able to estimate the probability of eventually finding the optimum for each technique to rule out those for which the given problem is essentially unsolvable. As a first attempt at estimating this probability, one might try to describe the class of objective functions for which each optimization method gives reasonable performance. For the more common methods, such as conjugate direction and gradient-based methods, sufficient conditions which ensure rapid convergence to the true optimum are already known. For genetic algorithms, such conditions are developed in this dissertation. We shall concentrate on characterizing the class of functions on which genetic algorithms perform well.

In chapter 2 we describe genetic algorithms and present the results of previous research using them as function optimizers.

In sections 3.1 and 3.2 we introduce the Walsh functions and the Walsh transform. In the remainder of chapter 3 we use the Walsh transform to analyze the behavior of genetic algorithms when the search space is considered to consist of binary strings. This gives us some sufficient conditions which ensure good performance on functions of binary strings.

In chapter 4 we use some of the results from chapter 3 to find conditions which ensure that a function of one real variable can be easily optimized by a genetic algorithm. The analysis is done for several representations, but the only usuable results are for fixed-point encoding and floating-point encoding.

Chapter 5 summarizes our results and presents some ideas for further research.

CHAPTER 2

USING GENETIC ALGORITHMS AS FUNCTION OPTIMIZERS

2.1 Genetic Algorithms

The standard methods for optimizing a function are based on using the gradient and possibly higher derivatives of the objective function to guide the search. Such techniques require that the objective function possess the necessary derivatives. When the objective function is locally quadratic and the search space is not high dimensional, these techniques work very well. In fact, previous problems associated with instabilities due to round off error have been largely eliminated and symbolic optimization (algebraic manipulation of the partial derivatives and objective function) combined with "rational-mode" computations [9] gives very rapid convergence to the optimum with great accuracy. But when the objective function is not differentiable, or is not unimodal, or the dimensionality of the search space is large, such techniques perform very poorly.

In order to solve such optimization problems without resorting to random search or enumerative search we may try

the following approach. Instead of basing our search strategy on the geometry of quasi-quadratic functions, we make the assumption that the search space contains regions of good objective function values and regions of poor values. Our search method will be based on the simple idea that we want to expend most of our effort searching through the better regions. To acheive this, we might keep a table of the function values observed for each region of the search space. Most new points to be tested would come from the best regions (those with the most good objective function values), and as new points are tested we would update our table. As the search progressed, we would hope to zero in on the optimum by considering smaller and smaller subregions of the search space for further exploration.

At this point it is convenient to introduce the following notational conventions. We will be dealing with strings and binary representations of integers and real numbers. We will also be referring to families of functions and indexed sets. We reserve the use of subscripts to denote the bits within a binary number or the characters of a string. So we will write

$$j = j_1 j_2 \cdots j_\ell$$

to indicate that $j_1 j_2 \cdots j_\ell$ is the binary representation of j. Similarly

$$h = h_1 h_2 \cdots h_\ell$$

indicates that h is a character string consisting of the

ℓ characters h_1, h_2, and so on. We will generally use the letters i, j, k, m, n, p, and s as integers. The letter h will generally denote a schema (defined below). The length of the strings we are using will be given by ℓ.

Suppose we represent the points in the search space as binary strings of length ℓ. Then we may divide the search space into a large number of regions of varying size called schemata.

Definition 2.1.1

Let h be a string of length ℓ made up of the characters 0, 1, #. A binary string $s = s_1 s_2 \ldots s_\ell$ is an instance of the pattern h if and only if $s_p = h_p$ or h_p = #, for each p = 1, 2, ..., ℓ. That is, s must match h except where h contains a # ("don't care"). A schema (or hyperplane) is a subset of the set of binary strings of length ℓ consisting of all binary strings which match a specified pattern h. We will use h as the name of this schema and will refer to points in the search space as members of h or as instances of h.

$$h = \{k \mid h_p \neq \# \Longrightarrow k_p = h_p, \text{ for } p = 1, 2, \ldots, \ell\}.$$

As an example of definition 2.1.1, if ℓ = 3 then 1## denotes the set {100, 101, 110, 111}. That is, 1## is the set of strings which start with 1. Similarly, #01 denotes the set {001, 101}. We remark that each point in the search space is an instance of 2^ℓ schemata. To see this,

let $s = s_1 s_2 \ldots s_\ell$ be a binary string of length ℓ. Then s is an instance of any schema $h = h_1 h_2 \ldots h_\ell$ which satisfies $h_p = s_p$ or $h_p = \#$, for $p = 1, 2, \ldots, \ell$.

Definition 2.1.2

The order of a schema h, o(h), is the number of ones and zeroes in h. For example, with $\ell = 3$,

o(1##) = 1

o(#01) = 2

o(###) = 0.

Notice that the order of a schema must be between 0 and ℓ. Also, the number of points contained in a schema is a simple function of its order:

$|h| = 2^{\ell - o(h)}$.

Returning to the optimization method we began describing earlier, we may now outline a more concrete procedure. To begin, a small number of points chosen at random are evaluated. This allows us to construct a rather large table of estimated utilities of different schemata where the utility of a schema is the average value of the objective function for the set of points in that schema. The next few points to be tried should be chosen so as to give the best schemata (those with high estimated utilities) the most tests. However, it is also important to explore the schemata for which we have no entries in our table and to increase our confidence in the estimated

utilities of the schemata in which we have only a few test points.

Consider the following example. We take four random samples of the objective function and construct the partial table of estimated utilities shown in figure 2.1.1. (To keep things manageable we are using only 4-bit strings.) From the table we see that 1101 may be expected to have a large function value since it lies in several good schemata. 1001, 0111, and 1111 would also be good points to try since they will provide information about the schemata #00# and #11# (about which we have no information), and they also belong to some of the better schemata.

Unfortunately, storing and updating the information on all the schemata of even a modest search space is infeasible. The number of schemata for an ℓ-bit search space is 3^{ℓ} -- for $\ell = 10$ there are 59,049 hyperplanes. Even if we only look at low order schemata, we still have too many. And we need a way of choosing new sample points which is computationally reasonable.

Fortunately, there is a class of relatively simple algorithms which allocate trials to schemata in a nearly optimal manner without the need for a large table of estimated utilities. The algorithms we are talking about are the genetic algorithms developed and analyzed by Holland [10]. A genetic algorithm maintains a collection (population) of several strings which implicitly ranks a

Assume we know these 4 values of the objective function:

f(1011) = 7 f(0101) = 7 f(1100) = 6 f(0010) = 2.

We would be able to make the following estimates of the average utility of various schemata.

h	û(h)	h	û(h)	h	û(h)
####	5.5	##01	7.0	#111	?
0###	4.5	##10	2.0	0#0#	7.0
1###	6.5	##11	7.0	0#1#	2.0
#0##	4.5	000#	?	1#0#	6.0
#1##	6.5	001#	2.0	1#1#	7.0
##0#	6.5	010#	7.0	#0#0	2.0
##1#	4.5	011#	?	#0#1	7.0
###0	4.0	100#	?	#1#0	6.0
###1	7.0	101#	7.0	#1#1	7.0
00##	2.0	110#	6.0	0##0	2.0
01##	7.0	111#	?	0##1	7.0
10##	7.0	#000	?	1##0	6.0
11##	6.0	#001	?	1##1	7.0
#00#	?	#010	2.0	00#0	2.0
#01#	4.5	#011	7.0	00#1	?
#10#	6.5	#100	6.0	01#0	?
#11#	?	#101	7.0	01#1	7.0
##00	6.0	#110	?	etcetera	

FIGURE 2.1.1 Constructing a table of Estimated Utilities

large number of schemata. Let B(t) denote the population of strings at timestep t and let A(i, t) denote the i-th string in B(t). The class of genetic algorithms we wish to consider may be described by the following:

1. Generate a random initial population of M strings, B(0). Set t = 0.
2. For i = 1, 2, 3, ..., M, randomly choose a string s(i) from B(t). The probability of choosing string A(j, t) is proportional to its utility[1] (objective function value):

$$\text{prob}[s(i) = A(j, t)] = \frac{u(A(j, t))}{M\ \hat{u}(t)}$$

$$\text{where } \hat{u}(t) = \frac{1}{M} \sum_{j=1}^{M} u(A(j, t))$$

   is the average utility of the whole population B(t).
3. Set A(i, t + 1) = s(i).
4. Modify A(i, t + 1) by applying genetic operators (described below) to it.

---

[1]Actually, the utility must be non-negative so the objective function value may not be suitable. In cases where it is not known in advance that the objective function is non-negative, the utility function may be defined by

$$u(x) = f(x) - fmin + 1,$$

where f is the objective function and fmin is the smallest value of the objective function observed so far.

5. We have now generated B(t + 1). Increment t by 1 and repeat steps 2 - 4.

We will discuss the effects of genetic operators shortly, but let us suppose for the moment that we do not apply genetic operators in step 4. The algorithm will not generate any strings which are not present in the initial population B(0). It will merely change the number of copies of each string; the best string will eventually take over the population. Although this algorithm does not directly manipulate schemata, it is easy to show that it implicitly ranks a large number of schemata based on observed utilities. The relative ranking of a schema is reflected by the rate at which the (expected) number of instances of that schema is increasing (or decreasing). Schemata with higher estimated average utilities have a larger rate of increase. In order to be more precise, we make the following definitions.

Definition 2.1.3

The set of defining positions of a schema h is the set of indices of the ones and zeroes in h. That is,

$$\delta(h) = \{p \mid h_p \neq \#\} \subseteq \{1, 2, 3, \ldots, \ell\}.$$

Definition 2.1.4

Two distinct schemata, h and h', are said to be competing schemata if they have the same set of defining positions.

For example, let h = ##1#, h' = #0#1, and h'' = ##0#. Then $\delta(h) = \delta(h'') = \{3\}$ and $\delta(h') = \{2, 4\}$. Notice that the number of defining positions is equal to the order of the schema by definition 2.1.2. Schemata h and h'' have the same set of defining positions, and so they are competing schemata. The three schemata which compete with h' are #0#0, #1#0, and #1#1. Each complete set of competing schemata forms a partition of the search space.

Under the genetic algorithm without genetic operators, the expected number of copies of a string s in the population at time t + 1 is directly proportional to the number of copies of s at time t and to the utility of s. Let N(s, t) be the number of copies of string s at time t. Then

$$E[N(s, t + 1)] = \frac{u(s)}{\hat{u}(t)} N(s, t),$$

where $\hat{u}(t)$ is the average utility of the whole population at time t. In fact, the same relation holds for all the schemata with instances in the population at time t. Let $\hat{u}(h, t)$ be the average utility of all the instances of h in the population at time t, and let N(h, t) be the number of instances of h in the population at time t. Then

$$E[N(h, t + 1)] = \sum_{s \varepsilon h} \frac{u(s)}{\hat{u}(t)} N(s, t)$$

$$= \left(\frac{1}{N(h, t)} \sum_{s \varepsilon h} u(s) N(s, t) \right) \frac{1}{\hat{u}(t)} N(h, t)$$

$$= \frac{\hat{u}(h, t)}{\hat{u}(t)} N(h, t),$$

where $\hat{u}(h, t)$ is the observed average utility of h at time t.

Holland has shown that the optimal allocation of trials to competing schemata is to give exponentially more trials to the schema with the best observed average utility than to all other competing schemata.

The "optimal allocation" of trials to competing schemata is found by minimizing the expected loss assuming that each schema is a random variable with an associated distribution of utilities. If we knew that our estimated average utilities were accurate enough to correctly identify the schema with the true best average utility, then we would simply choose all future samples from that schema. But we cannot be certain that we have correctly identified the best schema, so we must increase our confidence in the estimated utilities of all competing schemata. To do this, we must sample from all the schemata. So to minimize the expected loss, we give most trials to the observed best schema and only a few trials to the rest.

More precisely, by theorem 5.1 of Holland [7]:

Given M trials to be allocated to 2 random variables, h_1, h_2, with means $u_1 > u_2$ and variances v_1^2, v_2^2, the minimum expected loss results when the number of trials allocated to the observed best random variable is

$$N \cong \sqrt{8 \pi b^4 \ln M^2} \exp(n/2b^2)$$

where n is the number of trials allocated to the observed worst random variable, and $b = v_1/(u_1 - u_2)$.

A similar result holds for more than 2 random variables.

Increasing (or decreasing) the number of instances of each schema in proportion to its utility at each timestep, results in a geometric growth (decline) for each schema. So the genetic algorithm without genetic operators allocates trials among competing schemata in a nearly optimal manner.

Let's consider the effect of the genetic operators used in step 4. What we would hope for is that adding genetic operators will allow us to explore the search space without disrupting the nearly optimal allocation of trials to competing schemata. The two genetic operators we will add to the algorithm are crossover and mutation.

The crossover operator generates a new string by combining the first part of one string with the second part of another (see figure 2.1.2). More precisely: We choose a second string, s', from B(t) (the first string is A(i, t + 1) = s) using the same probability distribution as when choosing s. A crossover point is now chosen. If the strings have length ℓ, then there are ℓ - 1 possible crossover points (the interstices), each of which is equally likely to be chosen as the actual crossover point.

Suppose we start with these 2 strings:

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

0 1 0 1 0 0 1 0 1 1 0 1.

We choose a random crossover point and break the strings at that point.

1 0 1 1 1 1 0 ¦ 0 0 1 1 0

0 1 0 1 0 0 1 ¦ 0 1 1 0 1

Exchange final segments:

1 0 1 1 1 1 0 ¦ 0 1 1 0 1

0 1 0 1 0 0 1 ¦ 0 0 1 1 0

We now have 2 new strings:

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

0 1 0 1 0 0 1 0 0 1 1 0.

FIGURE 2.1.2 Example of the Crossover Operator

The strings are then "broken" at the crossover point and recombined so that each of the two new strings consists of the starting portion of one of the original strings followed by the terminal segment of the other. In our version of the genetic algorithm we keep only 1 of the 2 crossover products.

From the foregoing description of crossover, it should be apparent that if both original strings are members of schema h, then both crossover products will be members of h. On the other hand, if string s belongs to h and string s' does not, then the results of crossing s and s' may or may not belong to h. At least one of the two crossover products will lie in h if the crossover point does not fall between the first and last defining position of h.

Definition 2.1.5

The definition length of a schema h is the difference between the first (smallest) and last (largest) defining position of h:

$$\ell(h) = \max(\delta(h)) - \min(\delta(h)).$$

The definition length of ##...# is taken to be zero. We note that the definition length of a schema may range from 0 to $\ell - 1$ inclusive.

Let $\hat{u}(h, t)$ be the average utility of all strings in B(t) which are members of h, let P(h, t) be the proportion of strings in B(t) which are instances of h (P(h, t) =

N(h, t)/M), and let $\hat{u}(t)$ be the average utility of all strings in B(t). If s is an instance of h, and s and s' are combined using crossover, then the probability that one of the resultants belongs to h is at least

$$1 - \frac{\ell(h)}{\ell - 1}\left[1 - \frac{\hat{u}(h, t)}{\hat{u}(t)} P(h, t)\right].$$

This is easily seen by noting that the probability of the crossover point falling within $\delta(h)$ is just

$$\frac{\ell(h)}{\ell - 1}$$

and the probability that $s' \notin h$ is

$$1 - \frac{1}{M} \sum_{s \varepsilon h} \frac{u(s)}{\hat{u}(t)} N(s, t)$$

$$= 1 - \frac{\hat{u}(h, t)}{\hat{u}(t)} \frac{N(h, t)}{M}$$

$$= 1 - \frac{\hat{u}(h, t)}{\hat{u}(t)} P(h, t).$$

So although using the crossover operator is likely to badly disrupt the allocation of trials among the long-definition schemata, it is unlikely to significantly affect the allocation of trials to the short-definition schemata. Moreover, crossover generates new instances of the schemata present in the current population as well as generating instances of schemata not already represented in the popula-

tion. This is almost exactly what we had hoped for.

The mutation operator generates a new string by altering one or more bits of a string. In applying the mutation operator to a string, we consider mutating each bit of the string independently of the other bits. The probability of mutating a given bit (referred to as p_m or "mutation probability" later) is a parameter of the genetic algorithm which we keep fixed during the course of an optimization run.[2] So if the string s belongs to schema h, the probability that s', the string resulting from applying the mutation operator to s, is also an instance of h is just

$$\text{Prob}(s' \in h \mid s \in h) = 1 - (1 - p_m)^{o(h)}$$
$$\cong o(h)\, p_m \qquad (p_m << 1).$$

So the mutation operator is more likely to significantly disrupt the allocation of trials to high order schemata than to low order schemata. We note that short-definition schemata necessarily have small orders. Thus neither the crossover operator nor the mutation operator will interfere appreciably with the nearly optimal allocation of trials to the short-definition schemata.

The following lemma, essentially Corollary 6.2.4 from

---

[2] It is possible to adjust the mutation probability during a run, but we do not consider doing so. Cavicchio [3] shows that some improvement is possible by properly changing the mutation probability and other parameters during a run, but we do not feel the added complexity is justified by the limited improvement when genetic algorithms are used as function optimizers.

Holland [10], shows the extent to which crossover and mutation interfere with the proper allocation of trials to competing schemata.

Lemma 2.1.6

For a genetic algorithm of the form shown in figure 2.1.3 with crossover probability p_c and mutation probability p_m, the expected proportion of schema h in the population at time t + 1 is at least

$$E[P(h, t+1)] \geq$$

$$\left(1 - p_c \frac{\ell(h)}{\ell - 1}\left[1 - \frac{\hat{u}(h, t)}{\hat{u}(t)}\right]\right)(1 - p_m)^{o(h)}\left(\frac{\hat{u}(h, t)}{\hat{u}(t)}\right)P(h,t)$$

The value of the mutation operator as a means to explore the search space is questionable. A genetic algorithm using mutation as the only genetic operator would be a random search which is biased toward sampling better hyperplanes rather than poorer ones. In order to explore the space reasonablely quickly, we would want p_m large. But that would disrupt the sampling rates for even the short schemata, and we would have essentially a pure random search. If p_m is small, the search would be very slow since the application of the mutation operator would often leave a string unchanged (variations of short schemata would be very infrequent).

