

# Complexity Reduction using Expansive Coding

David Beasley<sup>1</sup>, David R. Bull<sup>2</sup> and Ralph R. Martin<sup>1</sup>

<sup>1</sup> Department of Computing Mathematics,  
University of Wales College of Cardiff, Cardiff, CF2 4YN, UK.

<sup>2</sup> Department of Electrical and Electronic Engineering,  
University of Bristol, Bristol, BS8 1TR, UK.

**Abstract.** This paper describes a new technique for reducing the complexity of algorithms, such as those used in digital signal processing, using a genetic algorithm (GA). The method, referred to as *expansive coding*, is a representation methodology which makes complicated combinatorial optimisation tasks easier to solve for a GA. Using this technique, the representation, operators and fitness function used by the GA become more complicated, but the search space becomes less epistatic, and therefore easier for the GA to tackle. This reduction in epistasis (interaction between parameters) is essential if the difficult task of complexity reduction is to be successfully achieved. Expansive coding *spreads* the task's complexity more evenly among the operators, fitness function and search space. We demonstrate how this technique can be applied to two cases of reduction of complexity of algorithms: a multiplier for quaternion numbers, and a Walsh transform computation. We suggest why the technique is more successful on the former task than the latter.

## 1 Introduction

Expansive coding is a new representation technique which we have devised to help solve complexity reduction tasks using a genetic algorithm (GA).

There have been several applications of GAs to the design of signal processing algorithms [4, 8, 13, 15]. These have mostly concentrated on the design of digital filters to satisfy particular frequency response templates. In some cases, the GA evolves a set of coefficients for a filter of fixed topology, while in other cases the GA must determine the complete configuration of the filter (i.e. coefficients and topology).

The topic of the work presented here is a variation on the conventional algorithm design task. Conventionally, the function an algorithm must perform is specified, and the task of the GA is to find a suitable configuration of processing elements which realises this function. Our research has concentrated on *algorithm simplification*. In this, we assume that both the function *and* a suitable configuration (perhaps an "obvious" one) are known. The task of the GA is to find *alternative* configurations which perform the same function, but are of lower complexity. Lower complexity may be measured in any desired way, for example: time complexity, component count, total area required, or a combination of these.

The ease with which a GA can solve such a task depends critically on the *representation* used. The most obvious, direct representations suffer from a high degree of interaction among the parameters (or *genes*). This interaction, known as *epistasis*, presents difficulty for any search algorithm, including GAs [5]. Our expansive coding technique introduces some isolation between the different parts of the task to be performed, and thereby reduces the interaction between them. By lowering the epistasis in this way, the task is made more tractable for a GA.

This paper describes the application of the expansive coding method to selected tasks in algorithm optimisation. We first describe the task domain, and then explain the expansive coding technique itself. We then show how the technique has been applied to two practical tasks: simplifying an algorithm to perform quaternion multiplication, and the derivation of the fast Walsh transform. Finally we give our conclusions.

## 2 The Task Domain

Many algorithms common in digital signal processing take the form of Eqn. 1 where, for a set of  $N$  inputs,  $x_1$  to  $x_N$ , and a set of  $M$  outputs,  $y_1$  to  $y_M$ , the output vector,  $\mathbf{y} = (y_1 \dots y_M)$ , is generated by the product of a coefficient matrix,  $\mathbf{a}$ , with the input vector  $\mathbf{x}$ .

$$y_k = \sum_{j=1}^N a_{jk} x_j \quad (1)$$

The direct implementation of such an algorithm requires  $N \times M$  multiplications, and  $(N - 1) \times M$  additions. In many instances, however, elements of the coefficient matrix,  $\mathbf{a}$ , may be related in some way. In this case, some equivalent computations may be duplicated, giving scope for algorithm simplification. For example, the multiplication of two complex numbers,  $(g + ih)$  and  $(x + iy)$ , simplistically requires four real number multiplications to form the real and imaginary coefficients of the result,  $[(gx - hy) + i(gy + hx)]$ . However, this may be achieved with only *three* real multiplications [12] by computing the product as, for example,  $[(gx - hy) + i((g + h)(x + y) - gx - hy)]$ .

Here, the four multiplications  $\{gx, hy, gy, hx\}$  are replaced by the three multiplications  $\{gx, hy, (g + h)(x + y)\}$ . In most cases, however, finding an improved computational arrangement is non-trivial. It is difficult because of the high degree of interaction between different elements of the coefficient matrix (i.e. if one element is changed, then several other elements may also need changing in order to compensate).

## 3 Expansive Coding

Expansive coding [2] involves splitting a large, epistatic, task into a number of *sub-tasks*, so that even though high epistasis may remain *within* each sub-task,

epistasis *between* sub-tasks is lower. Because the regions of high epistasis are localised, they are easier to deal with.

