*Complex Systems* **3** (1989) 129–152

# Genetic Algorithms and Walsh Functions: Part I, A Gentle Introduction

**David E. Goldberg**
*Department of Engineering Mechanics, The University of Alabama, Tuscaloosa, AL 35487, USA*

**Abstract.** This paper investigates the application of Walsh functions to the analysis of genetic algorithms operating on different coding-function combinations. Although these analysis tools have been in existence for some time, they have not been widely used. To promote their understanding and use, this paper introduces Bethke's Walsh-schema transform through the *Walsh polynomials*. This form of the method provides an intuitive basis for visualizing the nonlinearities being considered, thereby permitting the consideration of a number of useful extensions to the theory in Part II.

## 1. Introduction

Ever since the inner workings of genetic algorithms were unmasked by the development of the theory of schemata [5–8], researchers have known that genetic algorithms (GAs) work well when *building blocks* — short, low-order schemata with above-average fitness values — combine to form optima or near-optima. For many years, checking this requirement in a particular coding-function combination was a tedious computation. Then, Bethke [1] discovered an efficient method for calculating schema average fitness values using *Walsh functions*. Bethke's work shed much needed light on what makes a function simple or hard for a genetic algorithm; it also permitted the design of functions with a specified degree of *deception*, where low-order building blocks lead away from the optimum. Despite the fundamental importance of this work, it has largely been ignored in subsequent GA research.

This paper aims to remedy this oversight by explaining Bethke's technique in a straightforward manner. Specifically, we approach the analysis using *Walsh polynomials*. By doing so, we gain important physical insight concerning the processing of schemata and are better able to visualize the interactions of functions and codings using straightforward algebraic methods.

In the remainder of the paper, the Walsh functions are presented in monomial form, fitness functions are written as a weighted sum of the Walsh functions, and a direct proof of Bethke's schema average calculation is given.

Thereafter, these analysis techniques are illustrated through several specific examples: the design and analysis of both easy and hard functions is presented. To help promote the more widespread use of these techniques, Pascal code implementing the fast Walsh transform (FWT) is presented as an appendix to the paper. Part II will generalize the techniques of this paper to arbitrary coding-function combinations by defining a formal technique for performing static analysis of deception. It will also consider techniques of algebraic coding-function analysis and the extension of these methods to problems with constraints.

## 2. Bit strings, fitness values, and Walsh functions

In this section, we connect the binary-coded problems faced by many GAs to the Walsh functions. To do so, we must agree on some standard notation, introduce the Walsh polynomials, and show their relationship to the usual coding-function problem.

### 2.1 Some preliminaries

Before we embark on this mathematical journey, we need to establish the common terminology we will use along the way. We assume that our genetic algorithm processes $l$-bit strings, $\mathbf{x} = x_l x_{l-1} \ldots x_2 x_1$, where the bold face is used to denote an entire string (a bit vector) and each bit, $x_i \in \{0,1\}$, is subscripted by its position. We further assume that whatever our real-valued objective function $f$ and whatever our decision variables $\mathbf{d}$, there exists a mapping $\mathbf{g}$ (usually, but not necessarily, invertible) from the strings into the decision variables, $\mathbf{d} = \mathbf{g}(\mathbf{x})$. These assumptions allow us to conclude that for a given problem there exists a single-valued mapping from the strings into the reals:

$$f(\mathbf{d}) = f(\mathbf{g}(\mathbf{x})). \tag{2.1}$$

This notation simply formalizes the usual GA notion of coding the decision variables of a problem as a bit string [4]. For example, if we are maximizing the objective function $f(d) = d^2$, and the single decision variable $d$ is coded as a three-bit, unsigned binary integer, we obtain the following relationships among $f$, $d$, $\mathbf{g}$, and $\mathbf{x}$:

$$f(d) = d^2, \tag{2.2}$$

$$d = g(\mathbf{x}) = \sum_{i=1}^{3} x_i 2^{i-1}, \tag{2.3}$$

$$f(\mathbf{x}) = \left( \sum_{i=1}^{3} x_i 2^{i-1} \right)^2, \quad x_i \in \{0,1\}. \tag{2.4}$$

At this juncture, we simply notice how a polynomial at the decision-variable level induces a polynomial at the bit level.

Since we will also investigate important similarities among the strings, we must also introduce consistent notation to discuss the *schemata* or similarity subsets. Formally, a schema is a similarity subset containing strings with well-defined similarity at some number of positions. For example, the subset $\{000, 001, 010, 011\}$ is the schema containing all those strings of length $l = 3$ defined by a 0 in their leftmost position. Notationally, we may describe the schemata using *similarity templates* by considering length $l$ strings over an alphabet augmented by a wildcard or don't-care character, $*$. For example, the previously enumerated subset may be described by the similarity template $0**$. This template notation generalizes to permit the definition of any schema $\mathbf{h}$ in string form:

$$\mathbf{h} = h_l h_{l-1} \ldots h_2 h_1, \quad h_i \in \{0, 1, *\}. \tag{2.5}$$

The notation also makes it easy to count the number of unique schemata. Since a 0, 1, or $*$ can exist at any of the $l$ positions in the template, there must be $3^l$ different schemata over the $2^l$ binary strings. This is, of course, a small fraction of the $2^{2^l}$ subsets that can be formed among the set of $l$-bit strings.

To characterize the processing of different schemata by different genetic operators, we usually define a schema's *order* and its *defining length*. A schema $\mathbf{h}$'s order, $o(\mathbf{h})$, is the number of fixed positions of similarity in the subset. For example, the order of the schema $1**$ is 1 and the order of $00*$ is 2. The defining length of a schema, $\delta(\mathbf{h})$, is simply the distance between the outermost defining positions of a schema. For example, $\delta(1**) = 0$, $\delta(*00) = 1$, and $\delta(1*1) = 2$.

With these definitions, we may immediately calculate a lower bound on the expected number of a particular schema $\mathbf{h}$ following reproduction, crossover, and mutation (Goldberg, 1989; Holland, 1975):

$$m(\mathbf{h}, t+1) \geq m(\mathbf{h}, t) \frac{f(\mathbf{h})}{\bar{f}} \left[ 1 - p_c \frac{\delta(\mathbf{h})}{l-1} - p_m o(\mathbf{h}) \right], \tag{2.6}$$

where $m$ is the expected number of schemata, $t$ is the generation index, $f(\mathbf{h})$ is the average fitness of those strings representing the subset $\mathbf{h}$, $\bar{f}$ is the average fitness of the population, $p_c$ is the crossover probability, and $p_m$ is the mutation probability. This result, the *fundamental theorem of genetic algorithms* or simply the *schema theorem*, says that a schema grows when it is short, of low-order, and has above-average fitness. To examine a particular coding-function combination, we must examine whether the short, above-average schemata combine to form optimal or near-optimal strings. To do this, we must first consider the usual basis for a GA problem and see whether a change of basis would make schema average computation more convenient.