The real value of the mutation operator is that muta-

tion serves as a means of restoring lost bit values. Borrowing terminology from genetics, we refer to the 2 possible values of each bit as alleles, and to each position (from 1 to ℓ) of a bit within a string as a gene locus or simply a gene. When all or very nearly all members of a population have the same allele for a particular gene, one says that gene has been fixed or that the alternative alleles have been lost. Using only crossover, without mutation, a genetic algorithm tends to rapidly fix many genes. Some of the lost alleles would prove useful later in the search although they were not of use in the early part of the search. Without the mutation operator, these alleles would be permanantly lost. From the function optimization viewpoint, the mutation operator may be viewed as a combination of a random search method and a local search method. It ensures that the population maintains a reasonable variability and also provides the means for making small changes to a string (crossover tends to make large changes since it combines 2 strings). For a more complete analysis of the roles of crossover and mutation, see Holland [10] and De Jong [6]. [2] studies the effects of using each of the operators alone and various combinations of both operators for 2 particular objective functions.

It is possible to use other genetic operators, but we shall not do so. We shall, however, present 2 modifica-

tions to the basic algorithm which significantly improve its performance, are easy to implement, and do not require excessive additional computational effort. We give only a brief description of these modifications. For a more comprehensive study, consult De Jong [6].

Many researchers have pointed out that genetic algorithms often converge too rapidly to a sub-optimal point (see Cavicchio [5] and De Jong [6] in particular). That is, after a few generations (iterations of steps 2 - 4) all the strings in the population are nearly identical but none of them represents the optimum. Further progress is very unlikely since applying crossover will not produce a new string and using mutation produces strings which are worse than the original string. The only way to improve the strings at this point is to change several alleles at once because the population has clustered about a false peak. De Jong [6] shows that one of the causes, probably the principal cause, of such premature convergence is "genetic drift". Genetic drift refers to the alteration of allele frequencies due solely to stochastic effects associated with small populations -- primarily sampling variation and the constraint that each string must have an integral number of offspring.

As an extreme example of genetic drift, suppose our population consists of only two strings, s^1 and s^2. Further suppose the two strings have equal function values:

$u(s^1) = u(s^2) = 1$. We now choose a string from the population, modify it by use of a genetic operator, and it becomes the first string of the new population. We next choose another string from the population independent of which string we chose the first time. This string is modified and becomes the second and final string in the new population. The probability that the new population consists of two strings with a single common ancestor is 1/2, even though both s^1 and s^2 are "expected" to have one offspring in the new population. Ignoring the effects of the genetic operators, we would expect the population to consist of two identical strings after only a few generations.

The source of genetic drift in a genetic algorithm is the difference between the expected number of offspring for a given string and the actual number of offspring for that string. De Jong forces this difference to be small by modifying the algorithm to count the number of offspring for each string for the current generation. Once a string has had its expected number of offspring, it can no longer be chosen as a parent. So the number of offspring for each string must be at most the expected number rounded up.

The second modification to the genetic algorithm is simply to retain the best string each time we generate a new population. This is accomplished by adding the best string from B(t) to B(t + 1) as A(M + 1, t + 1) if it is

not already in B(t + 1). So the genetic algorithm now has the form shown in figure 2.1.3.

2.2 Studies of Function Optimization by Genetic Algorithms

Holland [10] has done a theoretical analysis of genetic algorithms as adaptive search strategies. His analysis shows genetic algorithms to be robust -- they perform well for a wide range of utility (objective) functions. This is due primarily to the fact that each trial of a point in the search space is a useful sample from a very large number of schemata. Holland refers to this property as implicit parallelism. Holland's analysis is done primarily in terms of expected values. Unfortunately, as we saw in examining genetic drift, the variances involved may be significant in any particular implementation of a genetic plan -- especially if the population is relatively small. Many researchers have noted that genetic algorithms may perform significantly worse than "expected" (using populations of 20 to 100 strings). See Cavicchio [5], De Jong [6], and Martin [13] for example. Several empirical studies have therefore been undertaken to investigate the performance of genetic algorithms for particular types of utility functions.

Bosworth, Foo, and Zeigler [4] did one of the earlier studies of genetic plans as function optimizers. They compared genetic plans to conjugate gradient methods for a

Generate a random initial population of M strings, B(0).

For t = 0, 1, 2, 3, ... repeat the following:

Set offspring counters (we will count down to zero):

$$c(i) = \frac{u(A(i, t))}{\hat{u}(t)}, \quad i = 1, 2, 3, \ldots, M$$

For i = 1, 2, ..., M, produce A(i, t+1) as follows:

Choose a string A(j, t) from B(t). The probability of choosing each string is proportional to its utility. If $c(j) \leq 0$, choose another string from B(t). For each value of i, we keep trying (this rarely requires more than 2 attempts) until we have chosen A(j, t) with $c(j) > 0$.

Choose a random number, r, between zero and one using a uniform distribution.

If $r \leq P_C$ then

Choose another string A(j', t) from B(t). Again the probability of choosing each string is proportional to its payoff and we require $c(j') > 0$.

Decrement c(j) and c(j') by 1/2.

Combine A(j, t) and A(j', t) using crossover to get A(i, t+1).

Else $(r > P_C)$

Decrement c(j) by 1.

Set A(i, t+1) = A(j, t).

Apply the mutation operator to A(i, t+1).

If the best string from B(t) is not in B(t + 1), add it as A(M +1, t+1).

FIGURE 2.1.3 A Genetic Algorithm

variety of objective functions. The genetic algorithm used in this study was somewhat different from the algorithms described in the previous section. Each string was an n-tuple of real numbers (so there were a very large number of alleles for each gene locus). Moreover, the mutation operator simultaneously changed all positions of a string slightly. The mutation operator was therefore a local search technique whereas crossover was a global search technique. The main conclusions of this research were that genetic algorithms could not compete with conjugate gradient techniques if the objective function was smooth, unimodal, and low dimensional. If the objective function was not smooth or was smooth but multimodal, the genetic algorithm was superior. (The conjugate gradient method usually failed totally.) As the dimensionality increased, the conjugate gradient techniques took longer and longer to find less nearly optimal points. The increase in running time for the genetic algorithm was much less and the ultimate value acheived almost unaffected as the dimensionality increased.

Zeigler, Bosworth, and Bethke [23] extend these results to the case of noisy functions. A noisy function is an objective function which is perturbed by the addition of a small (usually normally distributed) random number each time it is sampled. So the same point may have slightly different utilities if tested several times. It

was found that genetic algorithms are not sensitive to noise, but conjugate gradient methods are extremely sensitive to noise. This extreme sensitivity was further shown to be caused by poor approximations to the partial derivatives of the objective function and could be eliminated by supplying accurate gradient information.

De Jong's study [6] also compares genetic algorithms to gradient-based methods. De Jong starts with the basic genetic algorithm and, noting problems with premature convergence, makes several modifications to improve its performance over an "environment" consisting of 5 objective functions. This environment included functions which were continuous, discontinuous, convex, non-convex, unimodal, multimodal, quadratic, non-quadratic, low-dimensional, high-dimensional, and even a noisy function. The algorithms giving the best overall performance are essentially the same as the algorithm described at the end of the preceeding section and shown in figure 2.1.3.

De Jong defines two measures of performance for an optimizer working on a single objective function. Local online performance is defined as

$$p(f, T) = \frac{1}{T} \sum_{t=1}^{T} f(s(t))$$

where s(t) is the t-th point generated by the optimizer, f is the objective function, and T is the length of the

performance evaluation period. Local off-line performance is defined by

$$p^*(f, T) = \frac{1}{T} \sum_{t=1}^{T} f^*(s(t))$$

$$\text{where } f^*(s(t)) = \max_{1 \leq t' \leq t} f(s(t')).$$

Off-line performance measures only progress toward the optimum whereas on-line performance emphasizes maintaining good interim performance. On-line performance might be a suitable measure for an adaptive plan which is directly controlling a real system (such as a chemical plant) in real time. Off-line performance would be appropriate when the adaptive plan is finding the optimal control settings of a real system by experimenting with a model of the system.

De Jong then defines 2 corresponding robustness measures. If E denotes a set of objective functions (an enviroment), then the global on-line performance is

$$P(E, T) = \frac{1}{|E|} \sum_{f \varepsilon E} p(f, T)$$

and global off-line performance is

$$P^*(E, T) = \frac{1}{|E|} \sum_{f \varepsilon E} p^*(f, T)$$

where $|E|$ is the number of functions in E.

Using these 2 robustness measures and the environment, E, of 5 objective functions mentioned earlier, De Jong compares a genetic algorithm to a conjugate direction method and a variable metric method. T was 6000 evaluations. He concluded that the conjugate direction method and the genetic algorithm performed about equally well on E, but that there were large differences in the local on-line and off-line measures. On the 3 functions which were smooth and unimodal, the conjugate direction technique was clearly superior. On the other 2 functions, one of which was discontinuous and the other was multimodal, the genetic algorithm was obviously better. These results are very similar to those obtained by Bosworth, Zeigler and Foo even though the genetic algorithms in the two studies are rather different. It seems that, regardless of the exact algorithm used, genetic-like algorithms are appropriate function optimizers for objective functions which are not smooth and unimodal.

2.3 An Improved Robustness Measure

There is a flaw in De Jong's robustness measures. Since the global performance is simply the average of the local performances, and since the local performance is just an average of the objective function values, simple scaling changes to some of the functions in the environment under consideration can change the relative ranking of 2 function

optimizers. For example, suppose that E consists of just 2 functions, f and g, both with a range of from 0 to 50. Then in comparing 2 optimizers we may construct the first table shown in figure 2.3.1. We are maximizing f and g, so optimizer 1 is clearly superior. On environment E', which consists of f' and g, with f' simply being twice f, we would construct the second table in figure 2.3.1. Many, if not most, optimizers would perform identical searches for the functions f and f'. In particular, genetic algorithms and conjugate direction methods would. But on environment E', optimizer 2 would be judged superior using the robustness measures discussed so far.

Such simple scaling changes can alter the relative ranking of 2 optimizers because by averaging the local performance measures we have implicitly assumed that a difference in performance on one function has the same significance as an equal difference in performance on another function, regardless of the ranges (or any other attributes) of the 2 functions. One obvious solution is to pre-scale all the functions in our test environment to give them all the same range. (Since this is a <u>test</u> environment, we assume the experimenter knows the original range of each function.) We could, for example, make all functions have the range of zero to 100[3]. We would probably

---

[3]The actual range chosen will have no effect on the relative rankings of the optimizers. All that matters is that all test functions should have the <u>same</u> range.

Comparison of 2 optimizers on E

	Optimizer 1	Optimizer 2
p(f, T)	40	20
p(g, T)	10	40
P(E, T)	25	30

Comparison of 2 optimizers on E'

	Optimizer 1	Optimizer 2
p(f', T)	80	40
p(g, T)	10	40
P(E', T)	45	40

FIGURE 2.3.1 Robustness Measure is Sensitive to Scaling Changes

still not wish to assume that all the functions were equally difficult. We might decide to use a weighted average of the local performance measures as our robustness measure[4]. There are several ways to choose the weights for the functions which make up the environment. Two of the simplest are:

1) weigh each function in proportion to a subjective difficulty rating
2) weigh each function in proportion to the performance of a random search on that function.

Tables 2.3.1 and 2.3.2 are taken from De Jong's thesis. They compare the performance of 5 different optimizers on the 5 functions in his test environment E. The

---

[4]De Jong allows for the possibility of considering some functions to be more difficult than others. His robustness measures are actually weighted averages:

$$P(E, T) = \sum_{f \varepsilon E} w(f)\ p(f, T)$$

$$P^*(E, T) = \sum_{f \varepsilon E} w(f)\ p^*(f, T)$$

where $w(f)$ is the weighting factor for function f. We assume $0 \leq w(f) \leq 1$, for $f \varepsilon E$, and

$$\sum_{f \varepsilon E} w(f) = 1.$$

The set of weights actually used in his thesis are $w(f) = 1/5$ for all $f \varepsilon E$. So De Jong does consider the functions to be equally important in determining the robustness of an optimizer.

T=6000	Local PRAXIS	Global PRAXIS	Local DFP	Global DFP	Genetic Algorithm	Random Search
p(F1, T)	.003	.003	.002	.002	.114	.36
p(F2, T)	.051	.051	.003	.003	.221	.35
p(F3, T)	-25.47	-25.47	-2.5	-18.27	-28.2	-22.7
p(F4, T)	135.39	135.39	3.95	3.95	17.62	66.3
p(F5, T)	18.65	10.52	17.14	9.63	3.34	4.82
P(E, T)	25.72	24.1	3.72	-.937	-1.38	9.83

TABLE 2.3.1 Off-line Performance on E

T=6000	Local PRAXIS	Global PRAXIS	Local DFP	Global DFP	Genetic Algorithm	Random Search
p*(F1, T)	.005	6.55	.004	4.71	2.32	26.2
p*(F2, T)	1348.6	2967.6	.042	11.57	34.76	494.05
p*(F3, T)	-16.84	-16.84	-2.5	-2.5	-26.49	-2.5
p*(F4, T)	149.57	149.57	5.62	54.62	40.73	249.6
p*(F5, T)	21.98	203.17	18.57	187.56	34.34	473.3
P*(E, T)	300.7	662.01	4.35	51.19	17.12	248.13

TABLE 2.3.2 On-line Performance on E

object is to minimize each of the 5 functions, so that smaller numbers indicate better performance. Also included in each table is the global performance or robustness measure for each optimizer for environment E. Applying the pre-scaling ideas developed above, we devise an environment, E', consisting of 5 functions which are closely related to the 5 functions in E. In particular, if f' ε E' and f ε E are corresponding functions, then

$f'(x) = a \cdot f(x) + b$, for all x in the search space. And a and b are chosen so that the range of f' is from 0 to 100:

$$a = \frac{100}{f_{max} - f_{min}} \qquad b = -a \cdot f_{min}.$$

Tables 2.3.3 and 2.3.4 show the performances of the 5 optimizers in environment E'. The 5 optimizers used in this study should (except for possible problems associated with round-off errors) perform the same searches on f and f' so the local performances are changed in the same way that f' relates to f since

$$p(f', T) = \frac{1}{T} \sum_{t=1}^{T} f'(s(t))$$

$$= \frac{1}{T} \sum_{t=1}^{T} [a \cdot f(s(t)) + b]$$

T=6000	Local PRAXIS	Global PRAXIS	Local DFP	Global DFP	Genetic Search	Random Search	weight
p(F1',T)	.004	.004	.0025	.0025	.145	.458	.0231
p(F2',T)	.0013	.0013	.00008	.00008	.00566	.00896	.00045
p(F3',T)	8.236	8.236	50.0	21.33	3.273	13.273	.6697
p(F4',T)	10.847	10.847	.316	.316	1.412	5.312	.2680
p(F5',T)	3.537	1.908	3.234	1.729	.469	.766	.0387
P(E', T)	4.525	4.199	10.711	4.676	1.061	3.964	
Equal Weights							
P(E', T)	8.560	8.497	33.697	14.437	2.592	10.354	
Weighted by Random Search							

TABLE 2.3.3 Off-line Performance on E'

T=6000	Local PRAXIS	Global PRAXIS	Local DFP	Global DFP	Genetic Search	Random Search	weight
p*(F1',T)	.006	8.33	.005	5.99	2.95	33.3	.1581
p*(F2',T)	34.527	75.977	.00108	.296	.8899	12.649	.06006
p*(F3',T)	23.927	23.927	50.0	50.0	6.382	50.0	.2374
p*(F4',T)	11.983	11.983	.450	4.376	3.263	19.997	.0950
p*(F5',T)	4.204	40.515	3.521	37.387	6.681	94.649	.4494
P*(E', T)	14.929	32.146	10.795	19.610	4.033	42.119	
Equal Weights							
P*(E', T)	10.783	30.908	13.497	30.055	5.377	62.334	
Weighted by Random Search							

TABLE 2.3.4 On-line Performance on E'

$$= a \frac{1}{T} \sum_{t=1}^{T} f(s(t)) + \frac{1}{T} \sum_{t=1}^{T} b$$

$$= a \cdot p(f, T) + b.$$

So it is a simple matter to compute the performances in E' -- it does not require any new computer runs.

The first set of robustness measurements in these tables are derived by simply averaging the performances on each of the 5 "scaled" functions. Thus the weights are all equal to 1/5. (This amounts to subjectively deciding that all 5 functions are equally difficult or equally important in the robustness measure.) First, we note that the genetic algorithm has the best (smallest -- we are minimizing here) robustness measure for both on-line and off-line performance. This was not the case for environment E. Second, we note that of the 5 optimizers only the genetic algorithm does better than random search in terms of global off-line performance. This is because the other 4 optimizers do so much worse than random search on either F3 or F4. Finally, we note that the relative ranking of the optimizers has changed considerably by properly scaling the test functions.

The second set of robustness measurements are derived using weights proportional to the performance of the random search on each function. (These weights are shown in the rightmost column of each table.) This means that those

functions on which random search worked well do not count much in the robustness measure. Again, the genetic algorithm does best in terms of both on-line and off-line performance. And again, the relative rankings of the optimizers is different from the previous rankings.

The change in rankings related to the change in the weights associated with the functions is appropriate. In deciding that performance on one (type of) function is more important than performance on another, we are favoring the optimizer(s) which performs best on that (type of) function. If we change our priorities, we must expect a corresponding change in the rankings of the optimizers. We note that we may also change the rankings of the optimizers by changing the set of functions in the environment. It is possible to assume that all conceivable (properly scaled) test functions are included in E, but only a very small number of them have non-zero weights. So adding new functions is not much different from changing the weights of the functions.