The steps involved in designing the representation and the GA can be described as follows:

- **Splitting.** The task is split into sub-tasks, so that any combination of valid sub-solutions gives a valid overall solution. Sub-task representations are concatenated together to form a chromosome.
- **Local constraints.** These must be placed on each sub-task, to ensure that the sub-solutions represented are always valid. They can be enforced by using, instead of conventional crossover and mutation, task-specific operators which always maintain the validity of a sub-task.
- **Merging algorithm.** The fitness is calculated by attempting to merge the sub-solutions into a global solution. Methods for doing this will be task-specific. The more merging that is successfully carried out, the fewer distinct sub-solutions will remain, and the higher the fitness.

The GA is left with the greatly simplified task of juxtaposing relatively weakly-interacting sub-solutions to find the overall solution of highest fitness. The search is simplified because the search space is restricted to the space of *valid* algorithms.

To prevent *crossover* from disrupting the validity of any sub-solution, it will often be necessary to restrict crossing sites to lie between sub-tasks. This means that the effective number of symbols in the chromosome will be equal to the number of sub-tasks. Similarly, *mutation* operators will normally work on one sub-task at a time.

## 4 The Quaternion Multiplier Task

Quaternions [11, 14] have *four* components, and may be written as  $q \equiv a + ib + jc + kd$ , where  $a, b, c$  and  $d$  are real numbers, and  $i, j$  and  $k$  are the three *quaternion operators*. Each of these is analogous to the complex number operator,  $i$ . If two quaternions,  $q_1 \equiv (a + ib + jc + kd)$  and  $q_2 \equiv (p + iq + jr + ks)$  are multiplied to give  $(w + ix + jy + kz)$ , then the components to be computed are:

$$w = (ap - bq - cr - ds), \quad x = (aq + bp + cs - dr), \quad (2)$$

$$y = (ar - bs + cp + dq), \quad z = (as + br - cq + dp). \quad (3)$$

Quaternion multiplication may be viewed as 16 input–output mappings. For example, the mapping  $a \rightarrow w$ , is achieved by multiplying by the value  $p$ . This can be represented as a linear signal flow graph, as shown in Fig.1.

This method requires sixteen real-number multiplications. Our task is to devise an algorithm which uses fewer than this.

In Fig. 1, the result of each multiplication is used only once. However, because of redundancy, it is possible to *reuse* results, allowing inputs and/or outputs to *share* multiplications. A generic circuit arrangement for this is shown in Fig. 2.

