Complex Systems **9** (1995) 11–28

Nonbinary Transforms for Genetic Algorithm Problems

Paul Field[*]
Department of Computer Science,
Queen Mary and Westfield College, University of London,
Mile End Road, London E1 4NS, England

Abstract. The Walsh transform is a powerful tool for transforming from a string-based to a schema-based perspective. It has traditionally been used in the study of deception to see fitnesses from a schema perspective but it has uses in studying the dynamics of genetic algorithms (GAs) and the structure of GA models.

In this paper, Walsh coefficients and the closely related partition coefficients are generalized to nonbinary alphabets. For both cases, the matrix form of the transform and inverse transform are presented. The relationship between the two types of coefficient is examined and, finally, an efficient algorithm for performing the nonbinary transforms is developed.

1. Introduction

Problem representation is an important issue in genetic algorithm (GA) research because the way in which a problem is represented can affect the performance of the GA. At the heart of representation is the choice of an alphabet from which the strings that the GA manipulates are built. From the perspective of GAs as "schema processors," binary strings seem best because they provide the greatest number of schemata. But this is simplistic. We at least need to ask whether the extra schemata provided by a low cardinality representation are in some sense useful. Although it is sometimes used to support the binary case, the principle of minimal alphabets [8] states "the user should select the smallest alphabet that permits a natural representation of the problem." We should consider the implications of forcing a problem to be binary if this is unnatural; we could be doing damage that outweighs the supposed benefits of extra schemata. Unfortunately, the majority of GA theory centers on binary representations and provides no help with these important questions.

[*]Electronic mail address: `paulf@dcs.qmw.ac.uk`.

Perhaps the most important tool for studying binary-represented GA problems is the Walsh transform. Generally associated with the study of deception, the Walsh transform is also a general tool for transforming from a string-based to a schema-based perspective. In particular, [10] has shown that the GA can be viewed as proceeding through nonuniform Walsh transforms and [5] has applied the incremental combination transform (described in this paper), to a complete nonbinary GA model to reveal the structure within the model's matrices. More traditionally, Walsh coefficients have been used to simplify the computation of low order schemata fitnesses. Since low order schemata survive genetic operators better than high order ones, they and their fitnesses are important.

The Walsh transform is a powerful tool but it is limited to binary representations and so has nothing to contribute to the alphabet debate; nonbinary generalizations of the transform are needed. This problem is addressed in [11] by generalizing partition coefficients to the nonbinary case. Partition coefficients are close relatives of Walsh coefficients but, until now, have not been as well understood. Walsh coefficients are the result of a change of basis and there is a fast transform algorithm available to compute them. This paper presents a generalization of the Walsh transform called the *incremental combination transform*, describes the computation of nonbinary partition coefficients using a basis transform, and develops fast algorithms for both incremental combination and nonbinary partition transforms.

To begin we discuss the intuitive meaning of the coefficients and how they are generalized to the nonbinary case. For an introduction to Walsh coefficients and their uses see [6, 7]. For a simpler and mainly nonmathematical description of Walsh and partition transforms see [4]. For a description of nonuniform transforms, which, because they operate on samples of the search space, can be applied to real-world problems, see [3]. For a more practical and nonbinary discussion see [5].

2. Generalizing binary coefficients

This section describes the intuitive meaning of binary coefficients and shows how this meaning can be used to generalize them to nonbinary coefficients. Partition coefficients are used because they have a clearer intuitive meaning than Walsh coefficients.

Partition coefficients allow us to compute a string or schema's fitness from the contributions of its alleles. For example:

$$f(11) = e_{\#\#} + e_{1\#} + e_{\#1} + e_{11},$$

where $e_{\#\#}$ is the average fitness of all strings, $e_{1\#}$ is the extra fitness that—on average—a string gets for having a 1 as its first bit, and $e_{\#1}$ is the average extra fitness that a 1 at the second locus brings. Either or both of these coefficients could be negative, in which case the 1 is, on average, pulling the fitness of the string down. e_{11} is the extra fitness that a combination of 1s brings over and above the individual contributions of the 1s. This accounts for the whole being more (or less) than the sum of the parts.

In the binary case there is a nice symmetry. If a string does not have a 1 at a particular locus then it must have a 0. This means that the benefit a string has from a 0 at, say, the first locus is the loss it suffers from not having a 1 there (i.e., $-e_{1\#}$). Here is an example:

$$f(10\#) = e_{\#\#\#} + e_{1\#\#} - e_{\#1\#} - e_{11\#}.$$

This departs slightly from the definition of partition coefficients in [11] wherein redundant coefficients are allowed which, because we produce the coefficients from a change of basis, we must temporarily ignore. We disregard any coefficient whose label contains the implied allele (defined above). Using these redundant *implied coefficients* we could write the equation as: $f(10\#) = e_{\#\#\#} + e_{1\#\#} + e^{+}_{\#0\#} + e^{+}_{10\#}$. Higher order implied coefficients seem more intuitive as they hide the puzzling (from the view of coefficients as "benefits of combinations of 1s") questions of why $e^{+}_{10\#} = -e_{11\#}$ and $e^{+}_{00\#} = e_{11\#}$.