## 2.2 The canonical basis

One of the nice things about genetic algorithms is that by using an underlying string representation, all problems "look" the same to a GA. For example, in

| String | Fitness |
|---|---|
| 000 | $f_{000}$ |
| 001 | $f_{001}$ |
| ⋮ | ⋮ |
| 110 | $f_{110}$ |
| 111 | $f_{111}$ |

Table 1: GAs process only strings and their fitness values.

our three-bit problem, any deterministic function-coding combination may be reduced ultimately to a list of fitness values associated with each of the $2^3 = 8$ strings, as shown in table 1. This list-of-fitness-values approach to specifying a problem disguises the implied choice of a set of *basis* functions. A basis is simply a linearly independent set of functions which span the underlying space. In the present case, we may write any function over an $l$-bit string as the following sum:

$$f(j) = \sum_{i=0}^{2^l-1} f_i \delta_{ij}, \tag{2.7}$$

where the indices $i$ and $j$ are treated interchangeably as integers or their binary string counterparts, and the $\delta$ here is the Kronecker delta (not to be confused with the defining length) such that $\delta_{ij} = 1$ when $i = j$ and 0 otherwise. This choice of basis is useful for examining individual strings, but it becomes inconvenient when we wish to calculate schema averages. For example, direct calculation of the average fitness of an order $o(\mathbf{h})$ schema over strings of length $l$ requires the summation of $2^{l-o(\mathbf{h})}$ $f_i$ values. Furthermore, the canonical basis provides no intuitive insight into the interaction of the bitwise nonlinearities within the problem. For this reason, we seek a change of basis by considering the Walsh polynomials.

### 2.3 The Walsh polynomials

To introduce the Walsh polynomials, it is useful to define a number of simple functions over schemata. The $\alpha$ function is defined as follows:

$$\alpha(h_i) = \begin{cases} 0, & \text{if } h_i = *; \\ 1, & \text{if } h_i = 0, 1. \end{cases} \tag{2.8}$$

Using the $\alpha$ function, we can define the partition number $j_p$ of a schema $\mathbf{h}$ as follows:

$$j_p(\mathbf{h}) = \sum_{i=1}^{l} \alpha(h_i) 2^{i-1}. \tag{2.9}$$

In this way, each competing set of schemata—each similarity partition—is numbered uniquely. For example, the partition $***$ receives $j_p = 0$, $**f$ receives $j_p = 1$, $*f*$ receives $j_p = 2$, and $fff$ receives $j_p = 7$, where the $f$

| $j$ | $\psi_j(\mathbf{y})$ |
|---|---|
| 0 | 1 |
| 1 | $y_1$ |
| 2 | $y_2$ |
| 3 | $y_1y_2$ |
| 4 | $y_3$ |
| 5 | $y_1y_3$ |
| 6 | $y_2y_3$ |
| 7 | $y_1y_2y_3$ |

Table 2: Table of $\psi$ functions for $l = 3$.

denotes a fixed position (a 0 or a 1). In the future, we will drop the subscript $p$ unless the definition is unclear from the context.

The $\beta$ function may be defined as follows:

$$\beta(h_i) = \begin{cases} 0, & \text{if } h_i = 0, * \\ 1, & \text{if } h_i = 1. \end{cases} \tag{2.10}$$

With these definitions, we are prepared to define a different basis in the Walsh polynomials. To do so, we first define a mapping from the auxiliary string positions $y_i$ to the bit string positions $x_i$:

$$x_i = \frac{1}{2}(1 - y_i), \quad i = 1, \ldots, l. \tag{2.11}$$

Simply stated, a $-1$ in an auxiliary string maps to a 1 in the bit string, and a 1 in the auxiliary string maps to a 0 in the bit string. This simple linear mapping allows the usual multiplication operation to act as an exclusive-or operator (XOR — the $-1$ is necessary for the XOR operator; however, reversal of order — mapping a $-1$ to a 1 and mapping a 1 to a 0 — is needed only to make the definition agree with that of the Walsh function literature). With this definition, we may now calculate a set of $2^l$ monomials over the auxiliary string variables:

$$\psi_j(\mathbf{y}) = \prod_{i=1}^{l} y_i^{j_i}, \quad y_i \in \{-1, 1\}. \tag{2.12}$$

Notice how the index counter $j$ is used bit by bit in conjunction with the $\beta$ function to determine the presence or absence of a variable in the product. This is no accident, as it turns out that the functions are defined in the same sequence as the partition numbers $j_p$.

To better understand these terms, we enumerate them for the case $l = 3$ in table 2. To tie the table to the defining expression, simply note that for $j = 6 = 110_2$, bits two and three are set, thereby causing $y_2$ and $y_3$ to be included in the product.

Notice that the table contains all products with three terms or fewer, except those that would contain an exponent greater than one. Of course, with

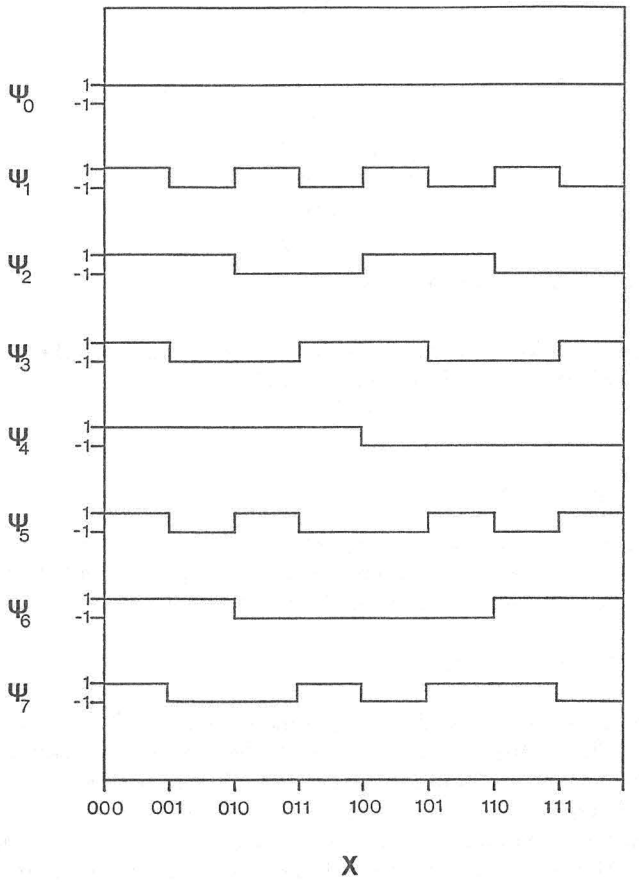


Figure 1: The eight $\psi$ functions versus $\mathbf{x}$ for string length $l = 3$.

the underlying group — multiplication over $\{-1, 1\}$ — exponents greater than one are redundant; even exponents result in a product of 1, and odd exponents result in the term itself:

$$y_i^a = \begin{cases} 1, & \text{if } a \text{ even;} \\ y_i, & \text{if } a \text{ odd.} \end{cases} \tag{2.13}$$

This fact will be useful in a number of subsequent calculations.

To better visualize the functions, function values are plotted versus $\mathbf{x}$ value in figure 1. Following Bethke [1], the numbering is bit-reversed from the usual binary-ordered numbering scheme of the Walsh function literature.