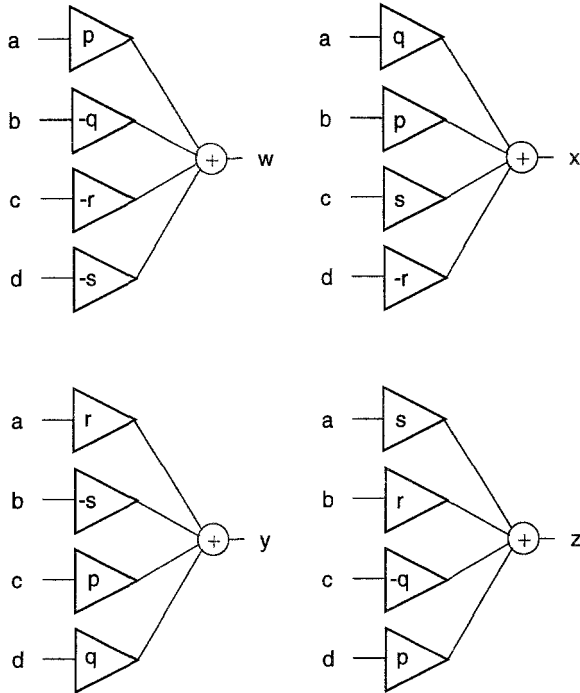


Fig. 1. Simple quaternion multiplication

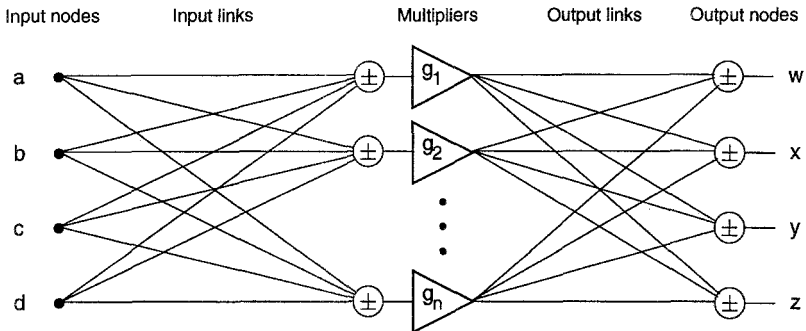


Fig. 2. Generic circuit for quaternion multiplier

Here there are  $n$  multipliers, which multiply by factors  $g_1, g_2, g_3 \dots g_n$ . Each of these gains,  $g_i$ , can be specified by four coefficients,  $h_p(g_i), h_q(g_i), h_r(g_i)$  and  $h_s(g_i)$ , such that  $g_i = h_p(g_i)p + h_q(g_i)q + h_r(g_i)r + h_s(g_i)s$ . Each of the input nodes is connected via a set of *input links* to the input of each multiplier. Each input link represents a *potential* connection between an input node, and an adder/subtractor node at the input to a multiplier. Each input link therefore represents a connection with a gain in  $\{-1, 0, 1\}$ . Similarly, the output of each multiplier is connected to each output node via an *output link*, with a gain also in  $\{-1, 0, 1\}$ .

With this arrangement, it is possible for several inputs to share a common multiplier, and also for the output of one multiplier to be shared among several outputs. By a suitable choice of input and output link gains, and multiplier gains, it is possible to represent a broad class of input/output transfer functions, a subset of which will correctly perform quaternion multiplication. The lower the value of  $n$ , the higher the fitness of the circuit. The question for the GA to answer is: What is the minimum value of  $n$ , and what gain values are required to achieve this?

Clearly, by analogy with complex multiplication, it is possible to perform quaternion multiplication using only 12 multipliers. If the GA can find this solution, it will at least demonstrate its competence; if it can improve on this solution, it will demonstrate its usefulness.

#### 4.1 Applying a Genetic Algorithm

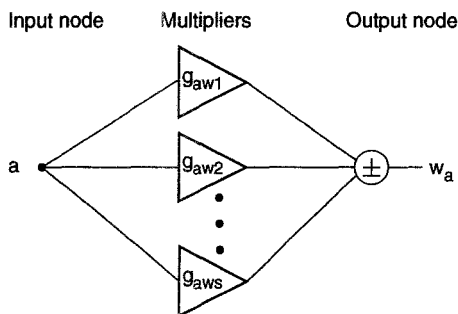
If we were to use a direct coding scheme, in which values for input link gains, output link gains, and multiplier gains were all simply coded into the chromosome, there would be a very high degree of epistasis. Changing any multiplier gain value can, potentially, alter *all* of the input/output transfer functions. Similarly, changing just one link gain value can make a valid solution invalid. So interdependent are the gains, it is impossible to improve a reasonably good chromosome by making small changes to it. No building blocks could ever form, since the fitness of any sub-group of genes is highly dependent on the values of most of the other genes. Expansive coding can be used to overcome these problems, as detailed below.

**Splitting.** The given task is split into sixteen sub-tasks, one for each input/output transfer function. Each sub-task has  $S$  multipliers of its own with which to fulfil the correct transfer function. Within the representation, these multipliers are *not* shared with any other sub-tasks. For example, the  $a \rightarrow w$  transfer function, as shown in Fig. 3, has multipliers with gains  $g_{aw1}, g_{aw2}, \dots, g_{awS}$ , and the mapping is therefore:  $w_a = a \sum_{i=1}^S g_{awi}$ . The total output is given by  $w = w_a + w_b + w_c + w_d$ . The other 15 mappings are treated in the same way.

Each multiplier in Fig. 3 can be represented by four gain coefficients,  $h_p, h_q, h_r$  and  $h_s$  (Fig. 5(c)). For simplicity, we assume that each of these will be in  $\{-1, 0, 1\}$ . Input and output link gains do not need to be represented explicitly; they can be deduced during the merging phase (see *Merging Algorithm*, below).

**Local Constraints.** Having split the task into sixteen sub-tasks, we now consider what local constraints should be applied to each of these. For a sub-task mapping input  $u \in \{a, b, c, d\}$  to output  $v \in \{w, x, y, z\}$ , the transfer function is

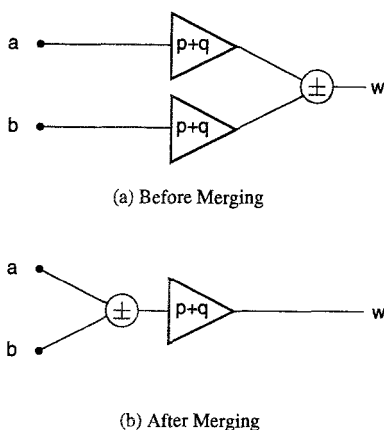
$$v_u = p \sum_{i=1}^S h_p(g_{uvi}) + q \sum_{i=1}^S h_q(g_{uvi}) + r \sum_{i=1}^S h_r(g_{uvi}) + s \sum_{i=1}^S h_s(g_{uvi}) \quad (4)$$



**Fig. 3.** Circuit for one sub-task

Eqns. (2) and (3) give the total gains required in each of the sixteen cases. For example, for the  $a \rightarrow w$  mapping, we require  $\sum h_p(g_{uvi}) = 1$ ,  $\sum h_q(g_{uvi}) = 0$ ,  $\sum h_r(g_{uvi}) = 0$ ,  $\sum h_s(g_{uvi}) = 0$ . This means that within each sub-task, the sums of the gain coefficients,  $h_p$ ,  $h_q$ ,  $h_r$  and  $h_s$ , must be maintained at specific values. To achieve this, chromosomes in the initial population are set up with valid sums, and the operators used are designed to maintain this validity (see *Operators*, below).