There is one more change to the robustness measure which we will consider. Instead of using the performance of the random search to determine the weights for each local performance measure, we can move the comparison to random search "back" into the local performance measure. To do this we let S be the search space consisting of 2^{ℓ} points represented by binary strings of length ℓ. We con-

sider S to be a sample space[5] and define a probability distribution over S with all points in S being equally likely. That is,

$$\text{prob}(s) = 2^{-\ell}, \text{ for all } s \in S.$$

The objective function $f : S \rightarrow R$ may now be considered a random variable. We may define

$$F(x) = \text{pr}(f \leq x) = 2^{-\ell} |B|$$

$$\text{where } B = \{s \in S \mid f(s) \leq x\}.$$

Since F is a cumulative distribution function, it has a range of 0 to 1. F(x) is just the probability of finding a value better than or equally as good as x by a single trial of random search (we are still trying to minimize the objective function). This gives a local on-line performance measure

$$\bar{p}(f, T) = \frac{1}{T} \sum_{t=1}^{T} F[f(s(t))].$$

If we define

$$F^*(x, t) = \text{pr}[f^*(t) \leq x]$$

$$= \text{pr}[\min_{1 \leq t' \leq t} f(t') \leq x]$$

$$= 1 - (\text{pr}[f > x])^t$$

$$= 1 - (1 - F(x))^t$$

(here f(t') is a sequence of independent samples of f),

---

[5]The reader who is unfamiliar with the notion of a sample space in probability theory may consult

Feller, W. An Introduction to Probability Theory and Its Applications, Volume I, John Wiley & Sons, Inc. 1957

then we have a local off-line performance measure

$$\bar{p}^*(f, T) = \frac{1}{T} \sum_{t=1}^{T} F^*[f(s(t)), t].$$

With $\bar{p}$ and $\bar{p}^*$ no pre-scaling of the objective functions is needed since F and F* supply the scaling. Using these 2 local performance measures would require a table of F and F* values for each test function. The computational effort required is clearly more than for the old local performance measures, but $\bar{p}$ and $\bar{p}^*$ provide the most direct possible comparison between an optimizer and a random search. It seems natural that if $\bar{p}$ and $\bar{p}^*$ are used as local performance measures, one would simply average the local performances (equal weights for each function) to get the robustness measures. (The different functions would implicitly be ranked by difficulty by using F and F* in the local performances.) Preparing a table of the performance of the 5 optimizers using $\bar{p}$ and $\bar{p}^*$ would require a complete new set of computer runs and we have not done so. We believe that the genetic algorithm would again prove superior, but this has not been demonstrated.

2.4 Fooling the Genetic Algorithm

We have seen that a genetic algorithm explores the search space by allocating most of its trials to the schemata with the best observed average payoffs. Moreover,

the crossover operator causes a significant disruption of the allocation of trials to longer schemata, but has little effect on the shorter schemata. This is consistent with the observations of several researchers that a genetic algorithm proceeds as follows (see De Jong [6], page 44 especially). Initially, short schemata are sampled and compared. Many trials are allocated to the short schemata with the best observed average payoffs. This generally causes some alleles to become rare, effectively reducing the dimensionality of the search space. Now longer schemata are worked on .... Eventually all the individuals in the population are nearly identical and the genetic algorithm has "converged" to some point in the search space.

Therefore, if the genetic algorithm fixes the wrong genes early in the search, it is likely to concentrate its exploration efforts in the wrong part of the search space and miss the optimum. The wrong genes may be fixed for at least 2 different reasons:

1. genetic drift -- due to stochastic effects, even though the estimated payoffs are close to the true payoffs, the wrong schemata are given too many trials, and the best alleles are lost.
2. the short schemata are misleading -- the best short definition schemata do not contain the optimum.

The first problem may be overcome by forcing the

number of offspring of each individual to be very close to the expected number and by avoiding very small populations (see [6]). The second problem seems much more difficult. It strikes at the assumptions which form the basis for the genetic algorithm. We now focus our efforts on characterizing the class of functions which have misleading short schemata. The next few paragraphs give a brief, intuitive discussion of which functions might have misleading short schemata.

Suppose the utility function were a linear combination of ℓ functions, each operating on a single gene:

$$u(s_1 s_2 \ldots s_\ell) = \sum_{p=1}^{\ell} [u(p)](s_p).$$

Then there would be no interaction between the genes; we could associate a utility with each allele and optimize the total payoff by adjusting each gene independently of the others[6]. We would expect the genetic algorithm to do quite well with such a "linear" utility function.

---

[6]Actually, the utility does not have to decompose into a linear combination of one-gene functions to allow "one gene at a time" optimization. Let $\bar{s}(q)$ denote the binary string which differs from s only at bit q. All that is necessary for successful "one gene at a time" optimization is that the utility function satisfy the following condition for $q = 1, 2, \ldots, \ell$:

if $u(s) > u(\bar{s}(q))$ for some s,

then $u(s') > u(\bar{s}'(q))$ for all s' with $s'_q = s_q$.

Now suppose the utility function were a linear combination of several simpler functions, each operating on a small number (not necessarily just one) of genes. There would be some interactions between the genes and we could not optimize the total payoff by setting each gene independently of the rest. If the genes which interact strongly are grouped close together on the strings, we would expect the genetic algorithm to optimize the total payoff quickly since the short schemata contain the information needed to locate the optimum. We will look at this situation more carefully (and from a slightly different point of view) in sections 3.4 and 3.5, and chapter 4.

Finally, suppose the utility function could not be decomposed into simpler functions operating on less than the whole string. Then the genes would all interact. If all genes interact strongly, then the short schemata would not be likely to provide much help in locating the optimum. Indeed, we may not be able to find the optimum except by exhaustive search (or by making use of special knowledge about the objective function). So we would not expect genetic algorithms to perform very well for such highly "non-linear" utility functions.

In chapter 3 we will use the discrete Walsh transform to analyze the relation between the utility function (as a function operating on binary strings) and the average utilities of the various schemata. Later, we will see

which functions are difficult-to-optimize utility functions. That is, we will consider how the representation of the objective function affects the genetic algorithm's ability to optimize it.

2.5 Summary

In this chapter we have described a class of adaptive search methods known as genetic algorithms. Previous research has shown that genetic algorithms make robust function optimizers -- they perform well on a wide variety of objective functions. We have suggested some improvements to the robustness measures used by De Jong in his study comparing genetic algorithms to 2 more common optimizers. Pre-scaling all the test functions to have the same range gives a different ranking of the optimizers -- genetic algorithms are seen to be superior in both on-line and off-line performance. We also suggested a robustness measure based on random search, but we have no results based on that measure. Finally, we discussed how the genetic algorithm might fail to optimize certain kinds of utility functions.

CHAPTER 3

WALSH SERIES ANALYSIS OF GENETIC OPTIMIZERS

3.1 Introduction

In the preceeding chapter, we saw how a genetic algorithm could be used to optimize many types of objective functions. Previous research, including both theoretical [10] and empirical [6], [4] studies, shows that genetic algorithms make robust, general-purpose function optimizers. Nonetheless, it seems clear that a genetic algorithm cannot optimize all objective functions. In this chapter we examine the class of functions for which genetic algorithms make good optimizers and look at some functions which are very difficult for the genetic algorithm.

3.2 Walsh Functions and the Discrete Walsh Transform

The set of Walsh functions was first defined by the American mathematician J. L. Walsh in 1923 [21], but only in recent years have Walsh functions appeared in technical and engineering literature [17], [18], [19]. The Walsh functions form a complete orthogonal set of functions and induce the Walsh transform in the same way that the set of

trigonometric functions sin(nx) and cos(nx) or the complex exponentials e^{inx} induce Fourier transforms. The reason for the recent interest in Walsh functions is due to the fact that the Walsh functions are digital in nature, taking only 2 values, ±1, and changing values only at a finite number of points in the orthogonality interval. As we shall see, it is natural to define the Walsh functions using the binary number system. It is correspondingly simple to design computer hardware and software to rapidly compute Walsh functions and transforms [17], [18], but we will not pursue those possibilities.

We will be dealing with the set of <u>discrete</u> Walsh functions, which may be considered as samples from the continuous Walsh functions taken at evenly spaced intervals. There are many ways of defining the set of Walsh functions (see the list in [12], and also [11]). All of these definitions lead to the same set of functions, but there are several distinct orderings of the Walsh functions. The definitions we will use result in what are usually called "binary-ordered", "dyadic-ordered", or "Walsh-Paley" functions except that our indices are bit-reversed with respect to the usual binary indices.

The index of a Walsh function will be separated from the argument by a semicolon. Thus

$$wal(j;\ k) = wal(j_1 j_2 \ldots j_\ell;\ k_1 k_2 \ldots k_\ell)$$

indicates the Walsh function with index j and argument k

where the binary representation for j is $j_1 j_2 \ldots j_\ell$ and similarly for k. The usual interpretation is that the index j is an integer between 0 and $2^\ell - 1$ inclusive,

$$j = \sum_{p=1}^{\ell} j_p 2^{p-1}$$

and that k is either an integer like j or k is a binary fraction, $0 \leq k < 1$,

$$k = \sum_{p=1}^{\ell} k_p 2^{-p}.$$

For our purposes, we choose to regard j and k simply as binary strings (and we therefore will not suppress leading zeroes in the representation of j).

We will define the Walsh functions as products of simpler functions, the Rademacher functions. The Rademacher functions are simply square waves of various frequencies. At this point it is necessary to explain that we will be working with only a finite set of functions and finite length strings[1]. We will let ℓ be the length of our strings and we will define a set of ℓ Rademacher functions and 2^ℓ Walsh functions.

---

[1]It is not strictly necessary to limit the index set and domain of the Walsh and Rademacher functions in this way. One can work with countably infinite sets instead. We do not need the more general results and will use finite sets to simplify the exposition.

Definition 3.2.1

We recursively define the discrete Rademacher functions by

$$r(1;\ k) = r(1;\ k_1k_2\ldots k_\ell) = \begin{cases} 1, & \text{if } k_1 = 0 \\ -1, & \text{if } k_1 = 1 \end{cases}$$

$$r(p;\ k) = r(p;\ k_1k_2\ldots k_\ell)$$
$$= r(p - 1;\ k_2k_3\ldots k_\ell 0),\ 1 < p \leqq \ell$$

This is the usual definition. Note that the index p is not a string but an integer between 1 and ℓ inclusive. We remark that definition 3.2.1 is the discrete counterpart of definition 3.2.2.

Definition 3.2.2

We recursively define the continuous Rademacher functions by

$$R(1;\ t) = \begin{cases} 1, & \text{if } 0 \leqq t < 1/2 \\ -1, & \text{if } 1/2 \leqq t < 1 \end{cases}$$

$$R(p;\ t) = R(p - 1;\ 2t \bmod 1),\ p > 1$$

where t is a real number, $0 \leqq t < 1$, and p is an integer, $p \geqq 1$.

Figure 3.2.1 shows the discrete Rademacher functions for $\ell = 3$ and the corresponding continuous Rademacher functions.

Another way of viewing the Rademacher functions is to think of the domain of binary strings of length ℓ as members of an ℓ-dimensional space rather than as integers

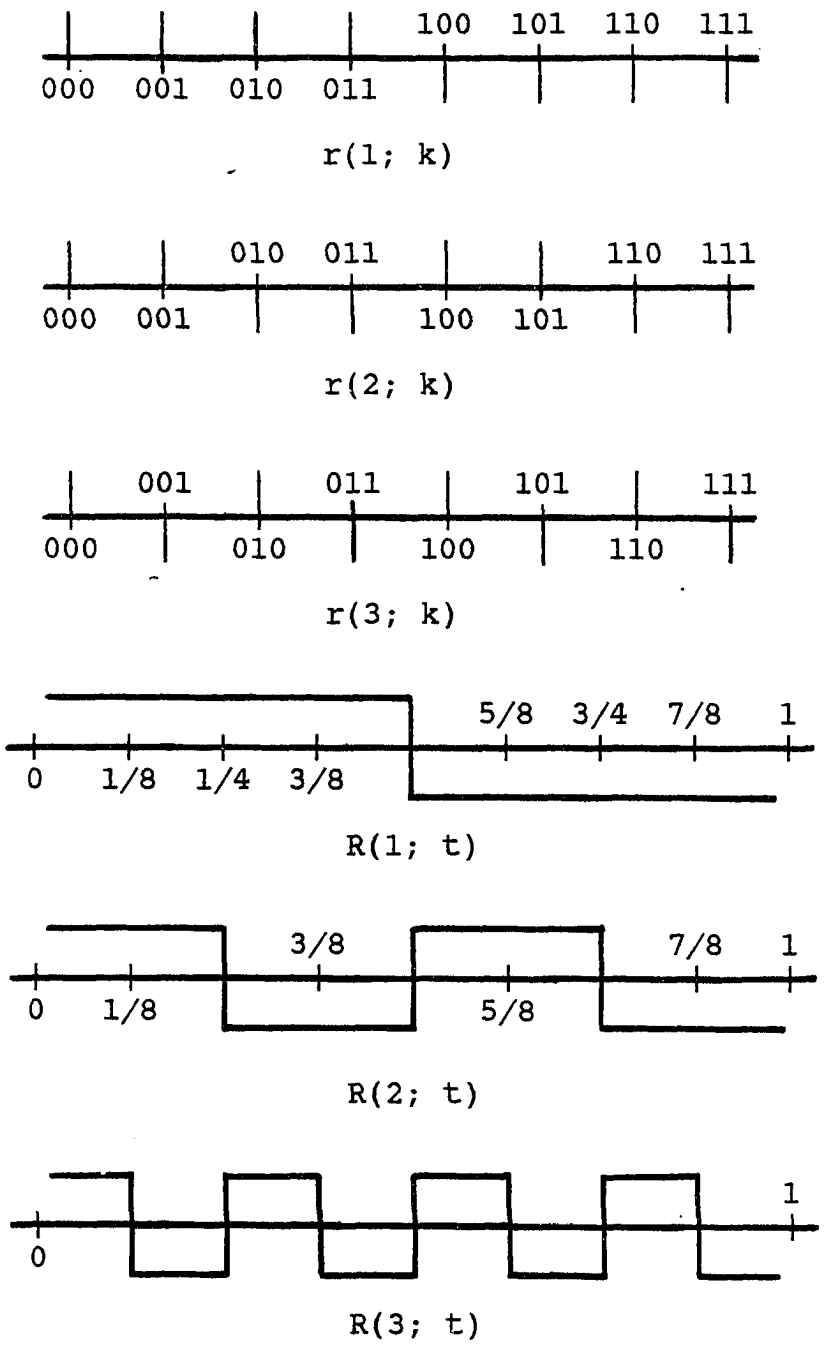


FIGURE 3.2.1 Discrete and Continuous Rademacher Functions

or fractions. From this point of view, the Rademacher functions are simply slight modifications of projections of the different coordinates. This has implications with respect to schemata as we shall see shortly.

Definition 3.2.3

The discrete (binary-ordered) Walsh functions are defined as products of Rademacher functions:

$$wal(j;\ k) = wal(j_1 j_2 \cdots j_\ell;\ k)$$

$$= \prod_{p=1}^{\ell} [r(p;\ k)]^{j_p}$$

Both j and k, the index and the argument, are binary strings of length ℓ. So this defines a set of 2^ℓ functions. Each Walsh function is the product of those Rademacher functions whose indices p correspond to the 1 bits in the index j. Here we have used a bit-reversed ordering -- usually the bits in j are numbered from right to left, not left to right (and usually from 0 to $\ell - 1$, not 1 to ℓ). This ordering is more convenient for our purposes and does not change any of the properties usually associated with the binary-ordered Walsh functions[2]. The

[2]The bit-reversed ordering actually simplifies some of the formulas slightly (the subscripts are usually shorter). Also, the fast Walsh transform, based on the Cooley-Tukey fast Fourier transform, runs slightly faster since the Walsh coefficients are computed in "bit-reversed" order (which is the order we want) and need not be sorted into "normal" order.

eight discrete Walsh functions for $\ell = 3$ are shown in figure 3.2.2.

We notice that wal(00...0; k) is a constant function with value 1 regardless of k. Further, the Rademacher functions form a subset of the Walsh functions since

$$r(q;\ k) = wal(j(q);\ k),$$

$$\text{where } j(q) = j_1 j_2 \cdots j_\ell,\ j_p = \begin{cases} 0, & \text{if } p \neq q \\ 1, & \text{if } p = q. \end{cases}$$

Noting that

$$wal(j'(q);\ k) = [r(q;\ k)]^{j_q}$$

$$\text{where } j'(q) = j'_1 j'_2 \cdots j'_\ell,\ j'_p = \begin{cases} 0, & \text{if } p \neq q \\ j_p, & \text{if } p = q \end{cases}$$

we have shown lemma 3.2.4.

Lemma 3.2.4

$$wal(j_1 j_2 \cdots j_\ell;\ k) = \prod_{q=1}^{\ell} wal(j'(q);\ k).$$

The following lemmas are proved in the appendix.

Lemma 3.2.5

$$r(p;\ k) = \begin{cases} 1, & \text{if } k_p = 0 \\ -1, & \text{if } k_p = 1 \end{cases}$$

Lemma 3.2.6

$$wal(j_1 j_2 \cdots j_\ell;\ k_1 k_2 \cdots k_\ell) = (-1)^z$$

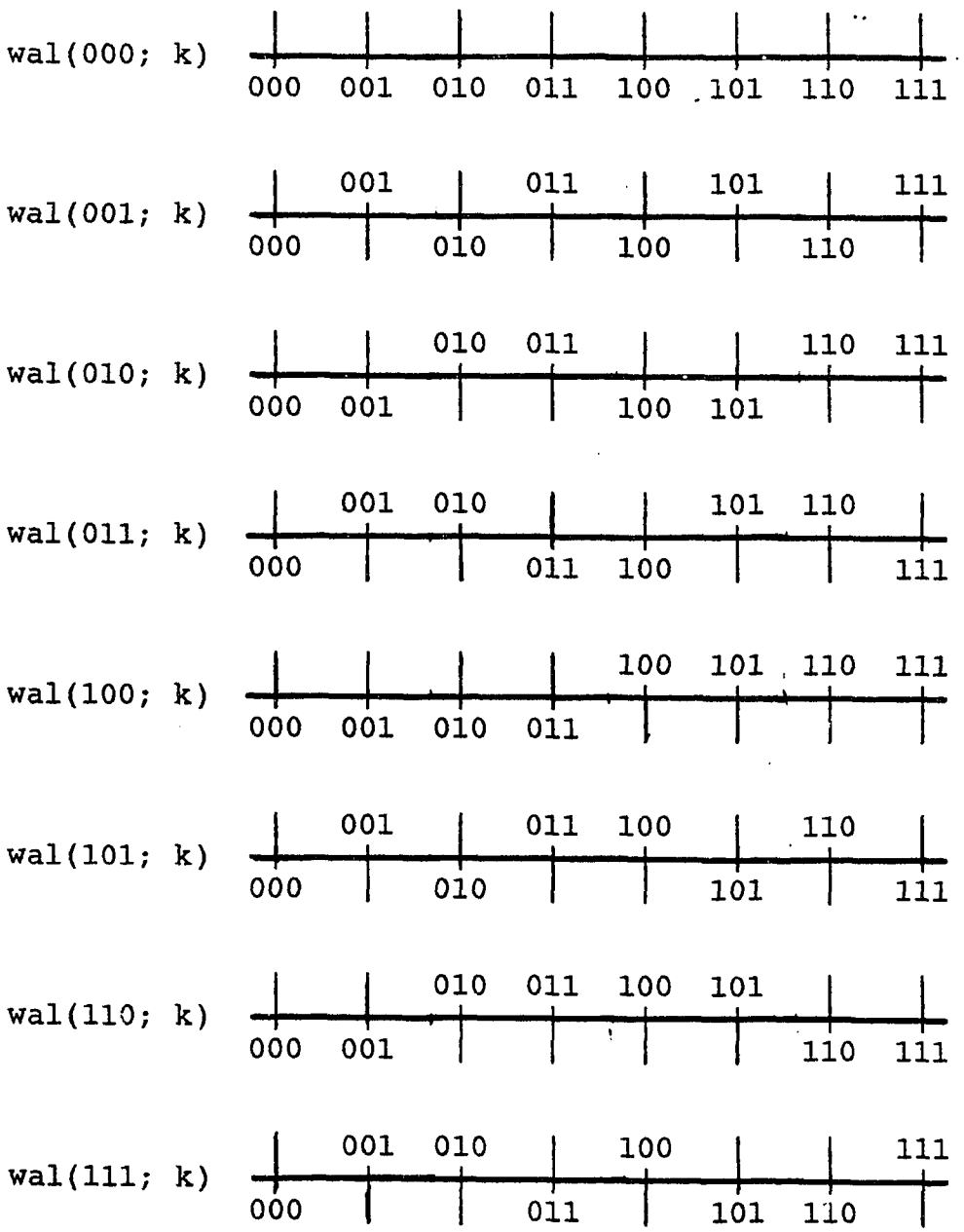


FIGURE 3.2.2 The Set of 8 Discrete Walsh Functions for $\ell = 3$

$$\text{where } z = \sum_{p=1}^{\ell} j_p k_p.$$

Lemma 3.2.7

The set of discrete Walsh functions is orthogonal:

$$\sum_{k} \text{wal}(m;\ k)\ \text{wal}(n;\ k) = \begin{cases} 0, & \text{if } m \neq n \\ 2^{\ell}, & \text{if } m = n \end{cases}$$

A discrete transform like the discrete Walsh transform or discrete Fourier transform may be considered to be an approximation to the corresponding continuous transform or may be viewed as a linear transformation defined on a vector space. From either point of view, the transformed values tell us how to synthesize the original function as a linear combination of the orthogonal functions for that transform.