The difference of sign between Walsh and partition coefficients in the binary case can be interpreted as a difference in which allele is implied from the absence of which. The Walsh coefficient $w_{1\#}$ can be interpreted as the benefit that a 0 at the first locus brings to a string and so $-w_{1\#}$ is the benefit for a 1.

The nonbinary case lacks the symmetry of the binary case. We must explicitly decide which allele we want to imply from the absence of the others. For notational convenience we will assume that 0 is this *implied allele*. Using the same intuitive meanings for the coefficients we can write the equations relating partition coefficients and fitnesses for a problem represented by a single cardinality 4 gene:

$$\begin{aligned}
f(0) &= e_{\#} - e_1 - e_2 - e_3 \\
f(1) &= e_{\#} + e_1 \\
f(2) &= e_{\#} + e_2 \\
f(3) &= e_{\#} + e_3.
\end{aligned} \tag{1}$$

Here, e_1 is the benefit of a 1, e_2 the benefit of a 2, e_3 the benefit of a 3, and the benefit of a 0 is the benefit of not having a 1, 2, or 3.

The generalization of the Walsh transform presented in this paper uses similarly-generated incremental combination coefficients. The equations relating fitnesses and incremental combination coefficients are the same as equations (1) except that additions and subtractions are exchanged.

We now have an informal feel for partition and incremental combination coefficients. In later sections the coefficients will be described formally by constructing general matrices to convert them to and from fitnesses. First, however, we must look at some of the background notation and assumptions behind the formal presentation.

3. Preliminaries

We assume that a GA processes strings of length l and that we have a fitness function f that maps every possible string to a positive real number. A value (or allele) at locus i on the string is drawn from a set of alleles (or gene) G_i. Although, in general, alleles could be any sort of object, their values are unimportant in this paper so we simplify matters by assuming that a gene consists of contiguous integers from 0 upwards. The cardinality of each gene is $c_i = |G_i|$. Since the number of alleles in a gene is important but their values are not, we can specify the set of strings, known as the representation of a problem, by a vector of gene cardinalities $\bar{c}$. Strictly speaking, matrices should be subscripted with the problem representation that they apply to (i.e., $\bar{f}_{\langle 2,3,2\rangle}$ or $\bar{f}_{\bar{c}}$) but sometimes subscripts will be dropped when they would clutter an equation without adding clarity.

We examine nonbinary transforms using matrix notation (i.e., $\bar{f} = V^{-1}\bar{v}$) and we need an ordering on strings and coefficients to determine their positions within their vectors. We order strings and coefficient labels so that $a < b \leftrightarrow \exists_j\, a_j < b_j \wedge \forall_{i<j}\, a_i = b_i$. We also require that $\# < \alpha$ where α is the implied allele and is less than any other allele. The ordering of nonimplied alleles is unimportant. Other orderings can be accommodated by applying permutation matrices to the transform matrices, but this would unnecessarily complicate the presentation. For the purposes of this paper we order strings by the numeric ordering of the alleles with the rightmost being in the least significant position. As an example, strings of two trinary genes: $G_1 = G_2 = \{0, 1, 2\}$ are ordered 00, 01, 02, 10, 11, 12, 20, 21, 22. We order the coefficient labels in the same way, with # being smaller than any number. For example:

$$\bar{f}_{\langle 2,3\rangle} = \begin{bmatrix} f_{00} \\ f_{01} \\ f_{02} \\ f_{10} \\ f_{11} \\ f_{12} \end{bmatrix} \quad \bar{v}_{\langle 2,3\rangle} = \begin{bmatrix} v_{\#\#} \\ v_{\#1} \\ v_{\#2} \\ v_{1\#} \\ v_{11} \\ v_{12} \end{bmatrix}.$$

4. The inverse partition transform

Section 2 describes how string fitnesses are constructed from partition coefficients, we start this section by looking at the matrix form of the inverse partition transform, which transforms partition coefficients into fitnesses. To travel the other way, fitnesses to coefficients, requires the partition transform that is described in section 5.

Obviously, the transform matrix depends on the representation of the problem. We denote the inverse partition transform matrix as E^{-1} and subscript it with the representation it applies to. The matrix form of the transform is simply a neat way of writing the equations relating fitnesses and partition coefficients. Using equations (1) as an example we have:

$$\bar{f}_{\langle 4\rangle} = E_{\langle 4\rangle}^{-1}\bar{e}_{\langle 4\rangle}$$

$$\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{bmatrix} = \begin{bmatrix} 1 & -1 & -1 & -1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} e_{\#} \\ e_1 \\ e_2 \\ e_3 \end{bmatrix}.$$

Each row of a transformation matrix corresponds to the calculation of a fitness. Any single gene transform matrix $E_{\langle c\rangle}^{-1}$ will be a $c \times c$ matrix with 1s in the first column (since every calculation requires the average fitness coefficient), 1s along the main diagonal (for the single coefficient that must be added to $e_{\#}$), and -1s from the second column onwards in the top row (subtracting the coefficients for the implied allele). Diagrammatically we have:

$$E_{\langle c\rangle}^{-1} = \left[\begin{array}{c|c} 1 & \begin{array}{c} -1 \\ \hline I_{c-1} \end{array} \end{array}\right],$$

where I_n is an $n \times n$ identity matrix.