We note that the Walsh functions, $\psi_j$, form a basis for the reals over the strings as did the Kronecker delta function discussed earlier. This fact may be proved easily by mathematical induction by showing that none of the terms of a given order may be written as a linear combination of lower order terms. Thus, we may write the fitness function as a linear combination of the Walsh monomials:

$$f(\mathbf{x}) = \sum_{j=0}^{2^l-1} w_j \psi_j(\mathbf{x}). \tag{2.14}$$

As we have done in this equation, we will sometimes write the Walsh function with the argument $\mathbf{x}$, where the substitution of the mapping from the auxiliary string $\mathbf{y}$ to the bit string $\mathbf{x}$ is understood. More importantly, we see where the name Walsh polynomial comes from: any mapping from an $l$-bit string into the reals may be written as a linear combination of the Walsh monomials, yielding a polynomial.

Note that the $\psi_j$ basis is orthogonal:

$$\sum_{\mathbf{x}=0}^{2^l-1} \psi_i(\mathbf{x})\psi_j(\mathbf{x}) = \begin{cases} 2^l, & \text{if } i = j; \\ 0, & \text{if } i \neq j. \end{cases} \tag{2.15}$$

This may be proved by considering the cases $i = j$ and $i \neq j$ separately. In the first case, the sum contains $2^l$ products where each term in the product is squared. Using equation (2.15), we obtain $\sum_0^{2^l-1} 1 = 2^l$. In the second case, the sum contains $2^l$ terms, but the products must differ at one or more positions by the definition of the polynomials. Where the terms are identical, the product may be dropped as before; however, where they are different, the terms must be retained, and since the sum is taken over all $2^l$ strings, half the terms will result in a product of $-1$ and half will result in a product of 1. Thus, the sum must add to 0, thereby proving the result.

Using the orthogonality result of equation (3.1), we may calculate the Walsh coefficients as follows:

$$w_j = \frac{1}{2^l} \sum_{\mathbf{x}=0}^{2^l-1} f(\mathbf{x})\psi_j(\mathbf{x}). \tag{2.16}$$

Direct computation of the Walsh coefficients in this manner is, of course, possible; however, more efficient techniques analogous to the fast Fourier transform — the fast Walsh transform (FWT) — are possible and desirable. One FWT implementation is presented in appendix A.

## 3. Calculating schema averages with Walsh coefficients

In this section, we derive a relationship between the schema average fitness values and the Walsh coefficients. The result is subsequently applied to the analysis of a number of coding-function combinations and to the design of intentionally deceptive functions.

### 3.1 Calculating average schema fitness values

Given the Walsh coefficients of a function — given the $w_j$ values — we already know how to back-calculate the function values; however, if any useful GA analysis is to be done, we must identify schema average fitness values in terms of the $w_j$. We could, of course, calculate the fitness values and then calculate the schema average fitness conventionally by summing fitness values over all strings in the subset; however, this seems overly tedious, and Bethke's work [1] has shown that the Walsh functions yield a more direct method.

To derive this relationship between a schema's average fitness, $f(\mathbf{h})$, and the function's Walsh coefficients, $w_j$, consider the usual average fitness calculation, summing over all strings contained in the schema (contained in the subset) and dividing by the number of strings in the schema:

$$f(\mathbf{h}) = \frac{1}{|\mathbf{h}|} \sum_{\mathbf{x} \in \mathbf{h}} f(\mathbf{x}), \tag{3.1}$$

where $|\mathbf{h}|$ is the cardinality (the number of elements) of the subset $\mathbf{h}$. Substituting equation (2.16) into equation (3), we obtain

$$f(\mathbf{h}) = \frac{1}{|\mathbf{h}|} \sum_{\mathbf{x} \in \mathbf{h}} \sum_{j=0}^{2^l-1} w_j \psi_j(\mathbf{x}), \tag{3.2}$$

where once again the mapping from the underlying string $\mathbf{x}$ to the auxiliary string $\mathbf{y}$ is understood. Reversing the order of summation results in the following equation:

$$f(\mathbf{h}) = \frac{1}{|\mathbf{h}|} \sum_{j=0}^{2^l-1} w_j \sum_{\mathbf{x} \in \mathbf{h}} \psi_j(\mathbf{x}). \tag{3.3}$$

Further progress in reducing this equation expression comes from reducing the sum

$$S(\mathbf{h}, j) = \sum_{\mathbf{x} \in \mathbf{h}} \psi_j(\mathbf{x}). \tag{3.4}$$

Substituting the Walsh product, we obtain

$$S(\mathbf{h}, j) = \sum_{\mathbf{x} \in \mathbf{h}} \prod_{i=1}^{l} [y_i(x_i)]^{j_i}. \tag{3.5}$$

Since there are $|\mathbf{h}|$ terms in the sum and since each term is a Walsh monomial (with a value $+1$ or $-1$), the sum is bounded by $-|\mathbf{h}|$ and $|\mathbf{h}|$.

To see what value the sum acquires for some given $j$ and $\mathbf{h}$, consider the following examples. Suppose $j = 6 = 110_2$ and $\mathbf{h} = 01*$. The summation contains exactly two terms (exactly $|01*|$ terms) and both terms equal $-1$, because $y_3 y_2 = (1)(-1) = -1$; thus, the sum equals $-2 = -|\mathbf{h}|$. A more specific schema with the same set characters would change the magnitude of the result, but not the sign. For example, with $\mathbf{h} = 011$ the product

$y_3y_2 = (1)(-1) = -1$ still, but since there is only one element of the subset, the sum is $-1$.

Schemata that do not fix the terms of the Walsh product present a different situation. Like the schema $01*$, the schema $1*1$ still induces a sum of two terms, but the signs no longer remain the same. Because the partition dictates a product containing the second and third position variables, and because the schema allows the second position to vary, the summation equals $(-1)(-1) + (-1)(1) = 0$.

This result generalizes immediately to arbitrary strings and partition numbers. A nonzero multiplier for a given Walsh coefficient will be obtained only when the schema has fixed bits (not $*$'s) at the fixed positions of the $j$th partition; otherwise, any positions in the partition left unfixed will have equal numbers of 1's and $-1$'s, resulting in a null sum overall. Furthermore, any nonzero multipliers can only take on values of either $+|\mathbf{h}|$ or $-|\mathbf{h}|$, because fixing the positions of the schema corresponding to those in the Walsh monomial fixes the sign for all terms in the summation. Additionally, the sign of the summation is determined solely by the number of 1's contained in the fixed positions of the schema corresponding to the fixed positions of the partition. If the number of 1's is even, the sum is positive; if the number of 1's is odd, the sum is negative.

That the nonzero terms in the summation have magnitude $|\mathbf{h}|$ immediately cancels out the same term in the denominator, and we may write the schema average fitness as follows:

$$f(\mathbf{h}) = \sum_{j \in J(\mathbf{h})} w_j \psi_j(\beta(\mathbf{h})), \; J = \{j : (\exists i) : \mathbf{h} \subseteq \mathbf{h}_i(j)\}, \tag{3.6}$$

where the sum is taken over all partitions $j$ such that the schema $\mathbf{h}$ is a subset of exactly one of the competing schemata $\mathbf{h}_i(j)$ defined by the partition $j$ (here we assume that the schemata are indexed from 0 to $2^{o(\mathbf{h})} - 1$ as the fixed positions determined by the partition number run in the usual binary fashion from all 0's to all 1's). In words, we have shown that the schema average may be calculated as a partial, signed sum of the Walsh coefficients, where the only coefficients included in the sum are those associated with partitions that contain the schema, and the sign of a particular coefficient is positive or negative as the number of 1's covering those positions is even or odd. This same result was presented in somewhat different form by Bethke [1]. Holland [9] has extended Bethke's method to permit calculation of schema averages for nonuniformly distributed populations. These calculations reduce to Bethke's calculation in the uniform case.