Let's consider the set of real-valued functions of binary strings of length ℓ to be a vector space. Each function is a vector which consists of 2^{ℓ} real numbers. The canonical basis B for this vector space is the set of functions which are zero everywhere except for one string and take the value one for that string:

$$B = \{b(j) \mid j \text{ is a binary string of length } \ell\}$$

where $b(j) : \{0, 1\}^{\ell} \dashrightarrow R$ is defined by

$$b(j;\ k) = \begin{cases} 0, & \text{if } j \neq k \\ 1, & \text{if } j = k. \end{cases}$$

The canonical basis functions for $\ell = 3$ are shown in figure 3.2.3. The eight Walsh functions for $\ell = 3$ (see figure 3.2.2) also form a basis for this vector space since they are linearly independent (orthogonality implies linear independence). Therefore, any real-valued function defined on binary strings of length ℓ can be represented as a linear combination of discrete Walsh functions. So the discrete Walsh transform is really a change-of-basis transformation[3].

Definition 3.2.9

If f is a real-valued function of binary strings of length ℓ, then the (discrete) Walsh transform of f is the function w such that

$$f(k) = \sum_{n} w(n)\ wal(n;\ k).$$

w is a real-valued function of binary strings of length ℓ.

Lemma 3.2.10

If w is the Walsh transform of f, then

$$w(m) = 2^{-\ell} \sum_{k} f(k)\ wal(m;\ k).$$

---

[3]The matrix representing this change-of-basis transformation is

$$H = [h(j;\ k)],\ h(j;\ k) = wal(j;\ k).$$

This is a Hadamard matrix [11].

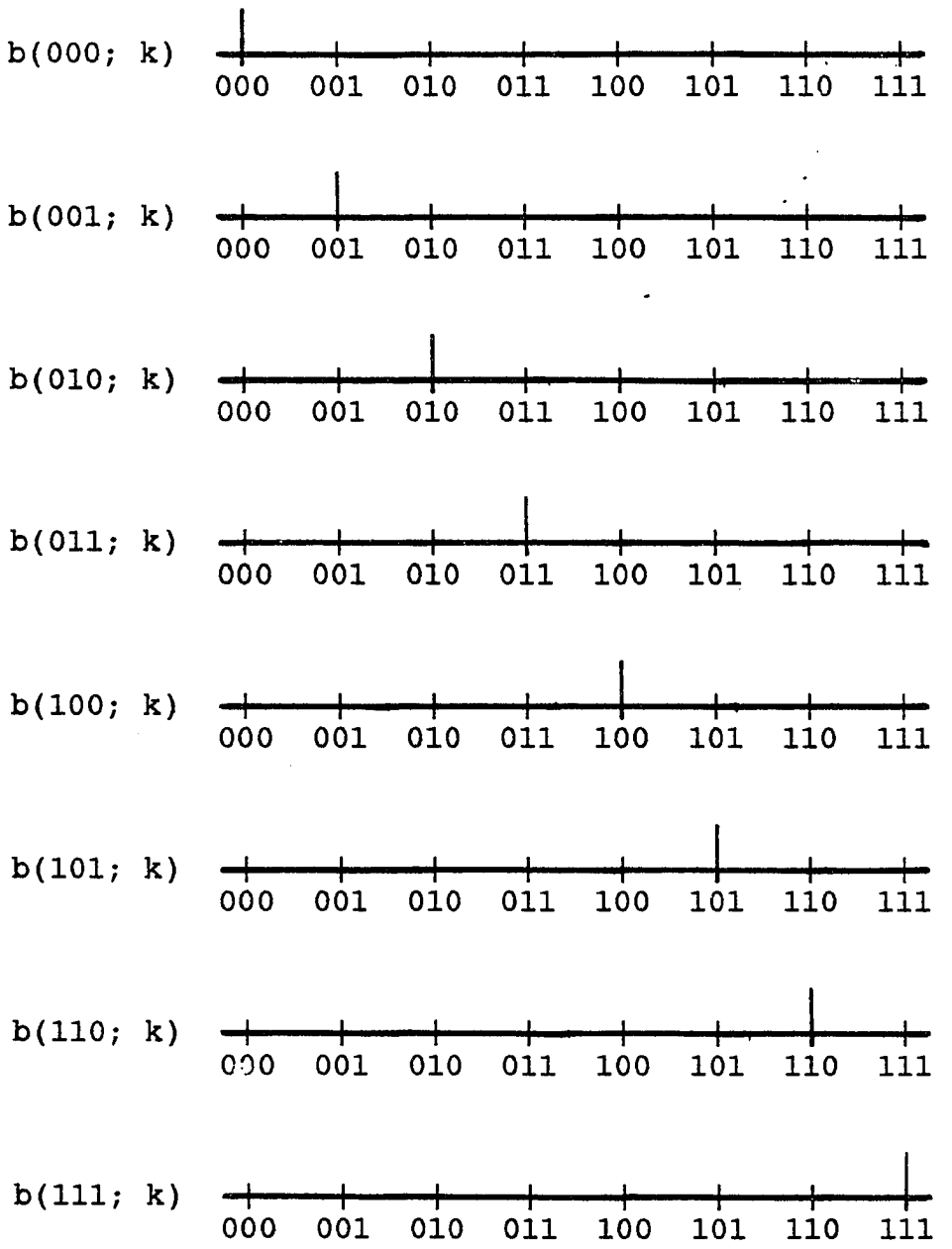


FIGURE 3.2.3 Canonical Basis Functions, $\ell = 3$

The proof of lemma 3.2.10 may be found in the appendix. We note that the Walsh transform is its own inverse (except for the $2^{-\ell}$ factor). Also, we point out the similarity between the Walsh and Fourier transforms by giving the following definition and lemma.

Definition 3.2.11

The (discrete) Fourier transform of a function f of binary strings of length ℓ is the function a such that

$$f(k) = \sum_{n} a(n)\ e^{inw_0 k}$$

$$\text{where } w_0 = \frac{2\pi}{2^{\ell}}.$$

Lemma 3.2.12

The Fourier transform of f is given by

$$a(m) = 2^{-\ell} \sum_{k} f(k)\ e^{-imw_0 k}.$$

Because of this similarity, it is possible to compute the discrete Walsh transform using the Cooley-Tukey algorithm for the fast Fourier transform but with wal(j; k) in place of $e^{-ijw_0 k}$ [17].

This concludes our discussion of Walsh functions and the Walsh transform. We now have the tools necessary for the analysis in the next section.

3.3 Schema Averages and the Walsh Transform

In this section we relate the utility of a schema to the Walsh transform of the objective function. The utilities of the various schemata will determine the way the genetic algorithm allocates its trials and, therefore, the expected time before it finds the optimum.

Definition 3.3.1

The characteristic function of a schema h is that function which has value 1 for strings belonging to h and is 0 for strings not contained in h. That is

$$c(h;\ s) = \begin{cases} 0, & \text{if } s \notin h \\ 1, & \text{if } s \in h \end{cases}$$

or equivalently

$$c(h;\ s) = \begin{cases} 0, & \text{if } s_p \neq h_p \text{ and } h_p \neq \# \text{ for some } p \\ 1, & \text{otherwise} \end{cases}$$

$(h = h_1 h_2 \ldots h_\ell,\ s = s_1 s_2 \ldots s_\ell)$

Lemma 3.3.2

Let $s = s_1 s_2 \ldots s_\ell$ be any binary string of length ℓ.

$c(\#\#\ldots\#;\ s) = 1.$

$$c(h^0(q);\ s) = \begin{cases} 0, & \text{if } s_q = 1 \\ 1, & \text{if } s_q = 0 \end{cases}$$

$$c(h^1(q);\ s) = \begin{cases} 0, & \text{if } s_q = 0 \\ 1, & \text{if } s_q = 1 \end{cases}$$

where $h^0(q)$ and $h^1(q)$ are schemata of order 1 with position q equal to 0 and 1 respectively:

$$h^0(q) = h_1^0 h_2^0 \ldots h_\ell^0, \; h_p^0 = \begin{cases} \#, & \text{if } p \neq q \\ 0, & \text{if } p = q \end{cases}$$

$$h^1(q) = h_1^1 h_2^1 \ldots h_\ell^1, \; h_p^1 = \begin{cases} \#, & \text{if } p \neq q \\ 1, & \text{if } p = q \end{cases}.$$

Proof: The lemma follows immediately from definition 3.3.1.

Lemma 3.3.3

Let s be any binary string of length ℓ, and let h be any schema, then

$$c(h; s) = c(h_1 h_2 \ldots h_\ell; s) = \sum_{q=1}^{\ell} c(h'(q); s),$$

where h'(q) is a schema of order 0 or 1 which matches h at position q:

$$h'(q) = h_1' h_2' \ldots h_\ell', \; h_p' = \begin{cases} \#, & \text{if } p \neq q \\ h_p, & \text{if } p = q \end{cases}.$$

Proof: The product will be zero if and only if $h_p \neq \#$ and $h_p \neq s_p$ for some p. By definition 3.3.1, this is c(h; s). Alternatively, one can view the product as corresponding to the intersection of the schemata h'(1), h'(2), ...,

$h'(\ell)$. Then the lemma simply states

$$h = h_1 h_2 \ldots h_\ell = \bigcap_{1 \leq q \leq \ell} h'(q).$$

We note the striking similarity between the formulas defining the Walsh functions and those defining the characteristic functions of the schemata. Compare lemma 3.2.4 to lemma 3.3.3 in particular.

Lemma 3.3.4

$$c(h^0(q);\ s) = \frac{1}{2}\ [1 + wal(j(q);\ s)],$$

$$c(h^1(q);\ s) = \frac{1}{2}\ [1 - wal(j(q);\ s)],$$

where $h^0(q)$ and $h^1(q)$ are defined as in lemma 3.3.3 and $j(q)$ is given by

$$j(q) = j_1 j_2 \ldots j_\ell,\ j_p = \begin{cases} 0, & \text{if } p \neq q \\ 1, & \text{if } p = q \end{cases}$$

Proof: If $k_p = 0$, then both sides of the first equation are 1 and both sides of the second equation are 0. If $k_p = 1$, then both sides of the first equation are 0 and both sides of the second equation are 1.

We recall from definitions 2.1.2 and 2.1.5 that the order of a schema h is the number of ones and zeroes in h and the definition length of h is the distance between the first and last defining positions of h. We similarly

define the order and definition length of a binary string as follows.

Definition 3.3.5

The order of a binary string s is the number of ones in s:

$$\text{order}(s) = o(s) = \sum_{p=1}^{\ell} s_p.$$

Definition 3.3.6

The definition length of a binary string s is the distance between the first and last ones in s:

$$\ell(s) = \max\{p \mid s_p = 1\} - \min\{p \mid s_p = 1\}.$$

The definition length of s = 00...0 is zero.

So if $\ell = 4$, s = 0101, and s' = 0110, then $o(s) = o(s') = 2$, and $\ell(s) = 2$, $\ell(s') = 1$.

In the following, we will often use the shorthand notation wal(h; s) or wal(s; h) where h is a schema and s is a binary string to stand for wal(j(h); s) or wal(s; j(h)) where j(h) is a binary string derived from h by changing # to 0:

$$\text{wal}(h;\ s) = \text{wal}(j(h);\ s)$$

$$\text{wal}(s;\ h) = \text{wal}(s;\ j(h))$$

$$\text{where } j(h) = j_1 j_2 \cdots j_\ell,\ j_p = \begin{cases} 0, & \text{if } h_p = 0, \# \\ 1, & \text{if } h_p = 1 \end{cases}.$$

(By symmetry, wal(h; s) = wal(s; h).)

The following theorem relates the characteristic function of a schema to the set of discrete Walsh functions. In fact, this theorem gives the Walsh transform of the characteristic function of a schema.

Theorem 3.3.7

Let h be a schema and let s be a binary string.

$$c(h;\ s) = 2^{-o(h)} \sum_{j \varepsilon \bar{h}} wal(j;\ h)\ wal(j;\ s)$$

where $\bar{h} = \bar{h}_1\bar{h}_2\ldots\bar{h}_\ell$, $\bar{h}_p = \begin{cases} \#, & \text{if } h_p = 0,\ 1 \\ 0, & \text{if } h_p = \# \end{cases}$.

Proof: We proceed by induction on ℓ.

Basis ($\ell = 1$). There are 3 cases: h = #, h = 0, and h = 1. In each case, we may verify the theorem by direct substitution and the application of lemma 3.3.4.

Induction Step. By lemma 3.3.3,

$$c(h;\ s) = c(h_1h_2\ldots h_{\ell-1}\#;\ s)\ c(\#\#\ldots\#h_\ell;\ s).$$

$$= c(h_1h_2\ldots h_{\ell-1};\ s_1s_2\ldots s_{\ell-1})\ c(h_\ell;\ s_\ell)$$

by definition 3.3.1. Using the induction hypothesis we have

$$c(h;\ s) = 2^{-o(h')}\ c(h_\ell;\ s_\ell) \sum_{j \varepsilon \bar{h}'} wal(j;\ h')\ wal(j;\ s')$$

where $h' = h_1 h_2 \ldots h_{\ell-1}$

$s' = s_1 s_2 \ldots s_{\ell-1}$

and $\bar{h}' = \bar{h}_1 \bar{h}_2 \ldots \bar{h}_{\ell-1}$.

We now have 3 cases: $h_\ell = \#$, $h_\ell = 0$, and $h_\ell = 1$.

Case 1 If $h_\ell = \#$, then $o(h) = o(h')$ and $c(h_\ell; s_\ell) = 1$, so

$$c(h; s) = 2^{-o(h)} \sum_{j \varepsilon \bar{h}'} wal(j; h') \, wal(j; s').$$

By lemma 3.2.4 (with $j_\ell = 0$),

$$c(h; s) = 2^{-o(h)} \sum_{j \varepsilon \bar{h}''} wal(j; h'') \, wal(j; s)$$

$$= 2^{-o(h)} \sum_{j \varepsilon \bar{h}} wal(j; h) \, wal(j; s)$$

where $h'' = h_1 h_2 \ldots h_{\ell-1} 0$

and $\bar{h}'' = \bar{h}_1 \bar{h}_2 \ldots \bar{h}_{\ell-1} 0$.

Case 2 If $h_\ell = 0$, then $o(h) = o(h') + 1$ and

$c(h_\ell, s_\ell) = \frac{1}{2}[1 + wal(1; k_\ell)]$,

so

$$c(h; s) = 2^{-o(h)} [1 + wal(1; k_\ell)] \sum_{j \varepsilon \bar{h}'} wal(j; h') \, wal(j; s')$$

where h', $\bar{h}'$, and s' are as in case 1 above. Expanding the right hand side gives

$$= 2^{-o(h)} \sum_{j \varepsilon \bar{h}'} \text{wal}(j; h') \text{wal}(j; s')$$

$$+ 2^{-o(h)} \sum_{j \varepsilon \bar{h}'} \text{wal}(j; h') \text{wal}(j; s') \text{wal}(1; k_{\ell}).$$

Applying lemma 3.2.3 several times:

$$= 2^{-o(h)} \sum_{j \varepsilon \bar{h}''} \text{wal}(j; h'') \text{wal}(j; s)$$

$$+ 2^{-o(h)} \sum_{j \varepsilon \bar{h}'''} \text{wal}(j; h'') \text{wal}(j; s)$$

where h'' is defined in case 1 and

$$h''' = h_1 h_2 \ldots h_{\ell-1} 1$$
$$\bar{h}''' = \bar{h}_1 \bar{h}_2 \ldots \bar{h}_{\ell-1} 1.$$

Noting that $\bar{h}$ is the disjoint union of $\bar{h}''$ and $\bar{h}'''$ we get

$$= 2^{-o(h)} \sum_{j \varepsilon \bar{h}} \text{wal}(j; h) \text{wal}(j; s).$$

Case 3 The argument here is exactly the same as in case 2 except that the first two equations contain a difference rather than a sum (change the only "+" to "-"). Notice that

$$\text{wal}(j_1 j_2 \ldots j_{\ell-1} 1; h''') = - \text{wal}(j_1 j_2 \ldots j_{\ell-1}; h')$$

(where h', h''' are defined above). So the difference changes to a sum with the third equation and the same conclusion is drawn.

We wish to find the average payoff for a schema in terms of the Walsh series for the payoff function. Theorem 3.3.7 plus lemma 3.3.8 below will give us the desired formula.