**Merging Algorithm.** The merging algorithm must bring together multipliers which have equal gains and compatible input/output connections. An example is shown in Fig. 4.



**Fig. 4.** Illustration of merging

In general, there will be several different ways of merging a set of multipliers, so to find the optimum merging pattern, an exhaustive search must be done. This is a slow process, but fortunately we can use an *approximate* fitness evaluation

method [9, pp138,206]. A *greedy algorithm* finds an optimal merging pattern in most cases, so our GA uses this to determine an approximate fitness for each chromosome during a run. Only when the GA has converged, and a solution found is the exhaustive search algorithm used to determine the exact fitness. After merging, the number of distinct multipliers with non-zero gain is taken as the fitness value. The GA must minimise this value.

**Two-Dimensional Chromosome Organisation.** Careful organisation of the chromosome will allow building blocks to form. A set of sub-solutions is well adapted if many of their multipliers can be merged. If they are also close together on the chromosome, they can form a building block. Merging can only take place between multipliers which share common input or output connections. So, a chromosome organisation is needed where sub-tasks are close together if they share common inputs, *or* if they share common outputs. This cannot be achieved with a conventional 1-dimensional chromosome, but is easily arranged on a 2-dimensional chromosome.

The most natural organisation is therefore a  $4 \times 4$  array of the sub-tasks, (Fig. 5(a)), where each sub-task is represented by  $S$  multipliers, (Fig. 5(b)). Each *row* of the array contains sub-tasks relating to the same output node. Conversely, each *column* contains sub-tasks relating to the same input node. Since merging can only take place between multipliers in the same row or column, it is possible for building blocks to form as coherent rows or columns evolve.

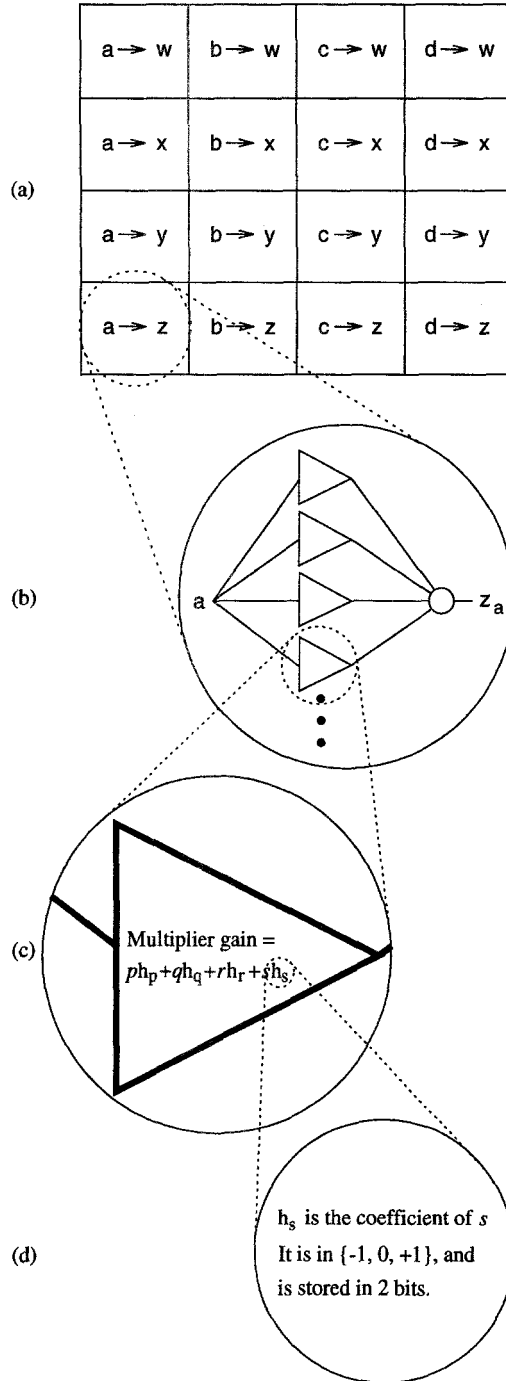
**Operators.** To avoid creating invalid chromosomes, crossover points are only allowed at sub-task boundaries. A 2-D analogy of 2-point crossover is used, with the chromosome treated as a torus.