This computation — we shall call it the *Walsh-schema transform* — is driven home in table 3 with a partial enumeration of the length $l = 3$ schemata and their fitness averages written in terms of the Walsh coefficients. Notice how the low-order schemata are specified with a short sum and how the high-order schemata are specified with a long sum. This is exactly the opposite situation of a schema average fitness calculation expressed using the canonical basis. There, low-order schemata require many terms

| Schema | Fitness average as sum of Walsh coefficients |
|---|---|
| *** | $w_0$ |
| **0 | $w_0 + w_1$ |
| **1 | $w_0 - w_1$ |
| 1** | $w_0 - w_4$ |
| *00 | $w_0 + w_1 + w_2 + w_3$ |
| *01 | $w_0 - w_1 + w_2 - w_3$ |
| *11 | $w_0 - w_1 - w_2 + w_3$ |
| 11* | $w_0 - w_2 - w_4 + w_6$ |
| 001 | $w_0 - w_1 + w_2 - w_3 + w_4 - w_5 + w_6 - w_7$ |
| 101 | $w_0 - w_1 + w_2 - w_3 - w_4 + w_5 - w_6 + w_7$ |

Table 3: Some $l = 3$ schemata and their fitness averages as partial Walsh sums.

and high-order schemata require relatively few terms. It is this inversion of the schema average computation that makes the Walsh basis so useful for identifying and designing deceptive problems.

### 3.2 What do the Walsh coefficients mean?

The relationship between $f(\mathbf{h})$ and the coefficients $w_j$ is now clear, but the analysis leaves us with little intuition regarding the meaning of the $w_j$. To develop a better intuitive feel for the Walsh coefficients, we adopt the perspective of building blocks more literally. Specifically, we imagine that higher order schemata are constructed from their lower order constituents and we investigate how new coefficients are accumulated in the increasingly accurate summation. For example, with strings of length $l = 3$ consider how the schema $*01$ might be constructed from its underlying constituents $***$, $*0*$, and $**1$. Comparing the schema average fitness computation for each of the schemata, we note an interesting thing:

$$\begin{aligned} f(***) &= w_0; \\ f(**1) &= w_0 - w_1; \\ f(*0*) &= w_0 + w_2; \\ f(*01) &= w_0 - w_1 + w_2 - w_3. \end{aligned}$$

As schemata become increasingly specific, additional terms are included in their Walsh sum. For example, the fitness of schema $**1$ and that of schema $***$ differ by an amount $-w_1$. Similarly, the fitness of schema $*1*$ differs from that of schema $***$ by an amount $w_2$. Calling the fitness difference associated with the schema $\mathbf{h}$, $\Delta f$, we may view the fitness of a higher order schema as estimated by the summation of the lower order $\Delta f$s. In the particular case, we would say that the sum $w_0 - w_1 + w_2$ is a first order estimate of the fitness of the schema $*01$. Moreover, this calculation leads us to calculate the $\Delta f$ for the schema $*01$ as simply $-w_3$. Thus, we can interpret a Walsh coefficient as representing the magnitude of the difference between

| $j$ | x | $f(\mathbf{x})$ | $w_j$ |
|---|---|---|---|
| 0 | 000 | 10.00 | 7.55 |
| 1 | 001 | 15.00 | −2.50 |
| 2 | 010 | 0.00 | 5.00 |
| 3 | 011 | 5.00 | 0.00 |
| 4 | 100 | 10.10 | −0.05 |
| 5 | 101 | 15.10 | 0.00 |
| 6 | 110 | 0.10 | 0.00 |
| 7 | 111 | 5.10 | 0.00 |

Table 4: Walsh coefficients of a bitwise linear function.

a higher order approximation to a schema's fitness and its next lowest order approximation.

More rigorously, $\hat{f}^{(o)}(\mathbf{h})$ may be defined as the $o$th-order approximation to the fitness of the schema $\mathbf{h}$; the magnitude of the difference between a schema average and its next lowest order approximation is simply the highest order Walsh coefficient in the sum. This perspective of assembling functions from lower order schemata coincides nicely with the way GAs actually work; it will also enable us to understand what kinds of functions are easy and hard for genetic algorithms.

## 4. Example calculations using the Walsh-schema transform

In this section we solidify our understanding of the calculations of the previous section with some examples. We examine a linear function and a number of nonlinear functions to try to better understand what makes a function easy or hard for a simple GA.

### 4.1 Walsh coefficients and bitwise linear functions

Before we examine the role of Walsh coefficients in analyzing nonlinear functions, it is useful to consider a bitwise linear function:

$$f(x_3x_2x_1) = 10 + 5x_1 - 10x_2 + 0.1x_3, \quad x_i \in \{0,1\}.$$

Recognizing that only positive coefficients should be included in the sum to attain a maximum, we calculate an optimum of $f^* = 15.1$ at $\mathbf{x} = 101$. Less obvious is the fact that the average value of the function is $\bar{f} = 7.55$, as may be calculated directly by enumeration.

To explore the relationship between the Walsh coefficients, schema averages, and the function optimum, we take the Walsh transform of the function using the code of the appendix and present the coefficients along with function values in table 4. Scanning the table, we note that the order-zero Walsh coefficient has the same value as the average function value. This is no accident, because $f(***) = w_0$. We also note that the only other nonzero schemata are associated with the one-bit partitions ($j = 1, 2, 4$). This makes

sense when we recognize that we are dealing with a bitwise linear function, and a first-order estimate is exact. This same line of reasoning suggests that the optimum may be found by the consideration of the first-order Walsh coefficients alone. We simply choose a 0 or a 1, depending upon whether the corresponding Walsh coefficient associated with the corresponding position is negative or positive respectively.

These results generalize immediately to an arbitrary $l$-bit, linear function:

$$f(\mathbf{x}) = b + \sum_{i=1}^{l} a_i x_i, \quad x_i \in \{0,1\}. \tag{4.1}$$

Some straightforward calculation yields the following Walsh coefficients:

$$w_j = \begin{cases} b + \frac{1}{2}\sum_{i=1}^{l} a_i, & \text{if } j = 0; \\ -\frac{a_j}{2}, & \text{if } j \in \{2^{i-1}, \quad i = 1,2,\ldots,l\}; \\ 0, & \text{otherwise.} \end{cases} \tag{4.2}$$

Thus, only the order-zero and order-one coefficients can be nonzero, and every bit predicts the optimum correctly, depending upon the sign of the associated Walsh coefficient.