Lemma 3.3.8

$$u(h) = \frac{1}{|h|} \sum_{s} u(s)\, c(h;\, s)$$

where $|h| = 2^{\ell - o(h)}$ = number of points in h.

Proof: By definition

$$u(h) = \frac{1}{|h|} \sum_{s \varepsilon h} u(s).$$

$$= \frac{1}{|h|} \sum_{s} u(s)\, c(h;\, s)$$

since c(h; s) is zero outside of h and 1 for $s \varepsilon h$.

We are now able to show theorem 3.3.9.

Theorem 3.3.9

$$u(h) = \sum_{j \varepsilon \bar{h}} wal(j;\, h)\, w(j)$$

where $\bar{h} = \bar{h}_1 \bar{h}_2 \ldots \bar{h}_\ell$, $\bar{h}_p = \begin{cases} 0, & \text{if } h_p = \# \\ \#, & \text{if } h_p = 0,\ 1 \end{cases}$

and w(j) is the j-th Walsh coefficient of the function u:

$$w(j) = 2^{-\ell} \sum_{s} wal(j; s)\, u(s).$$

Proof: By lemma 3.3.4,

$$u(h) = 2^{o(h)-\ell} \sum_{s} u(s)\, c(h; s).$$

$$= 2^{-\ell} \sum_{s} [u(s) \sum_{j \varepsilon \bar{h}} wal(j; h)\, wal(j; s)]$$

by lemma 3.3.8. A simple rearrangement gives

$$= \sum_{j \varepsilon \bar{h}} [wal(j; h)\, 2^{-\ell} \sum_{s} u(s)\, wal(j; s)].$$

By definition of w(j),

$$= \sum_{j \varepsilon \bar{h}} wal(j; h)\, w(j).$$

So what does this tell us? Remembering that wal(j; h) = ±1, this theorem says that each schema average is the sum of a subset of the Walsh coefficients of the utility function u. See table 3.3.1 for a list of some schema averages in terms of Walsh coefficients. In particular, we note that $\bar{h}$ is a schema with no one bits specified. So whether or not w(j) adds into u(h) depends on where the ones are in j. The following pair of corollaries are a simple consequence of this observation.

$$u(\#\#\#\#) = w(0000)$$

$$u(0\#\#\#) = w(0000) + w(1000)$$
$$u(1\#\#\#) = w(0000) - w(1000)$$
$$u(\#0\#\#) = w(0000) + w(0100)$$
$$u(\#1\#\#) = w(0000) - w(0100)$$
$$u(\#\#0\#) = w(0000) + w(0010)$$

$$u(00\#\#) = w(0000) + w(1000) + w(0100) + w(1100)$$
$$u(10\#\#) = w(0000) - w(1000) + w(0100) - w(1100)$$
$$u(01\#\#) = w(0000) + w(1000) - w(0100) - w(1100)$$
$$u(11\#\#) = w(0000) - w(1000) - w(0100) + w(1100)$$

$$u(0\#\#0) = w(0000) + w(1000) + w(0001) + w(1001)$$
$$u(1\#\#0) = w(0000) - w(1000) + w(0001) - w(1001)$$
$$u(0\#\#1) = w(0000) + w(1000) - w(0001) - w(1001)$$
$$u(1\#\#1) = w(0000) - w(1000) - w(0001) + w(1001)$$

$$u(10\#1) = w(0000) - w(1000) + w(0100) - w(0001)$$
$$- w(1100) + w(1001) - w(0101) + w(1101)$$

$$u(0110) = w(0000) + w(1000) - w(0100) - w(0010)$$
$$+ w(0001) - w(1100) - w(1010) + w(1001)$$
$$+ w(0110) - w(0101) - w(0011) + w(1110)$$
$$- w(1101) - w(1011) + w(0111) + w(1111)$$

TABLE 3.3.1 Some Schema Averages in terms of Walsh Coefficients for $\ell = 4$

Corollary 3.3.10

Let j be a binary string and h be any schema. If the order of j exceeds the order of h, then w(j) does not affect u(h).

Corollary 3.3.11

Let j be a binary string and h a schema. If the definition length of j exceeds the definition length of h, then u(h) does not depend on w(j).

In the next section we will apply theorem 3.3.9 and its corollaries to derive conditions which ensure that a function may be easily optimized by a genetic algorithm. In section 3.5 we use corollaries 3.3.10 and 3.3.11 to design some objective functions for which the genetic algorithm makes a very poor optimizer.

3.4 Characterizing Easy Functions

Corollaries 3.3.10 and 3.3.11 suggest that if the Walsh coefficients of the objective function decrease rapidly with increasing order and length, then the objective function will be easily optimized by a genetic algorithm. In particular, we may show the following.

Theorem 3.4.1

Let s* be the optimum of f, and let w(j) denote the j-th Walsh coefficient of f. If there is some index q,

$1 \leq q \leq \ell$, such that

$$|w(k)| > \sum_{\substack{j \neq k \\ j_q = 1}} |w(j)|,$$

$$\text{where } k = k_1 k_2 \ldots k_\ell,\ k_p = \begin{cases} 0, & \text{if } p \neq q \\ 1, & \text{if } p = q \end{cases}$$

then

$$s_q^* = \begin{cases} 0, & \text{if } w(k) > 0 \\ 1, & \text{otherwise.} \end{cases}$$

Proof: Assume the theorem is false. Then

$$s_q^* = \begin{cases} 1, & \text{if } w(k) > 0 \\ 0, & \text{otherwise.} \end{cases}$$

Let s be the string which is identical to s* except for the q-th bit. We observe that if $j_q = 0$, then wal(s;j) = wal(s*;j). And if $j_q = 1$, then wal(s;j) = -wal(s*;j). So

$$f(s) - f(s^*) = \sum_j [wal(s;j) - wal(s^*;j)]w(j)$$

$$= -2 \sum_{j_q = 1} wal(s^*;j)w(j)$$

$$= 2|w(k)| - 2 \sum_{\substack{j \neq k \\ j_q = 1}} wal(s^*;j)w(j)$$

$$\geq 2|w(k)| - 2 \sum_{\substack{j \neq k \\ j_q = 1}} |w(j)| \geq 0.$$

Thus s* is not the optimum, contradicting our assumptions. Therefore the theorem is true.

Corollary 3.4.2

$$\text{Let } W(q) = \sum_{\substack{j \neq k(q) \\ j_q = 1}} |w(j)|,$$

$$\text{where } k(q) = k_1 k_2 \ldots k_\ell,\ k_p = \begin{cases} 0, & \text{if } p \neq q \\ 1, & \text{if } p = q. \end{cases}$$

If $|w(k(q))| > W(q)$ for each $q = 1, 2, \ldots, \ell$, then

$$s^* = s_1^* s_2^* \ldots s_\ell^*,\ s_p^* = \begin{cases} 0, & \text{if } p\ w(k(p)) > 0 \\ 1, & \text{otherwise} \end{cases}$$

is the optimum.

Moreover, if h and h' are 2 competing schemata with s* being an instance of h, then f(h) > f(h').

Proof: Let s*, h, h' be as specified above. s* is clearly the optimum by theorem 3.4.1. Let $I = \{i_1, i_2, \ldots, i_n\}$ be the set of defining positions for h and h'. Further, let {H(0), H(1), H(2), ..., H(n)} be a partition of the set of binary strings of length ℓ into n + 1 equivalence classes (n = order of h and h') such that any string j in equivalence class H(p) has $j_{i_p} = 1$. That is, all the strings in H(p) have bit i_p equal to 1. H(0) is the set of

all strings which have no 1's in any position p ε I. That is, $H(0) = \{j \mid j_p = 0 \text{ for all } p \varepsilon I\}$. Note that if $j \varepsilon H(0)$, then $wal(h; j) = wal(h'; j)$. So

$$f(h) - f(h') = \sum_{j} [wal(h; j) - wal(h'; j)]\, w(j)$$

$$= \sum_{0 \le p \le n} \sum_{j \varepsilon H(p)} [wal(h; j) - wal(h'; j)]\, w(j)$$

$$= \sum_{1 \le p \le n} 2|w(k(i_p))| + \sum_{\substack{j \varepsilon H(p) \\ j \neq k(i_p)}} [wal(h;j) - wal(h';j)]\, w(j)$$

$$\ge \sum_{1 \le p \le n} 2|w(k(i_p))| - 2 \sum_{\substack{j \varepsilon H(p) \\ j \neq k(i_p)}} |w(j)|$$

$$> 0.$$

Functions which satisfy the hypothesis of corollary 3.4.2 would be very easily optimized by genetic algorithms since the best schema in each set of competing schemata contains the optimum. But the conditions

$$W(q) < |w(k(q))|, \quad 1 \le q \le \ell$$

seem quite strong. What can we conclude from a weaker hypothesis?

By theorem 3.3.9 and its corollaries, we can see that the only Walsh coefficients which are not included in some schema h with definition length less than or equal to m are coefficients w(j) where definition length j is greater than

m. So if all such coefficients w(j) are very small, then the location of the optimum should be predicted by those schemata with definition length no greater than m. The following theorem shows how the averages over the short schemata may be combined to "predict" the average payoff for a longer schema.

Theorem 3.4.3

Let h be any schema. The average payoff over h is "predicted" by the average payoff over each schema h' which is refined by h and by one Walsh coefficient as follows.

$$u(h) = \sum_{h' \subset h} (-1)^{o-o'+1} u(h') + wal(h; j)\, w(j)$$

where $o = o(h)$,

$o' = o(h')$,

$$j = j_1 j_2 \cdots j_\ell, \quad j_p = \begin{cases} 0, & \text{if } h_p = \# \\ 1, & \text{if } h_p = 0, 1 \end{cases}$$

Proof: By theorem 3.3.9,

$$\sum_{h' \subseteq h} (-1)^{o-o'} u(h')$$

$$= \sum_{o'=0}^{o} (-1)^{o-o'} \sum_{\substack{h' \subseteq h \\ o(h')=o'}} \sum_{k \varepsilon \bar{h}'} wal(h';k)\, w(k)$$

where $\bar{h}' = \bar{h}'_1 \bar{h}'_2 \ldots \bar{h}'_\ell$, $\bar{h}'_p = \begin{cases} 0, & \text{if } h'_p = \# \\ \#, & \text{if } h'_p = 0, 1 \end{cases}$

The summations over $h' \subseteq h$ and $k \varepsilon \bar{h}'$ can be combined into a single summation by noting that each $k \varepsilon \bar{h}$ occurs

$$\binom{o - o(k)}{o' - o(k)}$$

times in the set $\{\bar{h}' \mid h' \subseteq h \text{ and } o(h') = o'\}$. (Having fixed the set of defining positions of k, we can ask how many ways can we choose the $o' - o(k)$ remaining defining positions of h'. Since there are $o - o(k)$ positions left to choose from [remember $h' \subseteq h$], we get the binomial coefficient given above.) So we may write

$$\sum_{h' \subseteq h} (-1)^{o-o'} u(h')$$

$$= \sum_{o'=0}^{o} (-1)^{o-o'} \sum_{k \varepsilon \bar{h}} \binom{o - o(k)}{o' - o(k)} \; wal(h; k)\, w(k)$$

(Notice that $wal(h'; k) = wal(h; k)$ for $k \varepsilon \bar{h} \subseteq \bar{h}'$.)

$$\sum_{h' \subseteq h} (-1)^{o-o'} u(h')$$

$$= \sum_{k \varepsilon \bar{h}} \sum_{o'=0}^{o} \binom{o - o(k)}{o' - o(k)} (-1)^{o-o'} \; wal(h; k)\, w(k)$$

$$= \sum_{k \varepsilon \bar{h}} (1 + (-1))^{o-o(k)} \; wal(h; k)\, w(k)$$

$$= wal(h; k)\, w(k), \text{ where } k \varepsilon \bar{h} \text{ with } o(k) = o.$$

$$= wal(h; j)\, w(j)$$

A simple rearrangement gives

$$u(h) = \sum_{h' \subset i} (-1)^{o-o'+1} u(h') + wal(h; j)\, w(j).$$

So if the Walsh coefficients w(j) are very small whenever the definition length of j exceeds m, then the location of the optimum can be determined from the average payoffs for the schemata with definition length less than or equal to m. But we have no assurance that the optimum will lie in the best schema of each set of competing schemata.

As an example, consider the Walsh series and corresponding objective function shown in table 3.4.1. All the Walsh coefficients of length 3 or more are zero. Looking at the schemata of length 1 and 2, we note that each of 0###, ##1#, 00##, #11#, and ##10 is better than its competitors. The optimum is s* = 0001, which is not an instance of #11#, ##1#, nor ##10. In the next section we will see some examples of functions for which the optimum lies in the worst schema of each set of competing schemata with length not exceeding m (m can be made as large as ℓ - 1).

3.5 Constructing Hard Functions

Corollary 3.3.11 suggests a way to construct functions which will fool the genetic algorithm into exploring the wrong regions of the search space. By properly choosing

j	w(j)	f(j)	h	f(h)
0000	8	11	####	8
0001	0	15	0###	10
0010	-1	13	1###	6
0011	-2	9	#0##	8
0100	0	3	#1##	8
0101	0	7	##0#	7
0110	2	13	##1#	9
0111	0	9	###0	8
1000	2	3	###1	8
1001	0	7	00##	12
1010	0	5	01##	8
1011	0	1	10##	4
1100	2	3	11##	8
1101	0	7	#00#	9
1110	0	13	#01#	7
1111	0	9	#10#	5
			#11#	11
			##00	5
			##01	9
			##10	11
			##11	7

Table 3.4.1 An Example of "Misleading" Schemata

the values of the short Walsh coefficients we may make some of the short schemata have poor averages. Then by carefully setting the long coefficients, the optimum can be placed in those poor schemata. It is not immediately obvious how to set the Walsh coefficients to properly place the optimum, but some simple analysis will reveal one way to do so.

We note that theorem 3.3.9 includes, as a special case, the inverse Walsh transform:

$$u(s) = \sum_j \text{wal}(j;\ s)\ w(j)$$

where s is a schema with no boxes -- that is, s is a binary string of length ℓ. Now if we make all $w(j) \geq 0$, then the optimum will be $s^* = 00\ldots0$ since

$$u(00\ldots0) = \sum_j w(j) \geq \sum_j \text{wal}(j;\ s)\ w(j).$$

This inequality must be strict if $w(j) > 0$ for all j. (Actually, we will assume $w(0) = 0$ since $w(0)$ is just the average of u over the whole search space.)

By theorem 3.3.9, we can see that if all the Walsh coefficients are stricly positive, and if h and h' are competing schemata with h containing $s^* = 00\ldots0$, then

$$u(h) - u(h') = \sum_{j \varepsilon \bar{h}} [1 - \text{wal}(j;\ h')]\ w(j)$$

> 0 since some $\text{wal}(j;\ h') = -1$.

So we have the schemata containing the optimum being the best of each set of competing schemata. Thus if $w(j) > 0$, for all $j \neq 0$, we have a very easy function[4].

So we want some negative Walsh coefficients. If we keep $w(j) \geq 0$ for short j, then the short schemata will indicate that zeroes are better than ones. If we make some of the other Walsh coefficients negative we may move the optimum away from s = 00...0 to s = 11...1. Looking at the inverse Walsh transform for u(11...1), we may write:

$$u(11...1) = \sum_{j} wal(j;\ 11...1)\ w(j)$$

$$= \sum_{o(j) \text{ is even}} w(j) - \sum_{o(j) \text{ is odd}} w(j).$$

---

[4]Actually, by choosing some of the coefficients to be near zero, we may have no discernible difference between competing short schemata, yet u(00...0) could still be much larger than u(s), $s \neq 00...0$. This would be a hard function since genetic drift would deterime how the search space was explored.

As an example, consider the function and associated schema averages given in table 3.5.1. Notice that all of the schema averages are close to zero whereas the payoff ranges from 10.96 to -7.98. So the estimated average payoffs will probably not be accurate enough to show which schema is the best of a set of competing schemata. The genetic algorithm may discover the optimum, but is likely to wander through the search space without zeroing in on the best schemata. We would expect poor average performance and a large variation between runs of the genetic algorithm.

The "hard" functions we will construct in this section would cause the genetic algorithm to rapidly and consistently converge to some suboptimal point. So making several runs would not reveal that the function was pathological.

o(j)	w(j)	u(j)
0	0	10.96
1	2^{-8}	-7.98
2	2^{-8}	5.98
3	2^{-8}	-3.99
4	2^{-8}	1.99
5	2^{-8}	-0.0078
6	2^{-8}	-1.99
7	1	3.98
8	2	-5.98

h	u(h)	h	u(h)
########	0	11######	-2^{-8}
0########	2^{-8}	000#####	$7 \cdot 2^{-8}$
1########	-2^{-8}	001#####	-2^{-8}
00######	$3 \cdot 2^{-8}$	011#####	-2^{-8}
01######	-2^{-8}	111#####	-2^{-8}

The value of this function and its Walsh transform depend only on the order of the argument, not the actual arrangement of zeroes and ones in the string. The schema averages are such that for all schemata of the same order, those with the same number of ones have the same average utility.

TABLE 3.5.1 A Function which Causes a Large Variation in the Performance of a Genetic Algorithm

So making w(j) < 0 for j's with even order does not help make s = 11...1 the optimum. But making w(j) < 0 for j's with odd order will help make s = 11...1 the optimum.

Probably the simplest (easist to describe and analyze) class of functions satisfying the criteria suggested above is that class whose Walsh coefficients are zero except that w(j) > 0 if o(j) = 1 and w(j) < 0 if o(j) = k where k is some relatively large odd integer. For example, with 12 bit strings we may consider the function defined by

$$w(j) = \begin{cases} 1, & \text{if } o(j) = 1 \\ -0.05, & \text{if } o(j) = 7 \\ 0, & \text{otherwise.} \end{cases}$$

A little thought should reveal that since the Walsh coefficients depend on only the order of the index, then the function values will depend on only the order of the argument. (Consider the inverse Walsh transforms of s and s' where o(s) = o(s').) Further, the average function value for a schema h depends on the order of h and the number of ones in h, not on the arrangement of the ones and zeroes and boxes. So u(1#0##1#...#1) = u(101#...#) = u(#...#110), etc. Table 3.5.2 gives the function values defined by the Walsh series above. Table 3.5.3 gives some average function values for different schemata.