Mutation is performed by making a small change to a coefficient value of a randomly chosen multiplier. Then, an equal and opposite change is made to the corresponding coefficient of another multiplier within the same sub-task. The effect is to maintain the sums of the gain coefficients,  $h_p, h_q, h_r$  and  $h_s$ , at their correct values.

## 4.2 Quaternion Multiplier Results

The expansive representation scheme was tested using a simple generational replacement GA, based on Goldberg's SGA [9], implemented in Pop-11 [1]. We used a form of linear fitness scaling, similar to *sigma truncation* [9, p124]. The number of reproductive opportunities for the most fit individual in each generation was 2.0. We used a crossover probability of 0.8, and mutation probability of 0.064 per sub-task. A run was terminated when the population average fitness, measured over a moving window of a fixed number of generations, stopped increasing.

We used  $S = 8$  for most runs. This gives a chromosome of 1024 bits, and a search space of approximately  $10^{150}$  valid chromosomes. At the outset we were aware of a solution to the task requiring only 12 multipliers. Our approach



**Fig. 5.** Chromosome organisation: (a)  $4 \times 4$  sub-task array; (b) a single sub-task; (c) a single multiplier; (d) a single gain coefficient.

found a solution with only 10 multipliers. (We subsequently discovered that a 10-multiplier solution is already known [7], and is the optimum within the task constraints we had used.) The results for different population sizes, averaged over 100 runs, are summarised in Table 1.

**Table 1.** Percentage of runs finding a 10-multiplier solution

Pop. size	Window size	Evaluations per run	Worst solution	% Success
100	30	16100	15 mults.	2
200	60	56600	13 mults.	33
400	100	147700	13 mults.	58

The effectiveness of the expansive coding technique is clearly demonstrated. Every run produced a solution superior to the obvious 16-multiplier arrangement. With larger populations, very good 10-multiplier solutions were found regularly. Although many of these were trivial re-arrangements of each other, our GA discovered two distinct kinds of solutions, with different structures.

## 5 The Walsh Transform Task

The Walsh transform is:

$$X_u = \frac{1}{N} \sum_{i=0}^{N-1} x_i \prod_{j=0}^{n-1} (-1)^{b_j(i)b_{n-1-j}(u)} \quad (5)$$

where  $N$  is the number of elements,  $n = \log_2 N$ , and  $b_k(z)$  is the  $k$ th bit of the binary representation of  $z$ . (Although the terms may look complicated, the  $\prod$  term expands to  $\pm 1$ , so it is quite simple in practice.) This is similar to the discrete Fourier transform, except that there are no complex roots of unity involved—elements are either added or subtracted. We were interested to see if a GA could discover for itself the Fast Walsh Transform, the equivalent of the FFT algorithm. For simplicity, we initially tried a Walsh transform with  $N = 4$ .

Each of the four outputs is just the sum of the four inputs, with some negated, and the total divided by 4 to normalise the result, as shown below (division by 4 will henceforth be ignored):

$$X_1 = (x_1 + x_2 + x_3 + x_4)/4, \quad X_2 = (x_1 + x_2 - x_3 - x_4)/4, \quad (6)$$

$$X_3 = (x_1 - x_2 + x_3 - x_4)/4, \quad X_4 = (x_1 - x_2 - x_3 + x_4)/4 \quad (7)$$

Clearly, this can be computed using 6 additions and 6 subtractions. The optimum arrangement is well known, and requires only 4 additions and 4 subtractions, as shown in Fig. 6.

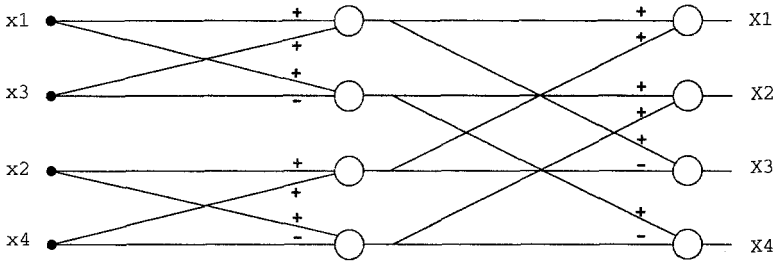


Fig. 6. Optimum solution for the Walsh transform task

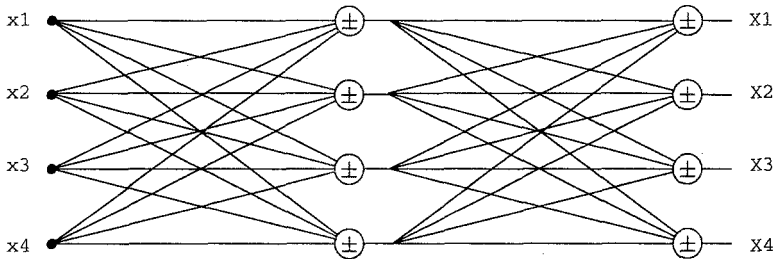


Fig. 7. Generic circuit for the Walsh transform task

A suitable generic circuit is shown in Fig. 7. Here we assume that we know that two stages of addition will be required.