This simple function is not of much direct, practical interest, because the optimization of known bitwise linear functions is not a difficult problem; however, when we consider the accurate manipulation of schema averages inherent in even simple GAs, we see that processing of one-bit averages amounts to the combined construction and utilization of a model that is at least of first-order accuracy. If we refer back to the expression for the schema theorem (equation 2.8), we realize that all one-bit schemata are treated with little crossover-mutation loss because the term

$$[1 - p_c \frac{\delta(\mathbf{h})}{l-1} - p_m o(\mathbf{h})] = 1 - p_m$$

is close to 1 for small $p_m$. Although the higher-order approximations to the function are treated less graciously by crossover, it is this *inverted cascade* of building blocks that constructs optimal or near-optimal solutions in GAs. This perspective of simultaneously building and using an increasingly refined, bitwise model of the fitness will become clearer as we study some nonlinear functions and deception.

### 4.2 An easy nonlinear function

Linear function optimization is unlikely to make us famous. Nor is it likely to provide us with any insight regarding the deception that can cause simple GAs difficulty. To start our brief foray into nonlinear function analysis, consider the coding-function combination we posed earlier in section 2 (equations 2–2.6). There, we described the fitness function $f(d) = d^2$, where $d$ was coded as an unsigned binary integer of length $l = 3$. This function takes on a maximum of $f^* = 49$ at $\mathbf{x} = 111$, as can be seen from the enumeration of fitness values in table 5. Using the FWT code of the appendix, the

| $j$ | x | $f(\mathbf{x})$ | $w_j$ |
|---|---|---|---|
| 0 | 000 | 1 | 17.5 |
| 1 | 001 | 2 | −3.5 |
| 2 | 010 | 4 | −7.0 |
| 3 | 011 | 9 | 1.0 |
| 4 | 100 | 16 | −14.0 |
| 5 | 101 | 25 | 2.0 |
| 6 | 110 | 36 | 4.0 |
| 7 | 111 | 49 | 0.0 |

Table 5: Function values and Walsh coefficients of $f(d) = d^2$, $d$ an unsigned binary integer.

Walsh coefficients have also been calculated and listed in the table. To determine whether this function should be difficult for a GA, we simply consider whether the low-order approximations to the function predict the optimum. The zeroth-order approximation to the optimum is simply $f(***) = w_0$. This, of course, tells us that the average fitness is 17.5, but it provides no further clues. The first-order approximation to the optimum point may be calculated as follows:

$$\begin{aligned} \hat{f}^{(1)}(111) &= w_0 - w_1 - w_2 - w_4; \\ &= 17.5 + 3.5 + 7 + 14; \\ &= 42. \end{aligned}$$

This estimate is very suggestive that we are on the right track, because it is the best among all order-one estimates (all its terms are positive).

Proceeding further, we notice that the second-order approximation to the optimum may be calculated as

$$\begin{aligned} \hat{f}^{(2)}(111) &= w_0 - w_1 - w_2 + w_3 - w_4 + w_5 + w_6; \\ &= 17.5 + 3.5 + 7 + 1 + 14 + 2 + 4; \\ &= 49. \end{aligned}$$

Note that the difference between the second- and first-order estimates, $\Delta\hat{f}^{(2)}(111) = 49 - 42 = 7$, is positive, thereby reinforcing the judgment of the first-order estimate. Moreover, the second-order estimate exactly calculates the optimum value of the string, a fact that is further verified by checking the value of the third-order coefficient, $w_7 = 0.0$.[1] This reasoning leads to an interesting conclusion. Despite the existence of nonzero, higher-order $w$'s, the coding-function combination is not difficult for a genetic algorithm, because low-order schemata correctly predict the optimum

[1]We won't belabor the point at this juncture, but the existence of no nonzero coefficients above the second-order looks suspiciously related to the degree of the underlying polynomial objective function. Actually this fact is related to the degree of the polynomial and to the linear mapping implied in the usual unsigned binary integer coding. We will come back to this notion when we generalize the result in Part II.

and all higher-order estimates only reinforce the low-order estimates. In other words, all roads lead to Rome and the GA is unlikely to make major errors.

Of course, this type of static analysis does not ensure that the GA will not stray due to stochastic effects, nor does the static analysis preclude the possibility that deceptive problems will be solved correctly due to dynamic effects which go unaccounted in the static analysis. The former difficulty, nonconvergence due to stochastic effects, occurs when certain building blocks possess insufficiently high signal-difference-to-noise ratios. When this occurs, those building blocks are not under much selective pressure and their proportion in the population tends to wander randomly. In small populations, this wandering may result in convergence to a suboptimal schema, a condition that biologists have called *genetic drift*. Connecting these problems to our easy nonlinear function and table 5, we recognize that problems are most likely when the $w$ values associated with a building block are relatively small. Comparing the values of the three, first-order $w$'s, $w_1 = -3.5$, $w_2 = -7.0$, and $w_4 = -14.0$, we see that the magnitude of the leftmost bit is four times that of the rightmost bit, suggesting that if we are to have difficulty optimizing this function, it will occur to the right. We note that if the rightmost bit were to become fixed at an incorrect value — if it were to hitchhike into a dominant position — a single mutation should correct the problem, and we would simply be waiting for the somewhat better string to take over. Deterministically, the time for this event could be estimated by an equation derived in a previous paper [3]:

$$t = \frac{r+1}{r-1} \ln(n-1), \tag{4.3}$$

where $t$ is the takeover time, $r$ is the ratio of the fitness of the best to the second best alternative, and $n$ is the population size. Stochastically, this equation is only valid if the difference between first- and second-best alternatives is significantly larger than the noise, induced or actual, of the fitness function (in many cases this assumption becomes valid — the noise reduces sufficiently — for low-$w$ building blocks late in a run near convergence).

### 4.3 Designing a deceptive function

Thus far, we have not seen how nonlinearities can cause much difficulty. Certainly there are problems in which first-order estimates fail to predict the optimum. In fact, we need not look very far to find them. Here we construct a deceptive function where the first-order schemata do not predict the optimum. To do this, we consider a two-bit function where, without loss of generality, we assume that point $f_{11}$ is the best. For a GA to be misled, the one-bit schemata must dictate either

$$\begin{aligned} f(*0) &> f(*1), \text{ or} \\ f(0*) &> f(1*) \end{aligned}$$

or both. These conditions are particularly easy to write in terms of the Walsh coefficients; with straightforward manipulation they reduce to

$$w_1 > 0, \text{ or}$$
$$w_2 > 0.$$

For maximum deception we would like to maintain both of these conditions simultaneously. To see that this is not possible in a two-bit problem, consider the three optimality conditions:

$$f_{11} > f_{00};\ f_{11} > f_{01};\ f_{11} > f_{10}.$$

These may be written in terms of the Walsh coefficients with some algebraic reduction as follows:

$$w_1 < -w_2;\ w_3 > w_2;\ w_3 > w_1.$$

Clearly the first of these conditions ($w_1 < -w_2$) cannot be met if both deception conditions are to be enforced. Thus, we pick the first of the deception conditions ($w_1 > 0$) and conclude that there are two cases to consider: $w_2 + w_3 > 0$ and $w_2 + w_3 < 0$. These correspond to the cases $f_{01} < f_{00}$ and $f_{01} > f_{00}$, and elsewhere [2] these have been called types I and II of the *minimal deceptive problem*, because the functions are the least nonlinear problems that can be deceptive. It is interesting that a simple GA consisting of reproduction and crossover is not usually misled by the MDP — the functions are GA-deceptive but not usually GA-hard (dynamically, the problem converges to the right answer despite static deception).