Looking at table 3.5.3, we see that all of the short schemata (up to definition length 6 at least) are misleading since the true optimum is s = 11...1. Running the

o(j)	w(j)	u(j)
0	0	-27.6
1	1.0	16.6
2	0	10.4
3	0	4.2
4	0	3.6
5	0	3.0
6	0	0
7	-0.05	-3.0
8	0	-3.6
9	0	-4.2
10	0	-10.4
11	0	-16.6
12	0	27.6

TABLE 3.5.2 A "hard" function

h	u(h)	h	u(h)
##########	0	1111########	-4
0##########	1	00000#######	5
1##########	-1	00001#######	3
00#########	2	00011#######	1
01#########	0	00111#######	-1
11#########	-2	01111#######	-3
000########	3	11111#######	-5
001########	1	000000######	6
011########	-1	000001######	4
111########	-3	000011######	2
0000########	4	000111######	0
0001########	2	001111######	-2
0011########	0	011111######	-4
0111########	-2	111111######	-6

TABLE 3.5.3 Selected schema averages for function of table 3.5.2

genetic algorithm on this function several times gives approximately the expected behavior. After a short time[5], the population consists of strings which contain only one or two ones. Eventually the population converges to a point consisting of 11 zeroes and a one. It may take quite a long time to converge since all of the 12 strings which contain just one one have equal utility and crossovers produce only worse strings.

Now we could have easily predicted this behavior by looking at table 3.5.2. Unless the initial population includes the optimum, it is unlikely to be generated. The genetic algorithm rapidly discovers that changing a 1 to a 0 increases the utility of a string and even without crossover the algorithm would proceed toward s = 00...0. Moreover, the optimum is rather isolated -- all the strings which are similar to it have low utility, whereas all but one of the strings which are very different from it have high utility. Does a hard function always have an isolated optimum?

It seems intuitively clear that if all the short schemata containing the optimum are to have poor average payoffs, then most of the points in those schemata must

---

[5]With a population of 50 strings, after 10 - 15 generations (about 350 - 600 function evaluations) none of the strings contain more than 3 ones and only a few contain 3 ones. 10 - 15 generations is a short time when compared to the time required for convergence on the functions of the environment E described in chapter 2.

have low payoffs. Obviously, if the longer schemata containing the optimum (excluding the schema consisting of only the optimum) are also poor schemata, then the optimum will be truly isolated. So it would seem that the degree to which the optimum is isolated depends on how "hard" the function is in the sense that as the length of the shortest schemata which are not misleading increases the optimum is more completely isolated.

As another example of a hard function, consider the function shown in table 3.5.4 with schema averages as shown in table 3.5.5. Here the optimum is not quite so isolated, since the strings which differ by only one bit have very good payoffs. The genetic algorithm again moves toward s = 00...0 and usually converges rapidly to a string consisting of 13 zeroes and 3 ones. In one run, the algorithm converged to a string consisting of 11 ones and 5 zeroes. Apparently, when the function values change rapidly as small changes are made in the strings, the genetic algorithm does not estimate the schema averages very well. Thus, there may be a large variation in running time, and even in the ultimate value achieved from one run to the next.

3.6 Summary

In this chapter we have explored the relations between schemata and the Walsh functions. Based on these relations

o(s)	w(s)	u(s)
0	0	-15792
1	1	-3054
2	0	220
3	0	374
4	0	40
5	-1	-70
6	0	-12
7	-1	30
8	0	0
9	0	-30
10	0	12
11	0	70
12	0	-40
13	0	-374
14	0	-220
15	0	3054
16	0	15792

TABLE 3.5.4 Another hard function

h	u(h)	h	u(h)
################	0	0000###########	4
0###############	1	0001###########	2
1###############	-1	0011###########	0
00##############	2	0111###########	-2
01##############	0	1111###########	-4
11##############	-2	00000##########	4
000#############	3	00001##########	4
001#############	1	00011##########	0
011#############	-1	00111##########	0
111#############	-3	01111##########	-4
		11111##########	-4

TABLE 3.5.5 Schema Averages for Function of Table 3.5.4

and on an analysis of how genetic algorithms explore the search space, we have shown how to generate objective functions which genetic algorithms fail to optimize. Such functions are specified by giving their Walsh transforms.

Here we have considered the objective function to be simply a real-valued function of a binary string. It seems intuitively clear, although we have not proved it, that the "hard" functions are not smooth with respect to small changes in their arguments. That is, in terms of Hamming distance, they have large "slopes", expecially near the optimum. Moreover, the optimum seems to be surrounded by very bad points and represents an isolated, local maximum.

CHAPTER 4

SOME EFFECTS OF REPRESENTATION

4.1 Introduction

In the last chapter we assumed that the search space consisted of binary strings with no special significance. Here we assume that the binary strings are encodings for real values. In sections 4.2, 4.3, and 4.4 we assume the objective function is a function of one real variable which may be encoded in three different ways: fixed-point, Gray-coded fixed-point, and floating-point. We try to determine natural properties (such as continuity or bounded slope) which will ensure that the objective function can be optimized by a genetic algorithm.

Section 4.5 deals with functions of several real variables. It is more difficult to find natural properties to ensure that these objective functions can be optimized by a genetic algorithm.

4.2 Fixed-Point Representation

There are many ways of representing real values as binary strings. Perhaps the most natural encoding is as

"fixed-point" binary strings. With fixed-point representation each string may be considered to be a binary number with the binary point in a fixed position. We will assume that the real variable takes values between 0 and 1, so that the binary point is to the left of the first bit in the string. If $\hat{s}$ is the real value represented by the string $s = s_1 s_2 \ldots s_\ell$, then

$$\hat{s} = \sum_{p=1}^{\ell} s_p 2^{-p}.$$

So 00...0 represents zero and 11...1 represents $1 - 2^{-\ell}$. We note that we can only represent real values of the form $k \cdot 2^{-\ell}$, $0 \leq k < 2^{\ell}$. So we have a limited precision (of course). The encoding scheme used by De Jong is a slight generalization of this involving a linear transformation applied to $\hat{s}$ as defined above so that the value represented by s may span any real range [a, b] with precision (De Jong called it "tolerance") δ. We note at this point that by using a precision of δ, we are implicitly assuming that the objective function makes no large changes in value when the argument changes by less than δ. Otherwise, we may find that the true optimum is not close to the best representable point.

Most of the literature on discrete Walsh transforms is based on this "fixed-point" binary representation. The function being transformed is assumed to be restricted to

a domain of [0, 1), or, equivalently, the function may be periodic with period 1. The reason that this representation is chosen is that it provides an almost one-to-one correspondence between the unit interval and the dyadic group (the infinite cross product of the 2-element cyclic group with itself). Some early work on the (continuous) Walsh transform by Fine [7] established the usefulness of viewing the Walsh functions as functions over the dyadic group.

In a more recent paper, Yuen [22] gives the following bound on the magnitude of the k-th Walsh coefficient when the function to be transformed has a continuous r-th derivative:

$$|w(k)| \leq 2^{-n(k)} \, ||f^{(r)}||$$

where r = order of k

$$n(k) = r + \sum_{p=1}^{\ell} p \, k_p$$

$$\text{and } ||f^{(r)}|| = \max_{0 \leq t < 1} |f^{(r)}(t)|.$$

This result is actually for a continuous function and a continuous (integral rather than summation) transform. The corresponding result for a discrete transform is

$$|w(k)| \leq 2^{-n(k)} ||\Delta^r f||$$

where r and n are as before and

$$||\Delta^r f|| = \max_{s} |\Delta^r f(\hat{s})|$$

where $\Delta^r f(\hat{s})$ is a normalized r-th difference

$$\Delta f(\hat{s}) = \frac{f(\hat{s} + 2^{-\ell}) - f(\hat{s})}{2^{-\ell}}$$

$$\Delta^{i+1} f(\hat{s}) = \frac{\Delta^i f(\hat{s} + 2^{-\ell}) - \Delta^i f(\hat{s})}{2^{-\ell}}.$$

We will not make the distinction between $f^{(r)}$ and $\Delta^r f$ because $\Delta^r f$ will normally be a good approximation to $f^{(r)}$.

From Yuen's result we can see that if the objective function is analytic with decreasing or constant values for $||f^{(r)}||$, then the Walsh coefficients w(j) where $\ell(j)$ is large must be negligible. To make this more exact, we give the following definition and theorem.

Definition 4.2.1

We will denote the sum of the magnitudes of all the Walsh coefficients whose indices have definition length m or greater by $\sigma(m)$. That is,

$$\sigma(m) = \sum_{\ell(j) \geq m} |w(j)|. \quad (0 \leq m < \ell)$$

Theorem 4.2.2

If there exists a real constant C such that

$$||f^{(r)}|| \leq 2^{\frac{r(r-1)}{2}} C, \quad \text{for } 2 \leq r \leq \ell,$$

then $$\sigma(m) = \sum_{\ell(j) \geq m} |w(j)| < (2/3)\,(3/4)^m\, C, \quad \text{for } m \geq 1.$$

Proof: By Yuen's result,

$$\sigma(m) \leq \sum_{\ell(j)\geq m} 2^{-n(j)} \, ||f^{(o(j))}||.$$

Let

$B(k,r,q) = \{j \mid \ell(j) = k,\ o(j) = r,\ \min\{p \mid j_p = 1\} = q\ \}.$

We note that the disjoint union of the $B(k,r,q)$'s for which $m \leq k \leq \ell - 1$, $2 \leq r \leq k + 1$, and $1 \leq q \leq \ell - k$ is just $\{j \mid \ell(j) \geq m\}$. Further, the number of strings in $B(k,r,q)$ is just

$$|B(k,r,q)| = \binom{k-1}{r-2}$$

Finally, let $n'(k,r,q) = \min\{n(j) \mid j \in B(k,r,q)\}$.

It is easy to see that

$$n'(k,r,q) = n(j)$$

where $j = j_1 j_2 \ldots j_\ell$, with

$$j_p = \begin{cases} 1, \text{ if } q \leq p \leq q + r - 2, \text{ or } p = q + k \\ 0, \text{ otherwise.} \end{cases}$$

So

$$n'(k,r,q) = r + q + k + \sum_{p=q}^{q+r-2} p$$

$$= k + rq + \frac{r(r-1)}{2} + 1.$$

This gives us

$$\sigma(m) < \sum_{k=m}^{\ell-1} \sum_{r=2}^{k+1} \sum_{q=1}^{\ell-k} \binom{k-1}{r-2} \, 2^{-k\,-rq\,-\,\frac{r(r-1)}{2}\,-\,1} \, ||f^{(r)}||$$

Rearranging:

$$\sigma(m) < \frac{1}{2} \sum_{k=m}^{\ell-1} 2^{-k} \sum_{r=2}^{k+1} \binom{k-1}{r-2} 2^{-\frac{r(r-1)}{2}} ||f^{(r)}|| \sum_{q=1}^{\ell-1} 2^{-rq}$$

Since

$$\sum_{q=1}^{\ell-k} 2^{-rq} < 2^{1-r}$$

and

$$||f^{(r)}|| \leq 2^{\frac{r(r-1)}{2}} C,$$

we may write

$$\sigma(m) < C \sum_{k=m}^{\ell-1} 2^{-k} \sum_{r=2}^{k+1} \binom{k-1}{r-2} 2^{-r}$$

Applying the binomial theorem

$$\sigma(m) < C \sum_{k=m}^{\ell-1} 2^{-k-2} (3/2)^{k-1} = \frac{C}{6} \sum_{k=m}^{\ell-1} (3/4)^{k}.$$

So $\sigma(m) < (2/3)\ (3/4)^{m}\ C.$

The following lemma is useful in proving two corollaries to theorem 4.2.2.

Lemma 4.2.3

Let $||f||$ and $||w||$ denote the largest magnitude of a function f and its Walsh transform:

$$||f|| = \max_{s} |f(s)|$$

$$||w|| = \max_{j} |w(j)|.$$

Then $||w|| \leq ||f|| \leq 2^{\ell} ||w||$.

Proof: By definition

$$w(j) = 2^{-\ell} \sum_{s} f(s) \, wal(j; s).$$

So

$$|w(j)| \leq 2^{-\ell} \sum_{s} ||f|| = ||f||.$$

Since this holds for all j, we have

$$||w|| \leq ||f||.$$

Similarly,

$$f(s) = \sum_{j} w(j) \, wal(j; s).$$

So

$$|f(s)| \leq \sum_{j} ||w|| = 2^{\ell} ||w||.$$

This holds for all s, therefore

$$||f|| \leq 2^{\ell} ||w||.$$

We now present two simple corollaries to theorem 4.2.2.

Corollary 4.2.4

If $||f^{(r)}|| \leq 2^{\frac{r(r-1)}{2} - \ell} ||f||$, for $2 \leq r \leq \ell$, then $\sigma(m) < (2/3) (3/4)^{m} ||w||$, $1 \leq m < \ell$.

Proof: Let $C = 2^{-\ell} ||f||$ in theorem 4.2.2:

$$\sigma(m) \leq (2/3)\ (3/4)^m\ 2^{-\ell}\ ||f||.$$

By the previous lemma,

$$\sigma(m) \leq (2/3)\ (3/4)^m\ ||w||.$$

Corollary 4.2.5

If

$$||f^{(r)}|| \leq 2^{\frac{r(r-1)}{2}+1} |w(k(q))|,\ 2 \leq r \leq \ell,\ 1 \leq q \leq \ell,$$

then f is an easy function as in corollary 3.4.2. That is, if h and h' are two competing schemata with the optimum being an instance of h, then f(h) > f(h').

Proof: By theorem 4.2.2 with C = 2 |w(k(q))|,

$$\sigma(1) < |w(k(q))|,\ 1 \leq q \leq \ell.$$

Now, $\sigma(1) = \sum_{\ell(j)\geq 1} |w(j)| > \sum_{\substack{j \neq k(q) \\ j_q = 1}} |w(j)| = W(q).$

So W(q) < |w(k(q))|. This is the premise of corollary 3.4.2, so the desired result follows by that corollary.

Corollary 4.2.4 gives conditions which ensure that σ(m) is small compared to ||w||, especially for large m. Unfortunately, as we saw in the last chapter, making σ(m) small is not sufficient to make f an "easy" function.

Corollary 4.2.5 gives sufficient conditions for f to be an easy function. These conditions essentially limit the rate at which the derivatives of f can grow. Notice that there is no explicit restriction for $f^{(1)}$. But there is an implicit restriction. By Yuen's result

$$|w(k(q))| \leq 2^{-l-q} \, ||f^{(1)}||.$$

So we need

$$||f^{(r)}|| \leq 2^{\frac{r(r-1)}{2} + 1} \, |w(k(q))| \leq 2^{\frac{r(r-1)}{2} - q} \, ||f^{(1)}||.$$

Since this must hold for all $q = 1, 2, \ldots, \ell$, we need

$$||f^{(r)}|| \leq 2^{\frac{r(r-1)}{2} - \ell} \, ||f^{(1)}||, \text{ or}$$

$$||f^{(1)}|| \geq 2^{\ell - \frac{r(r-1)}{2}} \, ||f^{(r)}||, \; 2 \leq r \leq \ell.$$

So the first derivative must "dominate" the other derivatives. The amount of domination required depends on ℓ.

It may seem strange that ℓ should enter into this relation. After all, once ℓ is large enough so that the discrete version of f is a good approximation to the continuous f, we don't change things much by making ℓ larger. But, in fact, as ℓ increases so does the size of the search space. And so does the number of ways that the genetic algorithm can be trapped on a false peak. Since this analysis is based on finding the best representable point, s*, it will not be acceptable to converge to some point s which differs from s* only in the low-order bits. So the 2^{ℓ} factor must appear.

This concludes our discussion of the fixed-point representation scheme.

4.3 Gray-Coded Fixed-Point Representation

A Gray code is a representation with the property that adjacent values differ by only one bit. For example, one 3-bit Gray code is:

$\hat{s}$	Gray code for $\hat{s}$	Normal fixed-point for $\hat{s}$
0	000	000
1/8	001	001
1/4	011	010
3/8	010	011
1/2	110	100
5/8	111	101
3/4	101	110
7/8	100	111

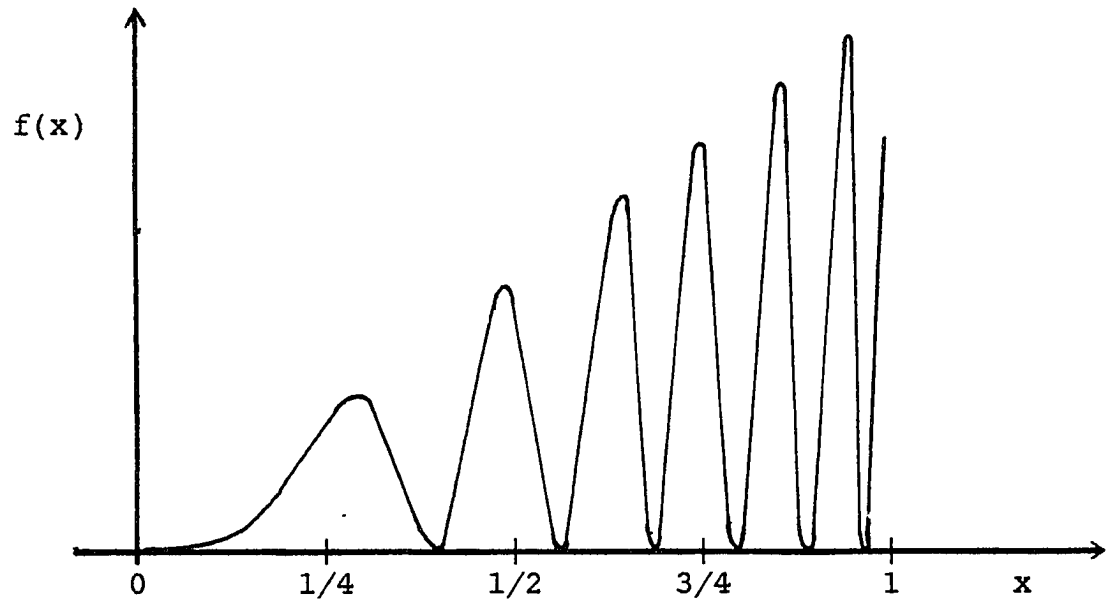
(This is not the only 3-bit Gray code.)

If we use such a Gray code with our genetic algorithm, we can take advantage of the fact that two points which are close together in the Euclidean sense have representations which differ in only a few bits. To show why this might be an advantage, we relate the following experiment.