Having dealt with the single gene case, we must move on to general representations. We will begin by looking at $E_{\langle 3,2,2\rangle}^{-1}$ and, by examining its structure, we will develop the general formula for $E_{\bar{c}}^{-1}$.

$$E_{\langle 3,2,2\rangle}^{-1} = \left[\begin{array}{cccc|cccc|cccc}
1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 \\
1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 \\
1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 \\
1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
\hline
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
\hline
1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 \\
1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 \\
1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{array}\right]$$

The matrix has been divided and shaded to show its structure. At the top level, the matrix consists of three types of identical blocks: all 0s, the light gray block, and the dark gray block. The dark gray block is simply the light gray block with all its elements negated. We could write:

$$E_{\langle 3,2,2\rangle}^{-1} = \begin{bmatrix} B & -B & -B \\ B & B & 0 \\ B & 0 & B \end{bmatrix}$$

which, structurally, looks very similar to $E_{\langle 3\rangle}$. Examining the structure of a block B in the same way:

$$B = \left[\begin{array}{cc|cc} 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \\ \hline 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \end{array}\right] = \begin{bmatrix} B_2 & -B_2 \\ B_2 & B_2 \end{bmatrix}$$

which, structurally, looks like $E^{-1}_{\langle 2\rangle}$, as does B_2 itself. What we have is the leftmost gene (cardinality 3) controlling the global structure of the matrix using $E^{-1}_{\langle 3\rangle}$ as a template into which smaller structures are placed. In the same way, the second gene controls mid-distance structure using $E^{-1}_{\langle 2\rangle}$ as a template and the rightmost gene controls local structure using $E^{-1}_{\langle 2\rangle}$.

The matrix operator called the *Kronecker product* (also known as the *direct* or *tensor product*) builds matrices in exactly this way:

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \dots & a_{1n}B \\ a_{21}B & a_{22}B & \dots & a_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \dots & a_{mn}B \end{bmatrix}$$

where A is an $m \times n$ matrix, B is $r \times s$, and $A \otimes B$ is $mr \times ns$.

The Kronecker product has the following properties (e.g., [9]):

$$(A \otimes B)^{-1} = A^{-1} \otimes B^{-1} \tag{2}$$

$$(A \otimes B)(C \otimes D) = AC \otimes BD \tag{3}$$

$$A \otimes (B \otimes C) = (A \otimes B) \otimes C \tag{4}$$

Using the Kronecker product we can write:

$$E^{-1}_{\langle 3,2,2\rangle} = E^{-1}_{\langle 3\rangle} \otimes E^{-1}_{\langle 2\rangle} \otimes E^{-1}_{\langle 2\rangle}$$

and in general:

$$E^{-1}_{\bar{c}} = E^{-1}_{\langle c_1\rangle} \otimes E^{-1}_{\langle c_2\rangle} \otimes \cdots \otimes E^{-1}_{\langle c_l\rangle}.$$

5. The partition transform

The Walsh transform matrix consists of orthogonal column vectors of equal length and so it is, bar scaling, its own inverse. Unfortunately, in general, neither the incremental combination transform nor the partition transform has orthogonal column vectors and so we have to state both the transform and the inverse transform explicitly.

We can write the partition transform matrix $E_{\bar{c}}$ as:

$$E_{\bar{c}} = \left(E^{-1}_{\bar{c}}\right)^{-1} = \left(E^{-1}_{\langle c_1\rangle} \otimes E^{-1}_{\langle c_2\rangle} \otimes \cdots \otimes E^{-1}_{\langle c_l\rangle}\right)^{-1}.$$

We can expand this using equation (2):

$$E_{\bar{c}} = E_{\langle c_1 \rangle} \otimes E_{\langle c_2 \rangle} \otimes \cdots \otimes E_{\langle c_l \rangle}$$

which simplifies our problem somewhat. We state without proof that:

$$E_{\langle c \rangle} = \begin{bmatrix} \frac{1}{c} & \frac{1}{c} & \frac{1}{c} & \cdots & \frac{1}{c} \\ -\frac{1}{c} & 1-\frac{1}{c} & -\frac{1}{c} & \cdots & -\frac{1}{c} \\ -\frac{1}{c} & -\frac{1}{c} & 1-\frac{1}{c} & \cdots & -\frac{1}{c} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\frac{1}{c} & -\frac{1}{c} & -\frac{1}{c} & \cdots & 1-\frac{1}{c} \end{bmatrix}.$$

Informally, the first row sums all the fitnesses and divides by c, the number of fitnesses. This is what we would expect for $e_{\#}$, the average fitness coefficient. All the remaining rows are the negation of row 1, except that they have a single fitness added. This is what we would expect from rearranging $f_i = e_{\#} + e_i$ to give $e_i = f_i - e_{\#}$.