### 4.4 A fully deceptive function

Because of the limited number of degrees of freedom in the two-problem, we were only able to construct a problem that was partially deceptive: only one of the two, single-bit schemata led away from the optimum. It is natural to wonder whether a problem can be *completely deceptive* in that all lower-order schemata lead away from the true optimum. This is possible [1], and in this section we give a constructive proof by assembling just such a function.

We begin, as we did with the two-problem, by assuming without loss of generality that point 111 is the best. For full deception we require that all order-one and order-two schemata lead away from the best. For the one-bit schemata this means that

$$f(**1) < f(**0),\ f(*1*) < f(*0*),\ \text{and } f(**1) < f(**0).$$

In terms of the Walsh coefficients we obtain the following inequalities:

$$w_1 > 0,\ w_2 > 0,\ \text{and } w_4 > 0.$$

For the two-bit schemata we require

$$f(*00) > f(*01),\ f(*00) > f(*10),\ \text{and } f(*00) > f(*11).$$

| $j$ | x | $w_j$ | $f_j$ |
|---|---|---|---|
| 0 | 000 | 0 | 14 |
| 1 | 001 | 1 | 10 |
| 2 | 010 | 2 | 6 |
| 3 | 011 | 3 | −14 |
| 4 | 100 | 4 | −2 |
| 5 | 101 | 5 | −14 |
| 6 | 110 | 6 | −14 |
| 7 | 111 | −7 | 14 |

Table 6: Inverse transform for designing a fully deceptive function.

as well as the two other sets of three conditions over the other two-bit partitions. These conditions reduce to

$$w_1 + w_3 > 0,\ w_2 + w_3 > 0,\ \text{and}\ w_1 + w_2 > 0;$$
$$w_1 + w_5 > 0,\ w_4 + w_5 > 0,\ \text{and}\ w_1 + w_4 > 0;$$
$$w_6 + w_4 > 0,\ w_6 + w_2 > 0,\ \text{and}\ w_2 + w_4 > 0.$$

Finally, the seven optimality conditions round out the picture and may be written in terms of the Walsh coefficients as follows:

$$\begin{aligned} -(w_1 + w_2 + w_4) &> w_7; \\ w_5 + w_3 &> w_2 + w_4; \\ w_6 + w_3 &> w_1 + w_4; \\ w_5 + w_6 &> w_4 + w_7; \\ w_5 + w_6 &> w_1 + w_2; \\ w_3 + w_5 &> w_2 + w_7; \\ w_3 + w_6 &> w_1 + w_7. \end{aligned}$$

Many sets of coefficients satisfy these conditions; however, noting the pairings of coefficients in the optimality conditions, we can say that the coefficients

$$w_j = j,\ j = 1, \ldots, 6$$

satisfy both optimality and deception conditions as long as $w_7 < -7$ and $w_0$ is chosen to make the fitness values nonnegative. Setting $w_7 = -7$ (for borderline deception) and $w_0 = 0$, we take the inverse transform of the coefficients and obtain the results presented in table 6. Notice that the worst value in the table is −14. This can be made non-negative by increasing $w_0$ by 14, thereby raising all fitness values 14 units. Notice also that points 000 and 111 have the same value. Point 111 can be made the best by decreasing the coefficient $w_7$ an amount $c$; this action raises the fitness of point 111 an amount $c$ and similarly increases the fitness values of any point whose binary representation contains an odd number of 1's (points 001, 010, and

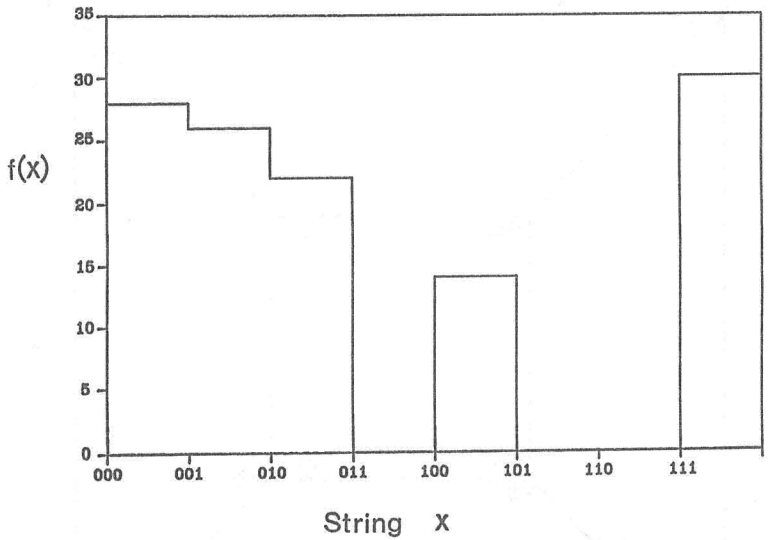


Figure 2: A fully deceptive function plotted as a function of a single, binary-ordered variable.

100). Contradistinctively, points 000, 011, 101, and 110 will have their fitness values reduced an amount $c$. Thus, to maintain non-negativity of the fitness values, an additional amount $c$ (on top of the 14 already added) must be added to the $w_0$ coefficient. Figures 2 and 3 show this fully deceptive 3-bit function plotted in binary-ordered and squashed-Hamming-cube formats respectively.

These methods may be developed more rigorously to design and analyze deceptive problems of higher order. Some of the questions that arise in generalizing these techniques are discussed in the next section.

## 5. Some questions and some partial answers

Our discussion thus far has dealt with some specifics of Walsh analysis of the function-coding combinations relevant to genetic algorithms: the details of Walsh functions, the Walsh-schema transform, and some sample calculations. Whether these techniques can be generalized depends on our ability to answer a number of key questions:

Can deception be defined more rigorously?

Can deception be analyzed using these methods under arbitrary string codings, parameter mappings, and functions?

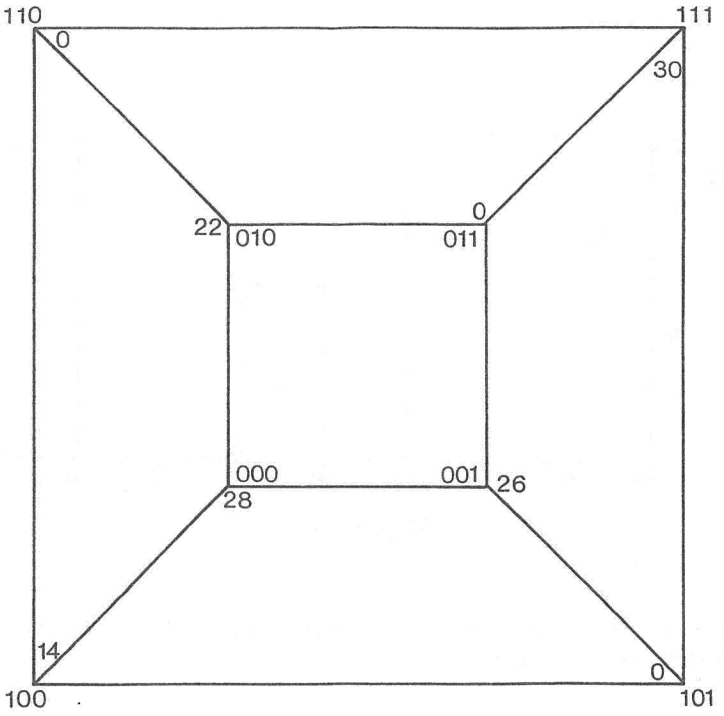


Figure 3: A fully deceptive function represented by a squashed Hamming cube shows the isolation of the fully deceptive optimum.