The genetic algorithm was used to optimize the function

$$f(x) = x \sin^2(20\, x^2),\ 0 \leq x < 1.$$

(See figure 4.3.1.) First, several runs were made using the normal fixed-point representation with 32-bit strings. The genetic algorithm repeatedly converged to a point near the



$$f(x) = x \sin^2(20\, x^2), \qquad 0 \leq x < 1$$

FIGURE 4.3.1 A function Which Is Easier to Optimize Using a Gray-Coded Representation

optimum, but not to the optimum. In each case, the final string s differed from the optimal string by several bits. Moreover, in order to generate a better string, several bits of s had to be changed simultaneously. In particular, the optimum is

$$s^* = 11110000...0 \quad <----> \quad \hat{s}^* = 0.93750$$

but the final string in some cases was

$$s = 111011100...0 \quad <----> \quad \hat{s} = 0.92988.$$

Changing any single bit or pair of bits makes s worse; we must change at least three bits simultaneously to improve s.

Next, several runs were made using a Gray-coded fixed-point 32-bit representation. In every case the true optimum was found. It was no longer necessary to change several bits at once to improve a string. In fact, for the strings representing points in the neighborhood of the optimum, a better string could be generated by changing just one bit.

So we have reason to hope that using the Gray code representation will make the class of "easy" functions easier to describe. Unfortunately, we have no result comparable to the one given in the last section relating the Walsh coefficients to the derivatives of the objective function for this representation. All we can say is that, based on experimental results, the use of a Gray code does seem to improve the performance on some functions and does

not significantly degrade the performance on any function tried so far[1].

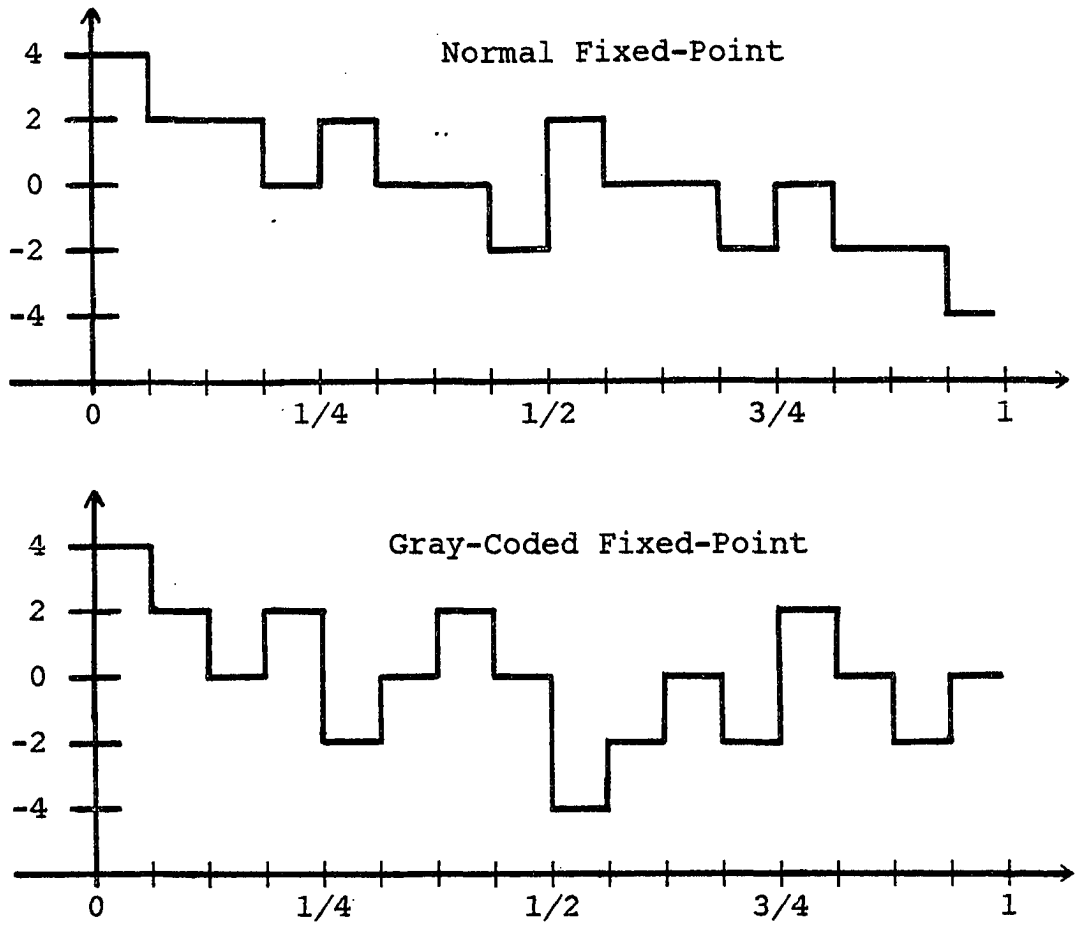
4.4 Floating-Point Representation

In section 4.2 we described the fixed-point representation in which the binary point had a fixed position within the string. "Floating-point" means that the position of the binary point may vary from one string to another. Each string must therefore include a specification of where the binary point should be located. So a portion of the string, say the last ℓ_e bits, is used to indicate the position of the binary point in the first ℓ_m ($\ell_e + \ell_m = \ell$) bits of the string. This is equivalent to saying that the string is made up of the exponent and mantissa of the real value it represents when that real value is expressed in scientific notation. That is, if $\hat{s}$ is the real value represented by the string $s = s_1 s_2 \ldots s_\ell$, then

$$\hat{s} = \hat{s}_m \, 2^{\hat{s}_e - c_e}$$

---

[1]Actually, we have constructed a function for which the use of a Gray code significantly degrades performance. This is the function whose Walsh transform has the first-order coefficients equal to one and all other coefficients are zero. (That only works for finite ℓ of course.) When interpreted as a function over a continuous interval [0, 1), this function has the appearance of being a staircase with smaller staircases on each step, etc. (See figure 4.3.2.) This is easy for the genetic algorithm with the normal representation since the schemata break the space up in the same patterns. But the Gray code representation makes this worse since it effectively suffles the steps around and breaks up the simple pattern within a pattern.



This is the function whose Walsh transform is:

$$w(j) = \begin{cases} 1, \text{ if } o(j) = 1 \\ 0, \text{ otherwise} \end{cases} \qquad (\text{for } \ell = 4)$$

Notice that the first-order schemata correctly predict the location of the optimum (for normal fixed-point encoding):

u(#...#0#...#) = 1

u(#...#1#...#) = -1

So zeroes are preferred in every position.

FIGURE 4.3.2 A Function Which Becomes Harder to Optimize with a Gray-coded Representation

$$\hat{s}_e = \sum_{p=\ell_m+1}^{\ell} s_p \, 2^{\ell-p}$$

and

$$\hat{s}_m = \sum_{p=1}^{\ell_m} s_p \, 2^{-p}.$$

$\hat{s}_e$ is actually the characteristic of $\hat{s}$ and differs from the exponent of $\hat{s}$ by the constant c_e.

c_e is generally chosen to give a balanced range of exponents -- the magnitudes of the most positive and most negative exponents should not differ by more than one. So c_e is generally 2^{ℓ_e-1}. And the range of exponents is usually $-c_e$ to $c_e - 1$ inclusive. We would like to assume that $0 \leq \hat{s} < 1$, so we will let $c_e = 2^{\ell_e} - 1$, and the range of exponents is $-c_e = 1 - 2^{\ell_e}$ to 0.

The use of a floating-point representation gives us a wider range of values than a fixed-point representation of the same length. With our choice for c_e, the smallest non-zero value we can represent is

$$\hat{s} = \left(2^{-\ell_m}\right) \quad 2^{1 - 2^{\ell_e}} = 2^{1 - \ell_m - 2^{\ell_e}}$$

And the largest value we can represent is

$$\hat{s} = \left(1 - 2^{-\ell_m}\right) \quad 2^{0} = 1 - 2^{-\ell_m}$$

Near $\hat{s} = 1$, we can represent points differing by $2^{-\ell_m}$.

Near $\hat{s} = 0$, we can represent points differing by

$2^{1 - \ell_m - 2^{\ell_e}}$. So we have less precision near $\hat{s} = 1$, but more precision near $\hat{s} = 0$.

Let s be a string whose mantissa starts with a zero bit. The value $\hat{s}$ represented by s is unchanged if the mantissa of s is shifted one bit left and the exponent (or characteristic) is decreased by one. So some values have more than one floating-point representation. (With fixed-point representation, any representable value has a unique representation.) We will say a string s is <u>normalized</u> if the mantissa s_m starts with a one bit. Thus there is only one normalized floating-point encoding of each representable value[2]. Most computer hardware (or software on machines lacking floating-point hardware) automatically normalizes the results of arithmetic operations on floating-point values.

So what happens if we use a floating-point representation with our genetic algorithms? First of all, it makes sense to do so. When we combine two normalized floating-point strings using crossover we get two normalized

---

[2]Now, in fact, there are strings which cannot be normalized because the exponent is already minimal even though the mantissa starts with a zero. We will assume the corresponding values are not representable. The smallest representable non-zero value then becomes

$$\left(2^{-1}\right) \; 2^{1 - 2^{\ell_e}} = 2^{-2^{\ell_e}}.$$

Also, zero cannot be properly "normalized". So we let the string which is all zeroes represent zero (both the mantissa and the characteristic are all zeroes).

floating-point strings as the result[3]. And mutating a normalized floating-point string will generally give a normalized result. If it doesn't, we can normalize the result.

How then does the use of floating-point representation affect the class of functions which are easily optimized by a genetic algorithm? To answer this, we notice that there are two "limiting cases" for floating-point representations. These are:

1. No characteristic. The string is just the mantissa, $\ell_e = 0$, $\ell_m = \ell$. This is effectively just the fixed-point encoding.
2. No mantissa. The string is just the characteristic or exponent, $\ell_e = \ell$, $\ell_m = 0$. In this case,

$$\hat{s} = 2^{\hat{s}_e - c_e}$$

or $\hat{s}_e = \log \hat{s} + c_e$ (all logarithms are $\log_2$)

or $$\sum_{p=1}^{\ell} s_p \, 2^{\ell - p} = \log \hat{s} + c_e.$$

So s is essentially the fixed-point representation of $\log \hat{s}$.

In between these two extremes, when $0 < \ell_e < \ell$ and $0 <$

---

[3]Actually, if one of the strings is zero, we may not get properly normalized results. In that rare case, we may simply normalize the results, possibly changing some unnormalizable string to zero.

$\ell_m < \ell$, we find there are 2^{ℓ_m} representable points in each interval $[2^{c-1}, 2^c)$, $c = -c_e, -c_e + 1, \ldots, -1, 0$. That is, there are 2^{ℓ_m} representable points in the range $[1/2, 1)$, and also in $[1/4, 1/2)$, and $[1/8, 1/4)$, etc. Within each interval the representable values are evenly spaced. So we have a mixture of the fixed-point and logarithmic properties[4].

If f and f°log (° indicates function composition) are both easy functions for genetic algorithms using a fixed-point encoding, then we would expect f to be an easy function when a floating-point encoding is used. So the theorems and corollaries of section 4.2 can be applied to both f and f°log to determine whether or not f is an easy function.

We would expect that some functions would be easier to optimize using a floating-point representation than with a fixed-point representation. The condition that both f and f°log be easy should really be replaced by a condition that f°float be easy where "float" is the partial function defined by:

---

[4]Since

$$\hat{s} = \hat{s}_m \, 2^{\hat{s}_e - c_e} \quad \text{and} \quad 1/2 \leq \hat{s}_m < 1$$

for a normalized floating-point string s, we have

$$\hat{s}_e = \lfloor \log \hat{s} \rfloor + 1 + c_e.$$

(Here $\lfloor x \rfloor$ = largest integer not exceeding x.) So s_e is essentially the fixed-point representation of log $\hat{s}$. This holds for $0 < \ell_e < \ell$.

y = float(x) if and only if there exists a binary string s of length ℓ such that s is the fixed-point representation of y and the floating-point representation of x.

So $y = \text{float}(x)$ iff $y = \hat{s}_m + \hat{s}_e \, 2^{-\ell_e}$

where

$$\hat{s}_e = \lfloor \log x \rfloor + 1 + c_e$$

$$\hat{s}_m = \frac{x}{2^{\lfloor \log x \rfloor + 1}}$$

But the function float is very awkward to work with.

4.5 Functions of Several Real Variables

Because the genetic algorithm operates on binary strings, we must encode several real values into a single string. In the preceding sections we have seen three ways of encoding one real value as a binary string. A simple way of representing a point x = (x(1), x(2), ..., x(n)) in n-dimensional Euclidean space is to concatenate the strings representing each of the real values x(i). That is, s = s(1)s(2)...s(n) represents $\hat{s} = (\hat{s}(1), \hat{s}(2), \ldots, \hat{s}(n))$ where s(i) represents $\hat{s}(i)$.

De Jong [6] used this type of representation with each s(i) being the fixed-point encoding for $\hat{s}(i)$. Bosworth, Foo, and Zeigler [4] also used this type of representation

but each s(i) was the floating-point encoding for $\hat{s}(i)$.

To begin, let's concentrate on the case where n = 2 and a fixed-point representation is used for each coordinate value. In this case we have an objective function f : R x R ---> R. And the ordered pair (x, y) ε R x R can be represented by s = s(x)s(y) where s(x) is the fixed-point encoding of x and similarly for s(y) and y. We assume for simplicity that s(x) and s(y) are each of length ℓ. Let z be the real value whose fixed-point representation of length 2ℓ is s. Then the following relations hold:

$$x = \sum_{p=1}^{\ell} s(x)_p \, 2^{-p} = \sum_{p=1}^{\ell} s_p \, 2^{-p}$$

$$y = \sum_{p=1}^{\ell} s(y)_p \, 2^{-p} = \sum_{p=\ell+1}^{2\ell} s_p \, 2^{\ell-p}$$

$$z = \sum_{p=1}^{2\ell} s_p \, 2^{-p} = \sum_{p=1}^{\ell} s_p \, 2^{-p} + \sum_{p=\ell+1}^{2\ell} s_p \, 2^{-p}$$

$$z = x + 2^{-\ell} \, y$$

$$x = 2^{-\ell} \, [2^{\ell} \, z]$$

$$y = 2^{\ell} \, z - [2^{\ell} \, z]$$

$$0 \leq z - x < 2^{-\ell} \quad \text{(remember } 0 \leq y < 1\text{)}.$$

Therefore, f : R x R ---> R is an "easy" function if and only if g : R ---> R is "easy" where

$$g(z) = f(2^{-\ell} \, [2^{\ell} \, z], \; 2^{\ell} \, z - [2^{\ell} \, z]), \quad 0 \leq z < 1.$$

In order to determine whether or not g is easy, we need to calculate its derivatives:

$$\frac{dg}{dz}(z) = \frac{df}{dz}(x, y)$$

where $x = 2^{-\ell}\,[2^{\ell} z]$ and $y = 2^{\ell} z - [2^{\ell} z]$.

$$\frac{dg}{dz}(z) = \frac{\partial f}{\partial x}(x, y)\,\frac{dx}{dz} + \frac{\partial f}{\partial y}(x, y)\,\frac{dy}{dz}$$

Here we have a problem: $x = 2^{-\ell}\,[2^{\ell} z]$, $y = 2^{\ell} z - [2^{\ell} z]$ are not differentiable functions. But we are really interested in differences anyway, and so we may use

$$\frac{\Delta g}{\Delta z}(z) = \frac{\Delta f}{\Delta x}(x, y)\,\frac{\Delta x}{\Delta z} + \frac{\Delta f}{\Delta y}(x, y)\,\frac{\Delta y}{\Delta z}$$

Now $\frac{\Delta x}{\Delta z}$ is approximately 1 for $\Delta z > 2^{-\ell}$ (see figure 4.5.1). $\frac{\Delta y}{\Delta z}$ is a bit more complex (again see figure 4.5.1), but generally $\frac{\Delta y}{\Delta z} = 2^{\ell}$. So we have

$$\frac{\Delta g}{\Delta z}(z) \cong \frac{\Delta f}{\Delta x}(x, y) + 2^{\ell}\,\frac{\Delta f}{\Delta y}(x, y).$$

Or $$\frac{dg}{dz}(z) \cong \frac{\partial f}{\partial x}(x, y) + 2^{\ell}\,\frac{\partial f}{\partial y}(x, y).$$

By a similar process,

$$\frac{d^2 g}{dz^2}(z) \cong \frac{\partial^2 f}{\partial x^2}(x, y) + (2^{\ell} + 1)\frac{\partial^2 f}{\partial x \partial y}(x, y) + 2^{\ell}\,\frac{\partial^2 f}{\partial y^2}(x, y)$$

And

$$\frac{d^3 g}{dz^3}(z) \cong \frac{\partial^3 f}{\partial x^3}(x, y) + (2^{\ell} + 2)\,\frac{\partial^3 f}{\partial x^2 \partial y}(x, y) + (2^{\ell+1} + 1)\,\frac{\partial^3 f}{\partial x \partial y^2}(x, y) + 2^{\ell}\,\frac{\partial^3 f}{\partial y^3}(x, y)$$

Etcetera.

By the theorems and corollaries of section 4.2, we see that the restrictions on $||g^{(r)}||$ which suffice to make g "easy" would more severely restrict the partial derivatives

x, y, z all take values in the range [0, 1).