6. The incremental combination transform

The nonbinary partition transform presented above is all that is needed to transform from a string-based to a schema-based perspective. In addition, partition coefficients have a clearer intuitive meaning than Walsh or incremental combination coefficients. So why do we need the incremental combination transform? Until now, partition coefficients have not been as well understood as Walsh coefficients and so Walsh coefficients have been used for nearly all transform-based GA theory. The easiest way to generalize such theory to nonbinary alphabets is to use the incremental combination transform and coefficients, which are nonbinary generalizations of the Walsh transform and coefficients. Using the nonbinary partition transform is slightly more difficult because it is not a generalization of the Walsh transform; it is only related. Section 7 will help those wishing to use the partition transform to generalize binary theory. Another reason for using the incremental combination transform is that its transform matrix is symmetric, whereas the partition transform matrix is not.

In the single gene case, the inverse incremental combination transform matrix $V^{-1}_{\langle c \rangle}$ will be $E^{-1}_{\langle c \rangle}$ with all elements except those in the first column negated. Diagrammatically we have:

$$V^{-1}_{\langle c \rangle} = \left[\begin{array}{c|c} 1 & \begin{array}{c} 1 \\ \hline -I_{c-1} \end{array} \end{array} \right]$$

This simply reflects the difference in signs between incremental combination and partition coefficients. The single gene IC transform matrix is:

$$V_{\langle c\rangle} = \begin{bmatrix} \frac{1}{c} & \frac{1}{c} & \frac{1}{c} & \cdots & \frac{1}{c} \\ \frac{1}{c} & \frac{1}{c}-1 & \frac{1}{c} & \cdots & \frac{1}{c} \\ \frac{1}{c} & \frac{1}{c} & \frac{1}{c}-1 & \cdots & \frac{1}{c} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{1}{c} & \frac{1}{c} & \frac{1}{c} & \cdots & \frac{1}{c}-1 \end{bmatrix}$$

In the same way as the partition transform, the general transform matrices are formed using the Kronecker product:

$$V_{\bar{c}} = V_{\langle c_1\rangle} \otimes V_{\langle c_2\rangle} \otimes \cdots \otimes V_{\langle c_l\rangle}$$
$$V_{\bar{c}}^{-1} = V_{\langle c_1\rangle}^{-1} \otimes V_{\langle c_2\rangle}^{-1} \otimes \cdots \otimes V_{\langle c_l\rangle}^{-1}.$$

We can easily show that this is a generalization of the Walsh transform. The Kronecker product formulation of the (inverse) Walsh (Hadamard) transform is given in [1] as:

$$W_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$
$$W_l = W_{l-1} \otimes W_1,$$

since $W_1 = V_{\langle 2\rangle}^{-1}$, both this and the incremental combination formulation reduce to the Kronecker product of l $V_{\langle 2\rangle}^{-1}$ matrices.

7. The relationship between incremental combination and partition coefficients

We have already seen that the difference between incremental combination and partition coefficients is simply one of sign. However, not all of the coefficients have different signs. We can investigate the relationship formally by finding the matrix T that transforms one set of coefficients into the other (for a particular representation $\bar{c}$). Note that T, like the transform matrices we have already seen, effects a change of basis:

$$T = VE^{-1}.$$

To find the structure of T, we expand the right-hand side using equation (3):

$$\begin{aligned} T_{\bar{c}} &= (V_{\langle c_1\rangle} \otimes V_{\langle c_2\rangle} \otimes \ldots \otimes V_{\langle c_l\rangle})(E_{\langle c_1\rangle}^{-1} \otimes E_{\langle c_2\rangle}^{-1} \otimes \ldots \otimes E_{\langle c_l\rangle}^{-1}) \\ &= V_{\langle c_1\rangle}E_{\langle c_1\rangle}^{-1} \otimes V_{\langle c_2\rangle}E_{\langle c_2\rangle}^{-1} \otimes \ldots \otimes V_{\langle c_l\rangle}E_{\langle c_l\rangle}^{-1} \\ &= T_{\langle c_1\rangle} \otimes T_{\langle c_2\rangle} \otimes \cdots \otimes T_{\langle c_l\rangle} \end{aligned}$$

where

$$T_{\langle c\rangle} = V_{\langle c\rangle}E_{\langle c\rangle}^{-1} = \begin{bmatrix} 1 & & & & 0 \\ & -1 & & & \\ & & -1 & & \\ & & & \ddots & \\ 0 & & & & -1 \end{bmatrix}.$$

Note that since $T_{\langle c\rangle}T_{\langle c\rangle} = I$, $T_{\langle c\rangle} = T_{\langle c\rangle}^{-1}$.

From T, we can generate a formula relating individual coefficients:

$$e_i = (-1)^d v_i,$$

where d is the number of defined (i.e., not #) loci in the coefficient label i.

8. Implied coefficients

Although they are redundant, it can be useful to calculate the coefficients whose labels contain the implied allele. These coefficients make it easier to see the effects of allele combinations involving the implied allele. They are also useful for calculating fitnesses by hand since they reduce the complexity of expressions and remove the problem of deciding whether to add or subtract higher-order coefficients (see section 2 for an example). However, there is no point in calculating them just for the sake of it. In the worst case (binary) there will be $3^l - 2^l$ implied coefficients in comparison to 2^l nonimplied coefficients. Schema values can be calculated without implied coefficients using the partition-schema and incremental combination schema transforms developed in [5].