Can constraints be incorporated into the analysis?

Can anything be learned about building block processing without performing complete Walsh transforms?

All these questions may be answered in the affirmative. In Part II, we shall address each one quite carefully. Here, let's take a peek at things to come.

To define deception more rigorously, we must avoid the loose talk of this paper concerning low-order schemata "leading" to the optimum. Instead we should define the point or points to which a genetic algorithm should be expected to converge under perfect sampling conditions. The definition of such a *static fixed point* will itself go a long way toward motivating a rigorous *analysis of deception* (ANODE) procedure.

Once this is done, we may use such an analysis procedure to calculate fixed points in arbitrary string codings, parameter mappings, and functions. We have already acknowledged how the use of intermediate mappings simply introduces the composition of additional functions. Instead of writing $f =$

$f(\mathbf{x})$, we might write $f = f(g(h(\mathbf{x})))$ where $h$ might be some odd decoding function and $g$ might be some parameter mapping. That many of our $f$, $g$, and $h$ functions may be written as finite or truncated polynomials and that the Walsh functions themselves may be written as monomial terms, suggests that perhaps we should look at polynomials as the *lingua franca* of coding-function analysis. Doing so raises the possibility of performing analysis of deception without the explicit use of transform methods. The development of such *algebraic coding-function analysis* procedures (ACFA procedures) will permit full or partial analysis of deception by inspection or simple algebraic manipulation.

Analyzing whether deception is induced in an otherwise undeceptive problem by the imposition of constraints may be tackled using transforms and some sensitivity analysis. Penalizing infeasible subsets of points sufficiently to cause them to fall below the constrained optimum may be viewed as the summed effect of pointwise changes to fitness values of the points in the infeasible subset. In turn, these changes may be viewed as changing $w$ values and schema fitness values, thereby permitting an analysis of the change in deception. The procedure is similar to that used in the design of the fully deceptive problem of section 4.4; it may be useful in a number of analysis and function design situations.

In Part II, we will examine these and other questions more rigorously. So doing will provide further insight into the building block processing that underlies genetic algorithm power.

## 6. Conclusions

In this paper, Walsh functions have been used to help analyze the workings of genetic algorithms. Specifically, the Walsh monomials and polynomials have been defined and schema average fitness values have been calculated as a partial, signed summation over the Walsh coefficients. Although this analysis agrees with those that have gone before, the introduction of a polynomial form presents a common language for discussing the usual composition of mappings that occurs in GAs when codings are decoded, mapped, and passed as parameters to some objective function.

A number of examples have demonstrated the usefulness of these methods in determining whether particular coding-function combinations should be easy or hard for a genetic algorithm. Additionally, a method has been demonstrated for using the Walsh techniques to design fully deceptive problems. The paper has also hinted at the generalization of these techniques, a matter to be covered more rigorously in the sequel.

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant MSM-8451610. I also acknowledge research support provided by the Digital Equipment Corporation and by Texas Instruments Incor-

```
{ Fast Walsh Transform include file }
const maxlength = 10;          { the maximum bit string length }
      maxpoints = 1023;        { 2^maxlength-1 }

type dataarray = array[0..maxpoints] of real;

var  datain, dataout:dataarray; { transform (or inverse) input, output }
     m:integer;                 { string length, 2^l                    }
     transformflag:boolean;     { if true then transform else inverse   }

procedure fwt(var datain,dataout: dataarray;
                  m             : integer;
                  transformflag : boolean);
{ Binary-Ordered Fast Walsh Transform - Cooley-Tukey Formulation }
{    Programmed by D. E. Goldberg & C. L. Bridges, Nov. 1988     }
var newalsh : dataarray;
    level, k, kplus, step, n, i, l,
    bunchstep, bunchstart, bunchstop : integer;
begin
 l := round(ln(m)/ln(2));  { m = 2^l            }
 n := m - 1;
 dataout := datain;        { initialize dataout }
 step := 1;
 bunchstep := 2;
 for level := 1 to l do begin
  bunchstart := 0;
  while (bunchstart < n) do begin
   k := bunchstart;
   bunchstop := bunchstart + step - 1;
   for i := bunchstart to bunchstop do begin
     kplus := k + step;
     newalsh[k]     := dataout[k] + dataout[kplus];
     newalsh[kplus] := dataout[k] - dataout[kplus];
     k := k + 1;
    end; { on to next bunch }
   bunchstart := bunchstart + bunchstep;
  end; { bunch loop }
  dataout := newalsh; { advance results a level }
  step := step * 2;
  bunchstep := bunchstep * 2;
 end;  { level loop }
 if transformflag then { divide by 2^l if transform f-->w }
   for i := 0 to n do dataout[i] := dataout[i]/m;
end; { fwt }
```

Figure 4: File `fwt.pas` contains the procedure `fwt`.

porated. The first draft of the FWT code was programmed by Clay Bridges, and the illustrations were prepared by Brad Korb.

## Appendix A. The Cooley-Tukey fast Walsh transform in pascal

This section presents a Pascal implementation of the fast Walsh transform (FWT). The code is very similar to the Cooley-Tukey formulation of the fast Fourier transform, except that sine and cosine multipliers are unnecessary and there is no imaginary part to be concerned with. Further information is available in papers by Kremer [10,11].

The program is contained in three files: `fwt.pas`, `fastwt.pas`, and `io.pas`. These three files are presented in figures 4, 5, and 6 respectively.