As with the quaternion multiplier task, we can split this into sub-tasks, where each sub-task consists of one input-output mapping, with a number of *paths* connecting the input node to the output node. One such sub-task with four *paths* ( $S = 4$ ) is shown in Fig.8.

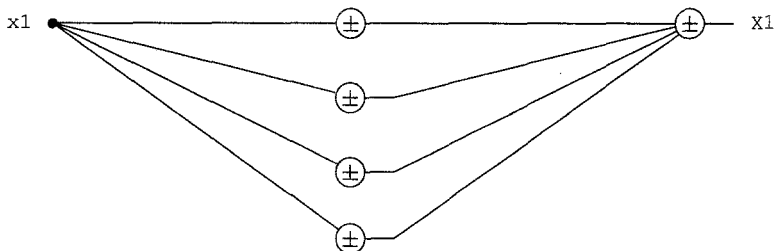


Fig. 8. One of the Walsh transform sub-tasks

Each *path* passes through two stages of addition/subtraction, each of which may effectively multiply the signal by  $+1$  or  $-1$ . Not all paths need be used. There are thus five possibilities for each path, which may be represented as:  $\{+, +\}$ ,  $\{+, -\}$ ,  $\{-, +\}$ ,  $\{-, -\}$  or “not present”. Thus each path can be repre-

sented in 3 bits, each sub-task in  $3 \times S$  bits, and the whole generic circuit of 16 sub-tasks in  $3 \times S \times 16$  bits.

### 5.1 Difficulties with Merging

In some ways this task is much simpler than the quaternion multiplier task—which involves multiplication by different coefficients. However, when the expansive coding technique is applied, this “simplicity” makes the merging phase much harder. The large number of different multiplier gain values in the quaternion task restricts the scope for the merging algorithm to a reasonable size. This makes it feasible for the merging algorithm to solve the task of finding the optimum merging pattern using either a greedy or an exhaustive search. But in the Walsh transform task, the basic elements are simply adder/subtractor units—effectively multipliers with gains of  $\pm 1$ . In this case, the scope for feasible merges is vastly increased, which means that an exhaustive search would take very much longer. Similarly, a greedy search would be most unlikely to give a good result.

At the same time, only a comparatively small chromosome (192 bits for  $S = 4$ ) is required to store the add/subtract polarity values—so the search space of the GA is not large. Yet the search space of possible ways to merge these sub-tasks is much larger. Clearly, the division of effort between the GA and the merging algorithm is out of balance—the GA has a simple task, while the merging algorithm has a difficult one. To solve this task effectively, the balance must be tipped, giving the GA more of the burden of search.

### 5.2 A Two-Chromosome Representation

The difficult task of merging the sub-tasks can be given to the GA by using a *two-chromosome* representation for each individual. The GA then has the task of searching in *two* search spaces, to find both a suitable set of path polarities, *and* an optimal way to merge them together.

The second, additional chromosome is arranged as an order-based chromosome. It holds a list of paths in the order in which they must be presented to the merging algorithm. To compute the fitness of an individual, the merging algorithm simply takes the components specified in the first chromosome, and processes them in the order specified by the second chromosome. This procedure is analogous to schedule building in a job-shop scheduling GA, where chromosomes represent the order in which jobs are placed by the schedule builder [16].

### 5.3 The Merging Process

As with the merging carried out in the quaternion multiplier task, two paths may be merged if they share common input or common output nodes, and if they have compatible polarities.

The merging algorithm takes the list of paths specified by the second chromosome, and considers the path at the head of the list for merging. An attempt is

made to merge it with the following path in the list. If this merge is not possible, then the path after that is tried. A maximum number of merge attempts, known as the *lookahead distance*, is made. If no merge is possible, then the first path is moved to the end of the list. If a merge *is* possible, the merged paths are placed at the end of the list. The new path at the head of the list is then processed. This continues until the whole list is traversed without any merges being made.

After the merging process is complete, we can compute the number of additions and subtractions required by the resultant graph. The (un)fitness of the individual is computed by forming a weighted sum. For our investigations, we used the following (arbitrary) weights: additions, 3, subtractions, 4.

## 5.4 Reproduction

Mating involves two parents, each having two chromosomes. One is a two-dimensional, value-based chromosome, the other is an order-based chromosome. The two chromosomes are crossed over and mutated quite independently. Different crossover and mutation routines are required for the two chromosomes, since they have completely different structures. There are separate crossover probability (*pcross*) and mutation probability (*pmutation*) parameters for each chromosome.