It is easiest to calculate implied coefficients at the same time as the usual coefficients and to do so only involves adding an extra row to the single-gene transform matrices:

$$E^+_{\langle c\rangle} = \frac{1}{c}\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ c-1 & -1 & -1 & \dots & -1 \\ -1 & c-1 & -1 & \dots & -1 \\ -1 & -1 & c-1 & \dots & -1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & -1 & \dots & c-1 \end{bmatrix}.$$

The second row is new, it computes the coefficient e_0^+ for a single gene. The complete transform matrix E^+ can be constructed using the Kronecker product in the usual way.

Similarly, to calculate implied incremental combination coefficients at the same time as the usual coefficients, the single-gene incremental combination matrices should be altered to be of the form:

$$V^+_{\langle c\rangle} = \frac{1}{c}\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1-c & 1 & 1 & \dots & 1 \\ 1 & 1-c & 1 & \dots & 1 \\ 1 & 1 & 1-c & \dots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & \dots & 1-c \end{bmatrix}.$$

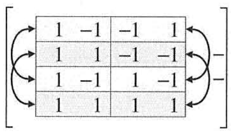
Again, the second row is new and the complete transform matrix V^+ can be constructed using the Kronecker product.

These transforms can be implemented as fast transforms using the techniques described in section 9.

9. Fast transforms

Performing an incremental combination or partition transform by naive matrix multiplication will take $O(N^2)$ operations (where N is the size of the transform matrix). Even assuming we can create the transform matrix with no time overheads we will still need $2N$ units of memory; N for the fitness vector, N for the coefficient vector. We could take advantage of 0s in the matrix but this only helps for high cardinality representations. However, there is an algorithm, analogous to the fast Walsh transform, that can perform the transforms (and their inverses) in only $O(N \log N)$ operations for the worst case (i.e., no zeros in the matrix) and can transform the data in-situ, thus needing only N units of memory.

The algorithm takes advantage of identical calculations in the transform matrix. For example, look at $E^{-1}_{\langle 2,2 \rangle}$:



$$\begin{bmatrix} 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

The arrows show identical calculations. Calculations in the right column have to be negated to become identical. Instead of $3 \times 4 = 12$ additions or subtractions, we only need two operations per column plus four operations to combine the columns, making eight operations.

These identical calculations are produced by the Kronecker product. A matrix generated by a Kronecker product is made up of identical blocks (bar scaling) so only the calculations for one block in each column of blocks need to be done. The incremental combination and partition transforms are, in general, created from many Kronecker products. We will form matrices X_i that perform only the necessary calculations for each column of blocks at each level of structure in a matrix.

The following construction of a fast algorithm is not specific to the transforms presented in this paper; it can be applied to any matrix formed by Kronecker products. This formulation has been developed independently and it is almost certainly not original; much work has been done on fast transforms (e.g., [13]). It does, however, have the virtue of being fairly simple and self-contained.

A matrix multiplication:

$$Mv = (B_1 \otimes B_2 \otimes \cdots \otimes B_n)v, \tag{5}$$

where each B_i is a $b_i \times b_i$ matrix, can be implemented as:

$$Mv = X_1(X_2(\ldots(X_n v)\ldots)), \tag{6}$$

where

$$X_i = I_{\gamma(1,i-1)} \otimes B_i \otimes I_{\gamma(i+1,n)} \tag{7}$$

$$\gamma(l,u) = \begin{cases} \prod_{i=l}^{u} b_i & l \leq u \\ 1 & l > u \end{cases} \tag{8}$$

and I_n is the $n \times n$ identity matrix.

9.1 Proof of equivalence of equations (5) and (6)

The first step is to prove by induction that

$$\prod_{i=1}^{k} X_i = B_1 \otimes B_2 \otimes \cdots \otimes B_k \otimes I_{\gamma(k+1,n)}. \tag{9}$$

The base case (when $k = 1$ is proved using equation (7):

$$\begin{aligned} X_1 &= I_{\gamma(1,0)} \otimes B_1 \otimes I_{\gamma(2,n)} \\ &= I_1 \otimes B_1 \otimes I_{\gamma(2,n)} \\ &= B_1 \otimes I_{\gamma(2,n)}. \end{aligned}$$

To prove the inductive case, equation (9) is assumed to be true for $k-1$:

$$\prod_{i=1}^{k-1} X_i = B_1 \otimes B_2 \otimes \cdots \otimes B_{k-1} \otimes I_{\gamma(k,n)}. \tag{10}$$