The file `fwt.pas` contains the procedure `fwt`, the fast Walsh transform

```
program fastwt;

{ Binary-Ordered Fast Walsh Transform - Cooley-Tukey Formulation }
{    Programmed by D. E. Goldberg & C. L. Bridges - Nov. 1988    }

{$I io.pas  }   { include i/o routines         }
{$I fwt.pas }   { include Fast Walsh Transform }

var  infile, outfile:text;
     infilename, outfilename, dataname:string;

procedure fwinput(var infile       : text;
                  var m            : integer;
                  var transformflag:boolean;
                  var datain       : dataarray;
                  var dataname     :string);
{ read in f or w values from infile with header }
var i : integer; ch:char;
begin
 readln(infile, dataname); { first line }
 readln(infile, m);        { second line }
 readln(infile, ch);       { third line }
 transformflag := (ch='t') or (ch='T');
 for i := 0 to m-1 do readln(infile, datain[i]); { remaining }
end;

procedure fwoutput(var outfile      :text;
                   var m            :integer;
                   var transformflag:boolean;
                   var dataout      :dataarray;
                   var dataname     :string);
{ write out f or w values to outfile with header }
var i : integer;
begin
 writeln(outfile, dataname);
 writeln(outfile, m);
 if transformflag then writeln(outfile,'F')
  else writeln(outfile,'T'); { invert transformflag in outfile }
 for i := 0 to m-1 do writeln(outfile,dataout[i])
end;

begin { main }
 writeln;
 writeln('============================================================');
 writeln('         Binary-Ordered Fast Walsh Transform ' ); writeln;
 open_input(infile,   interactive, 'Input  ', infilename);
 open_output(outfile, interactive, 'Output ', outfilename); writeln;
 fwinput(infile, m, transformflag, datain, dataname);
 fwt(datain, dataout, m, transformflag);
 fwoutput(outfile, m, transformflag, dataout, dataname);
 close(outfile);
 writeln('Data set ',dataname,' has ', m, ' values.');
 writeln('Input ',infilename, ' transformed to ',outfilename,'.');
 writeln('============================================================');
end.
```

Figure 5: File `fastwt.pas` contains the main program and data input and output routines for the program `fastwt`.

```
{ IO Routines- File opening routines }
type  query_type = (interactive,batch);

var   qflag:query_type;
      fn:string;

procedure page(var out:text);
begin write(out,chr(12)) end;

procedure open_input(var input:text; query_flag:query_type;
                     message:string; var filename:string);
begin
 if (query_flag=batch) then assign(input,filename)
  else begin
   write('Enter ',message,' filename: ');readln(filename);
   assign(input,filename);
  end;
 reset(input);
end;

procedure open_output(var output:text; query_flag:query_type;
                      message:string; var filename:string);
begin
 if (query_flag=batch) then assign(output,filename)
  else begin
   write('Enter ',message,' filename: ');readln(filename);
   assign(output,filename);
  end;
 rewrite(output);
end;
```

Figure 6: File `io.pas` contains the file utilities to open input and output files.

subroutine. The routine `fwt` takes `n` values (`n` a power of two) in the `datain` array (from 0 to `n-1`) and transforms them using the Cooley-Tukey algorithm into the `dataout` array. The Boolean flag `transformflag` determines whether the transform or its inverse is calculated as the flag is `true` or `false`.

The signal-flow graph of the algorithm is depicted in figure 7 for the algorithm when $n = 8$ (or when the string length $l = 3$). In the figure, the $l$ different levels of the calculation are apparent, as are the successive addition or subtraction of pairs of terms from the previous level.

The file `fastwt.pas` contains the main program, data input, and data output procedures. Data input and output are designed so that output from the program may be used immediately as input to the program (the inverse transform of the transform should return the original points).

The file `io.pas` contains utility routines to open input and output files. These are used in an interactive mode to obtain file names and to open files for the input device `infile` and the output device `outfile`.

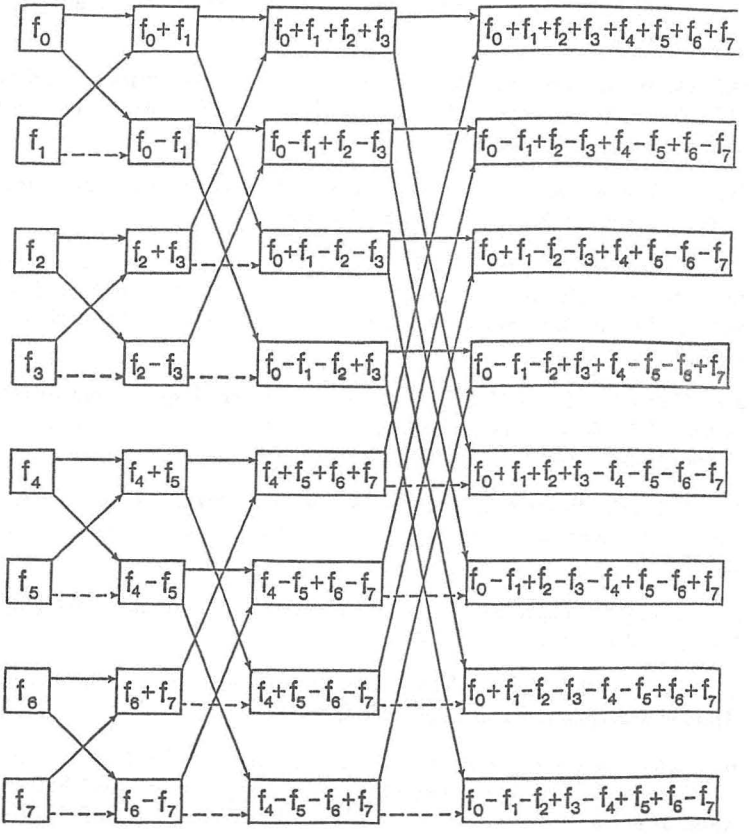


Figure 7: Signal flow graph for the Cooley-Tukey, fast Walsh transform algorithm with $l = 3$ and $n = 8$.

## References

[1] A.D. Bethke, *Genetic algorithms as function optimizers*, Doctoral dissertation, University of Michigan, 1980. *Dissertation Abstracts International*, 41(9), 3503B. (University Microfilms No. 8106101).

[2] D.E. Goldberg, "Simple genetic algorithms and the minimal deceptive problem." In L. Davis (Ed.), *Genetic algorithms and simulated annealing* (Pitman, London, 1987) 74–88.

[3] D.E. Goldberg, *Sizing populations for serial and parallel genetic algorithms* (TCGA Report No. 88004). Tuscaloosa: University of Alabama, The Clearinghouse for Genetic Algorithms, 1988.

[4] D.E. Goldberg, *Genetic algorithms in search, optimization, and machine learning*. (Addison-Wesley, Reading, MA, 1989).

[5] J.H. Holland, *Hierarchical descriptions of universal spaces and adaptive systems*. Technical Report ORA Projects 01252 and 08226, Department of Computer and Communication Services (University of Michigan, Ann Arbor, MI, 1968).

[6] J.H. Holland, "Hierarchical descriptions of universal spaces and adaptive systems." In A.W. Burks, ed., *Essays on cellular automata* (University of Illinois Press, Urbana, 1970) 320–353.

[7] J.H. Holland, "Processing and processors for schemata." In E.L. Jacks, ed., *Associative information processing* (American Elsevier, New York, 1971) 127–146.

[8] J.H. Holland, *Adaptation in natural and artificial systems* (University of Michigan Press, Ann Arbor, 1975).

[9] J.H. Holland (in press), "Searching nonlinear functions for high values," *Applied Mathematics and Computation*.

[10] H. Kremer, *Representations and mutual relations of the different systems of Walsh functions*. Paper presented at a Colloquium on the Theory and Applications of Walsh and Other Non-Sinusoidal Functions, Hatfield, Herts., UK, 1973.

[11] H. Kremer, *On theory of fast Walsh transform algorithms*. Paper presented at a Colloquium on the Theory and Applications of Walsh and Other Non-Sinusoidal Functions, Hatfield, Herts., UK, 1973.