$x = 2^{-\ell}\ [2^{\ell}\ z]$ $\quad y = 2^{\ell}\ z - [2^{\ell}\ z]$ $\quad z = x + 2^{-\ell}\ y$

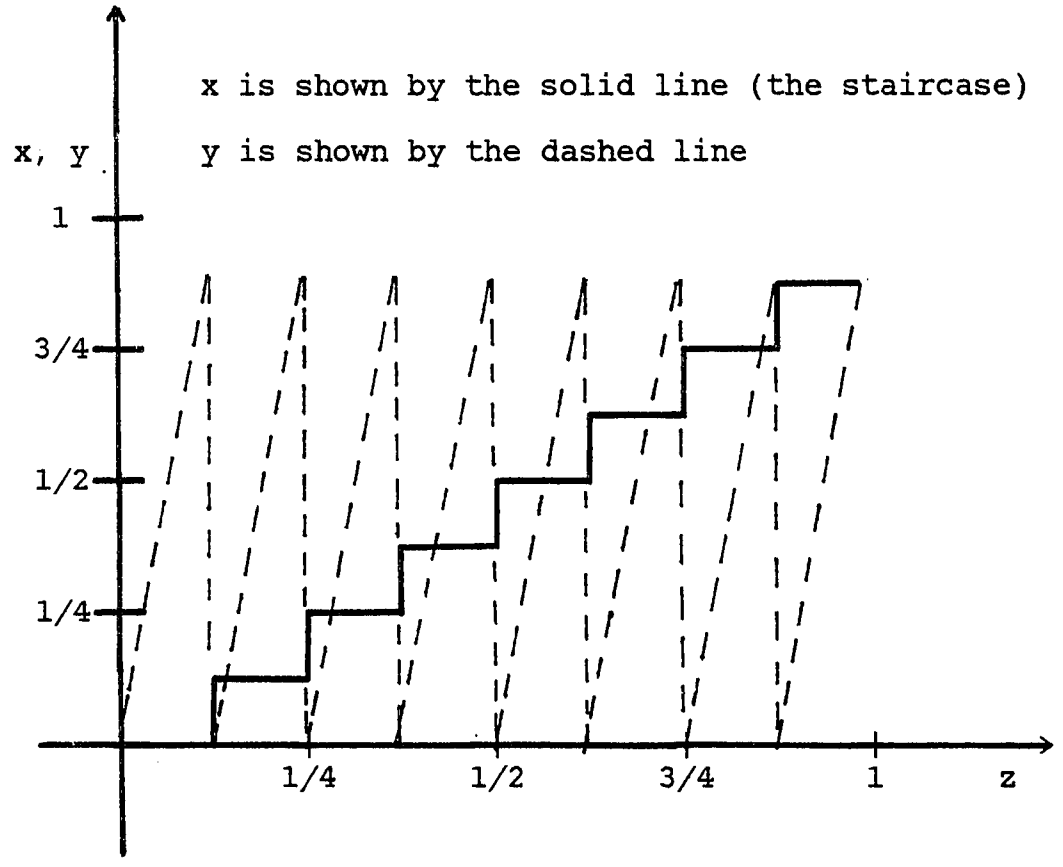


FIGURE 4.5.1 Encoding an Ordered Pair into a Single Value ($\ell = 3$)

of f involving y than those involving only x. But this is just an artifact of the way we derived those results! Theorem 4.2.2 and its corollaries are based on Yuen's relation between high-order derivatives and high-order Walsh coefficients. What we really need is a relation between the utilities of long-definition schemata and high-order derivatives. It is okay for some "high-order" Walsh coefficients to be large -- namely those with small rank.

In fact, intuition and empirical studies both suggest that two functions such as f and f' related by f'(x, y) = f(y, x) are (very nearly) equally difficult to optimize with a genetic optimizer.

So we must conclude that the results of section 4.2 are of little use in determining how easy or difficult it will be to optimize a function of several real variables.

We close this section with the following suggestion for another encoding of n-tuples into strings. Let $s(i) = s(i)_1 s(i)_2 \ldots s(i)_\ell$ be the fixed-point representation of $\hat{s}(i)$. Rather than concatenating the strings s(1), s(2), ..., s(n), why not interleave them so that $s = s(1)_1 s(2)_1 \ldots s(n)_1 s(1)_2 s(2)_2 \ldots s(n)_2 \ldots s(1)_\ell \ldots s(n)_\ell$? This way we may find that a good "region" of the search space is represented by a schema with all its defining positions in the first ℓ (or perhaps 2ℓ or 3ℓ) positions of the strings. This would be a (relatively) short-definition schema (especially if $n \gg 1$). In contrast,

when using the "concatenation" representation, a schema which specifies the values for the most significant bit in each $\tilde{s}(i)$ would have a definition length of $(n - 1)\,\ell$ which is nearly as long as the strings themselves. It may be worth comparing the performance of a genetic optimizer on a set of functions using first the "concatenation" and then the "interleaved" representations.

4.6 Summary

In this chapter we have described three encodings for real values as binary strings: fixed-point, Gray-coded fixed-point, and floating-point. We developed sufficient conditions which ensure that a function of one real variable can be optimized by a genetic algorithm when its domain is encoded using the fixed-point representation. These conditions were based on the relative magnitudes of the various higher-order derivatives of the objective function. We were unable to derive such conditions for the other two representations or for objective functions of several variables. (We were able to relate the conditions needed when the floating-point representation is used to those for the fixed-point representation.) Finally, we suggested a new representation based on interleaving rather than concatenation for encoding several real values into a single binary string.

CHAPTER 5

CONCLUSIONS AND DIRECTIONS FOR FURTHER RESEARCH

We have described a class of function optimizers known as genetic algorithms which have been shown empirically to be robust, general-purpose optimizers. Previous research [4], [5], [6], [23] has shown that genetic algorithms are especially appropriate when the objective function is multimodal, high-dimensional, and/or contaminated by noise. Genetic algorithms may also be used to optimize unimodal, low-dimensional, noiseless functions, but they are inferior to more traditional hill-climbing methods.

We have suggested two improved robustness measures for use in future empirical studies. The first of these (see section 2.3) can be easily computed from tables given by De Jong [6] without making new computer runs. The result is that the genetic algorithm is seen to be superior (for this particular set of 5 objective functions) to two other, more standard optimizers in both "on-line" and "off-line" performance. (See tables 2.3.1 - 2.3.4.) The second robustness measure requires new computer runs, and we have no results based on it. It would be interesting to see how the various optimizers are ranked according to this robust-

ness measure. This is an area for future research.

Most of our research has been directed toward answering the question "What class of functions can be optimized by genetic algorithms?" Since genetic algorithms require that points in the search space be represented as strings, and since binary strings have been used in most previous work (and also since binary strings are the easiest for this research), we started by investigating functions of binary strings. Using the Walsh transform, and based on the theoretical work by Holland [10] relating the behavior of genetic algorithms to the average objective function values over the various schemata, we have developed some conditions which are sufficient to ensure that a genetic algorithm will consistently optimize an objective function. Unfortunately, these were simply sufficient conditions and we have no necessary conditions. We were able, however, to design some functions which could not be optimized by the genetic algorithm. These functions could not be optimized by any methods we are aware of except, of course, for exhaustive search. Here is an obvious place for further research -- can any necessary conditions be found? Can the sufficient conditions be weakened? (See the discussion near the end of section 3.4.)

We really want to work with Euclidean search spaces -- objective functions whose arguments are real values, not binary strings. So we would like to use properties of

real-valued functions of real variables in describing the class of "genetically optimizable" functions. In chapter 4 we attempt to derive such conditions for a variety of representations. For functions of a single real variable, and using a fixed-point representation, we are able to give sufficient conditions on the relative magnitudes of the function and its derivatives to ensure it is "genetically optimizable." As this result is based on results from chapter 3, we have no necessary conditions. Here is another area for further research effort. If we could prove a stronger version of theorem 3.4.1 (and corollary 3.4.2) which was based on having the Walsh coefficients of length $\leq m$ be large with respect to $\sigma(m+1)$, then we could also prove a stronger version of theorem 4.2.2 (and corollaries 4.2.4 and 4.2.5) which stated that:

> If $||f^{(r)}|| \leq c(r)\ ||f||$, for $r \geq 2$, then f is an "easy" function.

Notice how this differs from corollaries 4.2.4 and 4.2.5. We would not need the values of $|w(k(q))|$ to determine whether f was easy or not. This would be very attractive since computing the w(k(q))'s is quite difficult for most realistic objective functions.[1]

In the remaining sections of chapter 4, we examined Gray-coded fixed-point, floating-point, and the "concatena-

---

[1]Actually, computing $||f^{(r)}||$ and $||f||$ may be quite difficult. But it would be an improvement to save the calculation of the w(k(q))'s.

tion" representations. Some empirical results using the Gray-coded fixed-point representation suggest that it may improve the end-of-run performance of genetic algorithms because it maps Euclidean neighborhoods into Hamming neighborhoods. It would be interesting to see the effect of using this representation for the set of 5 objective functions De Jong used in his research. The floating-point and "concatenation" representations proved to be largely intractable. We did suggest the possibility of using an "interleaved" representation for multi-dimensional search spaces. This is another topic for future research.

Several other topics for further research suggest themselves. We might consider modifying the genetic algorithm to explicitly estimate certain Walsh coefficients. In particular, estimates of the low-order coefficients w(k(q)) referred to by corollary 4.2.5 could be used to give the user more confidence in the results (or a warning, if several w(k(q))'s are very small, that this may be a "misleading" function). Eventually, we would want to find some computationally fast ways to decide if a genetic algorithm is appropriate for a given objective function. As noted earlier, calculating $||f^{(r)}||$, $2 \leq r \leq \ell$, is not a simple matter for most realistic objective functions. Ultimately, we would like to find some fast ways to choose the best optimizer for a given function.

We should also consider designing and testing a hybrid

optimizer which uses both genetic operators and hill-climbing techniques. Such a hybrid method may be able to take advantage of the genetic algorithm's ability to perform global searching and the hill climber's ability to very rapidly converge to a local optimum. This hybrid optimizer would be especially attractive for use on objective functions which were known to be twice differentiable, but suspected of being multi-modal.

The major problems to be solved here are:

1. How to represent points in the search space so that both optimization methods can be used.
2. How to choose points to be used as starting points for the hill-climber.
3. How to coordinate the activities of the two optimizers. Should both run in parallel? Should they alternate?

One possibility would be to use the genetic algorithm first, then apply the hill climber to the most promising points produced by the genetic algorithm. This would work best if we could stop the genetic algorithm before it converged to a single peak. Then, if we could identify points which are clustered about the same peak, we could use one point from each cluster as a starting point for the hill-climber. At this time, the change in representation from binary strings (fixed-point probably) to floating-point (or perhaps extended-precision or "rational") representation

could be made. We could then run the hill-climber on each selected starting point. The best final value found would be our guess at the optimum. Alternatively, we could repeat the cycle a second time, using the local optima found by the hill-climber, plus additional points chosen at random if needed, as the initial population for the second run of the genetic algorithm.

The two questions which must be answered to make this design for a hybrid optimizer workable are:

1. Exactly how can we decide when to stop the genetic optimizer?
2. How can we identify the "clusters" of points which are associated with the various local optima?

The second question might be answered by using a clustering algorithm from the literature [20]. We leave the first question as a problem for further research.

APPENDIX

APPENDIX

PROOFS FOR SOME LEMMAS FROM CHAPTER THREE

Lemma 3.2.5

$$r(p;\ k) = \begin{cases} 1, & \text{if } k_p = 0 \\ 0, & \text{if } k_p = 1 \end{cases}$$

Proof: We proceed by induction on p.

Basis (p = 1) This is part of definition 3.2.1.

Induction Step By definition 3.2.1

$$r(p;\ k) = r(p;\ k_1k_2\ldots k_\ell) = r(p - 1;\ k_2k_3\ldots k_\ell 0).$$

By the induction hypothesis

$$r(p;\ k) = r(p - 1;\ k_2k_3\ldots k_\ell 0) = \begin{cases} 1, & \text{if } k_p = 0 \\ 0, & \text{if } k_p = 1. \end{cases}$$

This is the desired result.

Lemma 3.2.6

$$wal(j_1j_2\ldots j_\ell;\ k_1k_2\ldots k_\ell) = (-1)^z$$

where $z = \sum_{p=1}^{\ell} j_pk_p$.

Proof: By definition 3.2.3

$$wal(j_1j_2\ldots j_\ell;\ k_1k_2\ldots k_\ell) = \prod_{p=1}^{\ell} [r(p;\ k_1k_2\ldots k_\ell)]^{j_p}$$

Using lemma 3.2.5 we may rewrite r(p; k) to get

$$\text{wal}(j;\ k) = \prod_{p=1}^{\ell} (-1)^{k_p j_p}$$

$$= (-1)^z$$

where $z = \sum_{p=1}^{\ell} j_p k_p$.

Lemma 3.2.7

$$\sum_k \text{wal}(m;\ k)\ \text{wal}(n;\ k) = \begin{cases} 0, & \text{if } m \neq n \\ 2^{\ell}, & \text{if } m = n. \end{cases}$$

Proof: First we note that

$$\text{wal}(m;\ k)\ \text{wal}(n;\ k) = \prod_{p=1}^{\ell} (-1)^{m_p k_p} \prod_{p=1}^{\ell} (-1)^{n_p k_p}$$

$$= \prod_{p=1}^{\ell} (-1)^{(m_p+n_p)k_p} = \prod_{p=1}^{\ell} (-1)^{j_p k_p}$$

$$= \text{wal}(j;\ k)$$

where $j = j_1 j_2 \cdots j_{\ell}$, $j_p = \begin{cases} 0, & \text{if } m_p = n_p \\ 1, & \text{if } m_p \neq n_p. \end{cases}$

So $\sum_k \text{wal}(m;\ k)\ \text{wal}(n;\ k) = \sum_k \text{wal}(j;\ k)$.

We note that $j = 00\ldots0$ if and only if $m = n$. So we have two cases to consider.

Case 1 $m \neq n$, $j \neq 00\ldots0$. Let q be an index such that $j_q = 1$. Then

$$\sum_{k_q=1} wal(j;\ k) = \sum_{k_q=1} \prod_{p=1}^{\ell} (-1)^{j_p k_p}$$

$$= \sum_{k_q=1} (-1)^{k_q} \prod_{p\neq q} (-1)^{j_p k_p}$$

$$= (-1) \sum_{k_q=0} \prod_{p=1}^{\ell} (-1)^{j_p k_p}$$

$$\text{So } \sum_{k} wal(j;\ k) = \sum_{k_q=0} wal(j;\ k) + \sum_{k_q=1} wal(j;\ k) = 0.$$

Case 2 $m = n$, $j = 00\ldots0$. So $wal(j;\ k) = 1$ for all k.

$$\sum_{k} wal(j;\ k) = \sum_{k} 1 = 2^{\ell}.$$

Lemma 3.2.10

If w is the Walsh transform of f, then

$$w(m) = 2^{-\ell} \sum_{k} f(k)\ wal(m;\ k).$$

Proof: By definition 3.2.9

$$f(k) = \sum_n w(n)\ wal(n;\ k).$$

$$\text{So} \quad \sum_k f(k)\ wal(m;\ k) = \sum_k \left(\sum_n w(n)\ wal(n;\ k) \right) wal(m;\ k)$$

$$= \sum_n w(n) \sum_k wal(n;\ k)\ wal(m;\ k).$$

By lemma 3.2.7, the summation over k vanishes except when $m = n$. So

$$\sum_k f(k)\ wal(m;\ k) = 2^{\ell}\ w(m)$$

$$\text{or} \quad w(m) = 2^{-\ell} \sum_k f(k)\ wal(m;\ k).$$

REFERENCES

REFERENCES

1. Bethke, A. D. "Comparison of Genetic Algorithms and Gradient-Based Optimizers on Parallel Processors: Efficiency of Use of Processing Capacity." Technical Report No. 197, Logic of Computers Group, University of Michigan. 1976.

2. Bethke, A. D. and B. P. Zeigler and D. M. Strauss. "Convergence Properties of Simple Genetic Algorithms." Technical Report No. 159, Department of Computer and Communication Sciences, University of Michigan. 1974.

3. Birta, L. G. "Some investigations in Function Minimization." IEEE Transactions on Systems, Man, and Cybernetics, Vol. 6, No. 3 (March, 1976), p. 186 - 197.

4. Bosworth, J. L. and N. Foo and B. P. Zeigler. "Comparison of Genetic Algorithms with Conjugate Gradient Methods." Technical Report No. 00312-1-T, Department of Computer and Communication Sciences, University of Michigan. 1972.

5. Cavicchio, D. J. Jr. "Adaptive Search Using Simulated Evolution." Doctoral Thesis, Department of Computer and Communictaion Sciences, University of Michigan. 1970.

6. De Jong, K. A. "Analysis of the Behavior of a Class of Genetic Adaptive Systems." Doctoral Thesis, Department of Computer and Communication Sciences, University of Michigan. 1975.

7. Fine, N. J. "On the Walsh Functions." Transactions of the American Mathematical Society, Vol. 65, #3 (May, 1949), p. 372 - 414.

8. Frantz, D. R. "Non-linearities in Genetic Adaptive Search." Doctoral Thesis, Department of Computer and Communication Sciences, University of Michigan. 1972.

9. Hanson, J. N. and P. Russo. "Experiments with Symbol Manipulation Aided Optimization, Newton's Method Example." Technical Report, Computer and Information Science Department, Cleveland State University. 1978.

10. Holland, J. H. Adaptation in Natural and Artificial Systems. University of Michigan Press. 1975.

11. Kremer, H. "Representations and Mutual Relations of the Different Systems of Walsh Functions." Theory and Applications of Walsh Functions and other Non-Sinusoidal Functions, Hatfield Polytechnic, 1973.

12. Landwehr, G. "A Comparison of Discrete Walsh and Fourier Transforms." Theory and Applications of Walsh Functions and other non-Sinusoidal Functions, Hatfield Polytechnic, 1973.

13. Martin, N. "Convergence Properties of a Class of Probabilistic Adaptive Schemes called Sequential Reproductive Plans." Technical report No. 146, Department of Computer and Communication Sciences, University of Michigan. 1973.

14. Polak, E. Computational Methods in Optimization. Academic Press. 1971.

15. Russell, D. L. Optimization Theory. W. A. Benjamin, Inc. 1970.

16. Schumer, M. A. and K. Steiglitz. "Adaptive Step Size Random Search." IEEE Transactions on Automatic Control, Vol. 13 (1968), p. 270+.

17. Shanks, J. L. "Computation of the Fast Walsh-Fourier Transform." IEEE Transactions on Computers, May, 1969, p. 457 - 459.

18. Symposium on the Application of Walsh Functions. Naval Research Laboratory, Washington, D.C. 1970, 1971, 1972 Proceedings.

19. Theory and Applications of Walsh Functions and other Non-Sinusoidal Functions. A Colloquim held at the Hatfield Polytechnic, June 28 and 29, 1973, Hatfield, Herts., UK.

20. Torn, A. A. "Cluster Analysis Using Seed Points and Density-Determined Hyperspheres as an Aid to Global Optimization." IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-7, No. 8 (August, 1977), p. 610 - 616.

21. Walsh, J. L. "A Closed Set of Orthogonal Functions." American Journal of Mathematics, Vol. 55 (1923), p. 5 - 24.

22. Yuen, C. K. "Approximation Errors of a Walsh Series." Symposium on the Application of Walsh Functions, Naval Research Laboratory, Washington, D.C. 1972 Proceedings.

23. Zeigler, B. P. and J. L. Bosworth and A. D. Bethke. "Noisy Function Optimization by Genetic Algorithms." Technical Report No. 143, Department of Computer and Communication Sciences, University of Michigan. 1973.