By definition of product, substituting in equations (10) and (7) and re-arranging using equations (3) and (4):

$$\begin{aligned} \prod_{i=1}^{k} X_i &= \left(\prod_{i=1}^{k-1} X_i\right) X_k \\ &= (B_1 \otimes B_2 \otimes \cdots \otimes B_{k-1} \otimes I_{\gamma(k,n)}) X_k \\ &= \left((B_1 \otimes B_2 \otimes \cdots \otimes B_{k-1}) \otimes I_{\gamma(k,n)}\right) (I_{\gamma(1,k-1)} \otimes (B_k \otimes I_{\gamma(k+1,n)})) \\ &= (B_1 \otimes B_2 \otimes \cdots \otimes B_{k-1}) I_{\gamma(1,k-1)} \otimes I_{\gamma(k,n)} \left(B_k \otimes I_{\gamma(k+1,n)}\right) \\ &= (B_1 \otimes B_2 \otimes \cdots \otimes B_{k-1}) \otimes \left(B_k \otimes I_{\gamma(k+1,n)}\right) \\ &= B_1 \otimes B_2 \otimes \cdots \otimes B_{k-1} \otimes B_k \otimes I_{\gamma(k+1,n)} \end{aligned}$$

which proves equation (9). From this we see that

$$\begin{aligned} \prod_{i=1}^{n} X_i &= B_1 \otimes B_2 \otimes \cdots \otimes B_n \otimes I_{\gamma(n+1,n)} \\ &= B_1 \otimes B_2 \otimes \cdots \otimes B_n \otimes I_1 \\ &= B_1 \otimes B_2 \otimes \cdots \otimes B_n \\ &= M. \end{aligned}$$

9.2 Operation counts for Mv and $X_1(X_2(\ldots(X_n v)\ldots))$

A matrix multiplication Nv involves multiplication and addition operations. A clever multiplication will avoid these operations for 0 elements in N and may be able to avoid multiplications for 1 and -1 elements too. Such a clever multiplication could potentially take advantage of 0, 1, or -1 in v but we assume that v is arbitrary and so is unlikely to contain these elements. To take advantage of v would also require dynamic run-time analysis, whereas our *a priori* knowledge of N allows a static examination. For simplicity, we will only count additions and subtractions but we will take advantage of 0 elements.

The number of operations (i.e., additions and subtractions) required to perform Nv can be calculated from the number of nonzero elements in N:

$$\mathrm{ops}(N) = d_N - r_N$$

where d_N is the number of nonzero elements in N and r_N is the number of nonzero rows. We subtract the number of nonzero rows because summing n elements only requires $n-1$ additions. To calculate $\mathrm{ops}(M)$, we need to know how nonzero elements and rows are combined by the Kronecker product. If $C = A \otimes B$:

$$\begin{aligned} d_C &= d_A d_B \\ r_C &= r_A r_B. \end{aligned}$$

This gives a straightforward equation for $\mathrm{ops}(M)$:

$$\mathrm{ops}(M) = \prod_{i=1}^{n} d_{B_i} - \prod_{i=1}^{n} r_{B_i}. \tag{11}$$

To work out $\mathrm{ops}(\prod X_i)$, we calculate $\mathrm{ops}(X_i)$:

$$\begin{aligned} d_{X_i} &= \gamma(1, i-1)\gamma(i+1, n) d_{B_i} = \frac{d_{B_i}}{b_i} \prod_{j=1}^{n} b_j \\ r_{X_i} &= \frac{r_{B_i}}{b_i} \prod_{j=1}^{n} b_j \\ \mathrm{ops}(X_i) &= \frac{d_{B_i} - r_{B_i}}{b_i} \prod_{j=1}^{n} b_j \end{aligned}$$

and then sum all the $\mathrm{ops}(X_i)$:

$$\mathrm{ops}(\prod X_i) = \sum_{i=1}^{n} \mathrm{ops}(X_i) = \left(\prod_{j=1}^{n} b_j\right) \sum_{i=1}^{n} \frac{d_{B_i} - r_{B_i}}{b_i}. \tag{12}$$

Table 1: How the incremental combination and partition transforms fit into the general fast transform formulation.

General formulation	IC transform	Partition transform
M	$V_{\bar{c}}$	$E_{\bar{c}}$
v	$\bar{v}$	$\bar{e}$
$B_1, B_2, \dots B_n$	$V_{\langle c_1 \rangle}, V_{\langle c_2 \rangle}, \dots V_{\langle c_l \rangle}$	$E_{\langle c_1 \rangle}, E_{\langle c_2 \rangle}, \dots E_{\langle c_l \rangle}$
b_i	c_i	c_i

10. The time complexity of the fast nonbinary transforms

The equations presented in section 9 are applicable to any Kronecker product formed matrix M. This section interprets them for the incremental combination and partition transforms listed in Table 1. Mappings similar to Table 1 apply for the inverse transforms.

It is important to calculate the number of operations needed for the transforms (and inverses) to predict how fast the fast transforms really are. The calculation is easiest for the inverse transforms, so we will start with those.