The first chromosome has a structure similar to that used by the quaternion multiplier, and the crossover and mutation operators work in the same way. For the second chromosome we use uniform order-based crossover [6]. For mutation we use a type of scramble sub-list mutation [6, p81]. In this, we choose a contiguous sub-section of the chromosome, and randomly scramble the order of its genes.

## 5.5 4-Input Walsh Transform Results

We experimented with a variety of crossover probabilities, mutation probabilities and population sizes. Like many GA researchers in the past, we found that no parameters are critical, so the GA performs well over a range of parameter values [10]. In our tests we used the performance measuring methods used for previous work [3].

With  $S = 4$  (4 paths per sub-task), and a *lookahead distance* of 2, we found that the best results were obtained with a population size of 32,  $pcross = 0.5$  and  $pmutation = 0.03$  for the first chromosome, and  $pcross = 0.5$  and  $pmutation = 0.2$  for the second chromosome. In 100 runs, we obtained an optimum solution in an average of 1942 evaluations ( $\pm 297$  with 95% confidence margin, standard deviation 1379). By comparison, a random search of the same task space found only one solution in 1 200 000 evaluations.

The results we obtained all require only one path per sub-task. Starting with 4 paths per sub-task, it is clear that the GA must do a significant amount of work simply to eliminate the redundant paths. When we ran the GA with fewer

paths per sub-task, we found that it converged much more quickly: in only 603 evaluations ( $\pm 139$ ) with  $S = 1$ .

We also tried increasing the lookahead distance to 4. This gave a further speed improvement to 242 evaluations ( $\pm 51$ ). An extension to 6 gave a smaller improvement, to 204 ( $\pm 55$ ).

Such a large speedup is worrying. The performance is now so good that solutions often appear in the initial population! It therefore seems that little credit can be given to the GA. It seems that our task representation is so attuned to the task, that little searching is required. This may mean that the GA is in a stronger position to tackle more difficult tasks of the same type.

## 5.6 8-Input Walsh Transform Results

We ran the GA on a similar Walsh Transform task, but with 8 inputs, and 64 sub-tasks. Each path has *three* stages of addition/subtraction, making the merging much more complicated. With 4 paths per sub-task ( $S = 4$ ), we found that the average fitness steadily improved during each run, eventually reaching the fitness of a typical, randomly generated individual with  $S = 1$ . But starting with  $S = 1$ , and using a variety of mutation and crossover rates, population sizes up to 512 and lookahead distances up to 8, we found that the population entirely failed to converge.

The uniform order-based crossover used for the second chromosome was found to be largely responsible. When this was replaced with PMX [9], the average fitness of the population converged, but still, little improvement in the maximum fitness of the population was observed. The “best” individuals of each run showed only trivial merges, with little evidence of a coherent structure emerging.

We also tried fixing the first chromosome of all members of the population, such that each contained an optimum set of genes throughout each run. In this case, the GA only had to search over the second chromosome. However, the performance was indistinguishable from before.

## 6 Conclusions

At first sight, expansive coding seems counter-intuitive, since it makes the search space larger. The increase in the number of parameters is, however, offset by the restriction of the search space to only *valid* chromosomes. Furthermore, the task is made simpler, since the interaction (epistasis) between the elements which the GA has to manipulate (the sub-tasks) is reduced.

With appropriate representation and operators, the inherent complexity of a task may be shifted, so that although the fitness decoding function becomes more complicated, the GA finds the task easier. In theory, this allows any task to be made trivially easy to solve, from the point of view of the GA [17]. This was the approach adopted with the quaternion multiplier task—a significant amount of the search effort was performed by the merging algorithm.

With the Walsh transform task, however, the same approach would have relied too heavily on the merging algorithm, and not enough on the GA. We therefore transferred much of the effort involved in the merging process *back* to the GA. By using *two* chromosomes, the idea is that the GA solves two interrelated tasks (a value-based task and an order-based task) at the same time.

Our initial results with the 4-input Walsh transform task seemed to show that this approach had been successful. But further work shows that, primarily, all the GA is doing is eliminating the redundancy introduced by having more than one path per sub-task. Once the GA has solved this part of the task, the representation ensures that a solution is found easily.

Much the same was found with the 8-input Walsh transform task. The GA can successfully eliminate redundant multiple paths. But it has difficulty finding solutions much better than a randomly generated population given one path per sub-task as a starting point.

It therefore appears that the introduction of the second chromosome has not, in fact, enabled the GA to solve the Walsh transform task. The merging task, encoded in the second chromosome, is, *by itself*, too difficult for our simple GA to solve.