Remembering that the size of the transform matrix is $N = \prod_{i=1}^{l} c_i$ (the product of the gene cardinalities), and noting that $\hat{c} = \sqrt[l]{N}$ (i.e., the geometric mean of the cardinalities), we can calculate the number of operations for direct multiplication by the transform matrix from equation (11):

$$\begin{aligned} d_{E^{-1}_{\langle c \rangle}} &= c + 2(c-1) = 3c - 2 \\ r_{E^{-1}_{\langle c \rangle}} &= c \end{aligned}$$

$$\operatorname{ops}(E^{-1}_{\bar{c}}) = \operatorname{ops}(V^{-1}_{\bar{c}}) = \prod_{i=1}^{l}(3c_i - 2) - \prod_{i=1}^{l} c_i \approx O(3^{\log_{\hat{c}} N} N).$$

In contrast, the fast formulation using equation (12) gives:

$$\begin{aligned} \operatorname{ops}(\textstyle\prod X_i) &= \left(\prod_{j=1}^{l} c_j\right) \sum_{i=1}^{l} \frac{3c_i - 2 - c_i}{c_i} \\ &= 2N \sum_{i=1}^{l} \left(1 - \frac{1}{c_i}\right) \\ &= 2\left(1 - \frac{1}{\tilde{c}}\right) N \log_{\hat{c}} N \\ &= O(N \log_{\hat{c}} N) \end{aligned}$$

where $\tilde{c}$ is the harmonic mean of the gene cardinalities: $\tilde{c} = l / \sum_{i=1}^{l} \frac{1}{c_i}$.

We can obtain the same time complexities for the noninverse transforms by noticing that:

$$E_{\langle c\rangle} = E'_{\langle c\rangle} E''_{\langle c\rangle} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ -1 & 1 & 0 & \dots & 0 \\ -1 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1 & 0 & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{c} & \frac{1}{c} & \frac{1}{c} & \dots & \frac{1}{c} \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

and because

$$\begin{aligned} d_{E'} &= d_{E''} = 2c - 1 \\ r_{E'} &= r_{E''} = c \end{aligned}$$

we can calculate that $\mathrm{ops}(E^{-1}_{\langle c\rangle}) = d_{E'} - r_{E'} + d_{E''} - r_{E''} = 2c - 2$. This is the same as for E^{-1} (although we have a division then as well) and so, informally, we can see that the inverse transform is as easy to compute as the transform. There is an equivalent breakdown for V.

11. Implementing the fast transform

To implement the fast transform, we must implement multiplication by an X_i matrix. Although it might initially appear that we need to generate each X_i matrix, in fact they can be hard coded. This is easy to see if we look at the structure of an X_i matrix. As an example, we will look at X_2 for the inverse partition transform with a problem representation $\langle 3, 2, 2\rangle$. Its formula is:

$$X_2 = I_{\gamma(1,1)} \otimes E^{-1}_{\langle 2\rangle} \otimes I_{\gamma(3,3)} = I_3 \otimes E^{-1}_{\langle 2\rangle} \otimes I_2 \qquad (13)$$

and the matrix itself is:

$$\begin{bmatrix} 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

In equation (13), the left-hand identity matrix I_3 controls the global structure so we see three blocks along the main diagonal of the X_2 matrix (shaded very light gray). The right-hand identity matrix controls the local structure. A good way to look at this is as if the $E^{-1}_{\langle 2\rangle}$ matrix has been scaled up or exploded and copies made of it along the main diagonal (exploded $E^{-1}_{\langle 2\rangle}$ matrices are shown boxed with their elements shaded). From the formula for X_i

we would expect to see $\gamma(1, i-1)$ blocks each containing $\gamma(i+1, n)$ exploded $E_{\langle c_i \rangle}^{-1}$ matrices. The distance between elements of each exploded matrix is $\gamma(i+1, n)$.

If we have routines that can multiply matrices (such as $E_{\langle c \rangle}$, $E_{\langle c \rangle}^{-1}$, $V_{\langle c \rangle}$, and $V_{\langle c \rangle}^{-1}$) taking account of an offset and an explosion factor, then to multiply by X_i we simply call the appropriate routine repeatedly with the appropriate parameters. These parameters are easily generated "on-the-fly" so the X_i matrix does not have to be explicitly calculated or stored.

A C implementation of the incremental combination transform using these ideas is presented in Appendix A. It should be noted that this implementation performs transforms *in situ* (requiring only N units of memory), whereas the fast Walsh transform implementation published in [6] uses $2N$ units of memory.

12. Conclusion

In this paper two related nonbinary transforms generalized from the binary case have been shown. In addition to providing the foundations for generalizing binary-based GA theory, this paper has also produced a fast (binary) partition transform as a special case of the fast partition transform. This has not, to the author's knowledge, been previously published for binary problems.

Uses of the nonbinary transforms developed in this paper, including a discussion of nonuniform transforms (for which all fitnesses in the search space do not need to be known), are presented in [5]. Also in [5], the nonbinary transforms are presented as part of a complete nonbinary theoretical framework for GAs that is used to try to resolve the alphabet problem discussed in the introduction.

Appendix A. C source code for the fast incremental combination transform

Note that C arrays are indexed from 0. Following this, the X matrices are also labeled from 0. Also note that `assert` is a debugging test and is not compiled in production code (i.e., code compiled with the `NDEBUG` macro defined).

A slightly more robust implementation with data file input/output, a command line interface, and support for the partition transform is available from the author.

For large data sets, a noticeable speed gain *may* result from avoiding the `B_multiply()` function call in `fast_multiply()` by substituting the particular instance of `B_multiply()` directly into the source code.

```
typedef void (*matrix_multiply_fn)
          (unsigned n, double data[],
           unsigned dataoffset, unsigned matrix_explode_scale);
unsigned gamma(int lower, int upper, unsigned b[])
{ int i;
  unsigned total;

  total = 1;
  for (i = lower; i <= upper; i++)
  { total *= b[i];
  }
  return(total);
}

void fast_multiply(double data[], unsigned b[],
                   unsigned n, matrix_multiply_fn B_multiply)
{ int i;
  unsigned blocks, subblocks, blocksize, subblock, block;
  unsigned blockstart;

  /* Xn (which isn't used) is made of nxn blocks */
  blocks = gamma(0,n-1,b);
  blocksize = 1;

  for(i = n-1; i >= 0; i--)
  { /* Multiply by Xi */
    subblocks = blocksize;     /* Quick ways of calculating */
    blocksize *= b[i];         /* variables without using   */
    blocks /= b[i];            /*'gamma' in each loop       */

    assert(blocks    == gamma(0,  i-1, b));
    assert(subblocks == gamma(i+1,n-1, b));
    assert(blocksize == gamma(i,  n-1, b));

    for (block = 0; block < blocks; block++)
    { blockstart = block*blocksize;
      for (subblock = 0; subblock < subblocks; subblock++)
      { B_multiply(b[i], data, blockstart+subblock, subblocks);
        /* 'blockstart+subblock' because each subblock is */
        /* 'shifted' down and right by one row/column from */
        /* the previous subblock                           */
      }
    }
  }
}

void InvVc_multiply(unsigned c, double data[],
                    unsigned dataoffset,
                    unsigned matrix_explode_scale)
```

```
{ double sum;
  unsigned row, column_data_idx;

  /* Multiply one row at a time (row 0 is calculated */
  /* in sum as we go along)                          */
  sum = data[dataoffset];
  column_data_idx = dataoffset;
  for (row = 1;  row < c; row++)
  { column_data_idx += matrix_explode_scale;
    sum += data[column_data_idx];
    data[column_data_idx] = data[dataoffset] -
                            data[column_data_idx];
  }
  data[dataoffset] = sum;
}

void Vc_multiply(unsigned c, double data[], unsigned dataoffset,
                 unsigned matrix_explode_scale)
{ double row0_sum;
  unsigned col, row, column_data_idx;
  row0_sum = 0;
  column_data_idx = dataoffset;

  /* Multiply row 0 */
  for (col = 0; col < c; col++)
  { row0_sum += data[column_data_idx];
    column_data_idx += matrix_explode_scale;
  }
  row0_sum /= c;
  data[dataoffset] = row0_sum;
  /* Multiply remaining rows */
  column_data_idx = dataoffset;
  for (row = 1; row < c; row++)
  { column_data_idx += matrix_explode_scale;
    data[column_data_idx] = row0_sum -
                            data[column_data_idx];
  }
}

void IC_transform(double data[],
                  unsigned cardinalities[],
                  unsigned genes, bool inverse)
{ fast_multiply(data, cardinalities, genes,
                inverse ? InvVc_multiply : Vc_multiply);
}
```

References

[1] Xiaoyun Qi, Abdollah Homaifar, and John Fost, "Analysis and Design of a General GA Deceptive Problem," in Belew and Booker [2].

[2] *Proceedings of the Fourth International Conference on Genetic Algorithms*, Richard K. Belew and Lashon B. Booker editors, University of California, San Diego (Morgan Kaufmann, San Mateo, 1991).

[3] Clayton L. Bridges and David E. Goldberg, "The Nonuniform Walsh-schema Transform," in Rawlins [12].

[4] Paul Field, "Walsh and Partition Functions Made Easy," presented at the AISB 1994 Workshop on Evolutionary Computing, University of Leeds, England.

[5] Paul Field, "A Multary Theory for Genetic Algorithms: Unifying Binary and Nonbinary Representations," PhD thesis, Queen Mary and Westfield College, London, 1995.

[6] David E. Goldberg, "Genetic Algorithms and Walsh functions: Part I: A Gentle Introduction," *Complex Systems*, **3** (1989) 129–152.

[7] David E. Goldberg, "Genetic Algorithms and Walsh Functions: Part II, Deception and its Analysis," *Complex Systems*, **3** (1989) 153–171.

[8] David E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning* (Addison-Wesley, Reading, 1989).

[9] A. Graham, *Kronecker Products and Matrix Calculus with Applications* (Ellis Horwood, Chichester, 1981).

[10] Gunar E. Liepins and Michael D. Vose, "Deceptiveness and Genetic Algorithm Dynamics," in Rawlins [12].

[11] Andrew J. Mason, "Partition coefficients, Static Deception, and Deceptive Problems for Nonbinary Alphabets," in Belew and Booker [2].

[12] Gregory J.E. Rawlins (editor), *Foundations of Genetic Algorithms* (Morgan Kaufmann, San Mateo, 1991).

[13] Douglas F. Elliott and K. Ramamohan Rao, *Fast Transforms: Algorithms, Analyses, Applications* (Academic Press, New York, 1981).