We suspect that this is because the structure of the Walsh transform task is, in certain respects, too “simple”. In the quaternion multiplier task, there are many constraints on which paths can be merged, since there are many possible multiplier gain values. This significantly reduces the area of the search space (of the merging task) which must be explored—to the extent that even an exhaustive search is feasible. However, in the Walsh task, each node is effectively a multiplier with a gain of  $\pm 1$ . The constraints on merging are therefore minimal, and almost the whole search space the merging task must be considered. This is a large search space. For the 8-input Walsh task, with 1 path per sub-task, the first chromosome has 256 bits, while the second is twice this size. Consequently, whether the merging is performed by a conventional search procedure (as in the quaternion task), or by the GA (using a second chromosome), it is a difficult task to solve.

It therefore seems that expansive coding is more useful where each sub-task is specified by a large number of bits. This allows a significant proportion of potential merges to be ruled out quickly, thereby reducing the amount of searching which must be done in the merge space. For example, in the quaternion task, each path is specified by 8 bits, whereas in the 4-input Walsh task, each path requires only 3 bits to specify it. Hence, for a randomly-distributed set of sub-task parameters, there will only be a 1 in 256 chance that two quaternion paths would have the same set of parameters, while for the Walsh task, the figure is 1 in 8. So the search space which needs to be considered while merging the quaternion task is of the order of 32 times smaller.

A further difficulty of the Walsh task is that the known, optimal solution provides such a large improvement over the simple solution—from  $O(n^2)$  to  $O(n \log n)$ . This is obtainable because of the high degree of regularity in the

task. A relatively “weak” search method, however, which knows nothing about this regularity, would be unlikely to find the optimal solution.

We are currently investigating the application of expansive coding to tasks which are more closely related to the quaternion multiplier task, yet are scalable in nature. Our initial investigations into manipulations involving symmetrical Toeplitz matrices show that simplifications, which are far from obvious, can be obtained. We are looking into ways of maintaining good performance as task size is scaled up.

GAs are good for tasks of intermediate epistasis [5]. On highly epistatic tasks, therefore, a suitable representation and operator set must be found which sufficiently reduces the epistasis. The expansive coding technique is one such approach. Complexity, in terms of epistasis in the original task, is traded for complexity in terms of an increased chromosome size, a more complicated fitness function, and the need for task-specific operators. We have therefore split one large dose of complexity into three smaller doses—making the task easier to tackle. It is important, however, to maintain a balance of complexity among these three areas. If too much complexity is transferred to the fitness function (e.g. in the merging), the GA will not be successful.

Our practical applications of this method show that it can be effective for certain types of tasks, so long as the complexity of merging does not become too great.

## References

1. A. Barrett, A. Ramsay, and A. Sloman. *POP-11: a Practical Language for Artificial Intelligence*. Ellis Horwood, Chichester, 1985.
2. D. Beasley, D.R. Bull, and R.R. Martin. Reducing epistasis in combinatorial problems by expansive coding. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 400–407. Morgan Kaufmann, 1993.
3. D. Beasley, D.R. Bull, and R.R. Martin. A sequential niche technique for multimodal function optimization. *Evolutionary Computation*, 1(2):101–125, 1993.
4. C-H.H. Chu. A genetic algorithm approach to the configuration of stack filters. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 219–224. Morgan Kaufmann, 1989.
5. Y. Davidor. Epistasis variance: Suitability of a representation to genetic algorithms. *Complex Systems*, 4:369–383, 1990.
6. L. Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.
7. H.F. de Groote. On the complexity of quaternion multiplication. *Information Processing Letters*, 3(6):177–179, 1975.
8. D.M. Etter, M.J. Hicks, and K.H. Cho. Recursive adaptive filter design using an adaptive genetic algorithm. In *IEEE Int Conf Acou Speech Sig Proc*, pages 635–638, 1982.
9. D.E. Goldberg. *Genetic Algorithms in search, optimization and machine learning*. Addison-Wesley, 1989.

10. J.J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Trans SMC*, 16:122–128, 1986.
11. W.R. Hamilton. *Elements of Quaternions*. Cambridge University Press, 1899.
12. D.H. Horrocks and D.R. Bull. The synthesis of complex signal multipliers. In V. Cappellini and A.G. Constantinides, editors, *Proc. Int. Conf. Digital Signal Processing*, pages 207–210. North-Holland, 1987.
13. S.J. Louis and G.J.E. Rawlins. Designer genetic algorithms: Genetic algorithms in structure design. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 53–60. Morgan Kaufmann, 1991.
14. R.R. Martin. Rotation by quaternions. *Mathematical Spectrum*, 17(2):42–48, 1983.
15. D. Suckley. Genetic algorithm in the design of FIR filters. *IEE Proc-G*, 138:234–238, 1991.
16. G. Syswerda. Schedule optimization using genetic algorithms. In L. Davis, editor, *Handbook of Genetic Algorithms*, chapter 21, pages 332–349. Van Nostrand Reinhold, 1991.
17. M. Vose and G. Liepins. Schema disruption. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 237–242. Morgan Kaufmann, 1991.