Changing Representations During Search: A Comparative Study of Delta Coding

Keith E. Mathias
Department of Computer Science
Colorado State University
Fort Collins, CO 80523
mathiask@cs.colostate.edu

L. Darrell Whitley
Department of Computer Science
Colorado State University
Fort Collins, CO 80523
whitley@cs.colostate.edu

Abstract

Delta coding is an iterative genetic search strategy that dynamically changes the representation of the search space in an attempt to exploit different problem representations. Delta coding sustains search by reinitializing the population at each iteration of search. This helps to avoid the asymptotic performance typically observed in genetic search as the population becomes more homogeneous. Here, the optimization ability of delta coding is empirically compared against CHC, ESGA, GENITOR, and random mutation hill-climbing (RMHC) on a suite of well-known test functions with and without Gray coding. Issues concerning the effects of Gray coding on these test functions are addressed.

Keywords

Delta coding, Gray coding, test suites, CHC, ESGA, GENITOR, hill-climbing

1. Introduction

Genetic algorithms, like other search methods, are very sensitive to the representation of the problem. The choice of representation can determine whether a particular search method will succeed or fail. Algorithms that use different representations may also display differences in performance.

In this paper we explore the idea of searching a problem space while dynamically changing the problem representation. This notion is closely related to the use of different operators during search. For example, recombination on a four-character alphabet representation is identical to using a two-character bit representation with a special recombination operator that restricts recombination to occur only at every other possible recombination site. Just as different problems might be more easily solved using a different operator, different problem representations will also change the difficulty of a search problem. There has been some interest in the adaptive use of recombination operators and mutation operators for genetic algorithms (Davis, 1989; Schaffer & Morishima, 1987, 1988; Starkweather, Whitley, & Mathias, 1990) and, to an even greater extent, in evolution strategies. The adaptation of standard deviations for mutation rates has long been used in the evolution strategies community (Schwefel, 1981), as has been the use of correlated mutation (Schwefel, 1987; Hoffmeister & Bäck, 1991); these adaptations allow evolution strategies to learn a preferred direction of search.

In this paper we do not attempt to learn which representation of the space is best; rather we dynamically switch the representation periodically to avoid the biases associated with

any one particular representation of the space. If some representations pose an "easier" search problem for a genetic algorithm, we can perhaps exploit these representations. Our remapping strategy also attempts to focus search around good solutions that have already been discovered.

The delta coding algorithm begins with the initial run of a genetic algorithm using a binary problem encoding. When the population diversity has been adequately exploited the best solution parameters are saved as the *interim solution*. The genetic algorithm is then restarted with a new random population. The parameter substrings in subsequent runs of the genetic algorithm are decoded such that they represent a distance or *delta value* ($\pm\delta$) away from the interim solution parameters. The delta values are combined with the interim solution so that the resulting parameters are evaluated using the same fitness function. This forms a new hypercube with the interim solution at its origin. In this way, delta coding produces a new mapping of the search space at each iteration. This allows the delta coding algorithm to explore other problem representations corresponding to different versions of the search space that may be "easier" with respect to genetic search (Mathias & Whitley, 1993).

Most genetic algorithms attempt to solve a particular problem by exploiting the diversity in a fixed-size population. During the early stages of the search when population diversity is high, the genetic algorithm makes rapid improvement in the solutions that are located. However, the degree of progress toward improved solutions becomes asymptotic as the population exhibits less and less diversity (Goldberg, 1989b). *Delta coding* (Whitley, Mathias, & Fitzhorn, 1991), an iterative genetic search strategy, sustains search by periodically reinitializing the population, thereby avoiding this phenomenon.

Delta coding also provides a mechanism that reduces and enlarges the size of the hypercube currently being searched. The reduction mechanism allows the algorithm to focus the search in subpartitions of hyperspace that appear promising. The expansion mechanism allows the algorithm to reexplore previously overlooked portions of the search space later in the search. Insights on how these mechanisms perform with respect to hyperplane sampling are explored in this paper as well as the issues associated with remapping the search space.

Delta coding has been shown to exhibit superior performance when compared with other genetic algorithms in some environments (Whitley, Mathias, & Fitzhorn, 1991). However, a more comprehensive comparison on standard test functions is important for general evaluation. Here, we compare its performance against that of the elitist simple genetic algorithm (ESGA), and the GENITOR (Whitley & Kauth, 1988) and CHC (Eshelman, 1991) genetic algorithms using the DeJong test suite (DeJong, 1975) and the Rastrigin, Schwefel, and Griewank functions (Mühlenbein, Schomisch, & Born, 1991). A mutation-driven stochastic hill-climbing algorithm is also included in these comparisons, which has been referred to as random mutation hill-climbing (Forrest & Mitchell, 1993; Horn & Goldberg, 1995).

The results of these standardized comparisons demonstrate the effectiveness of the delta coding algorithm in optimization. Compared with the algorithms tested here, delta coding performs very well on most of the functions. The CHC algorithm exhibits the best overall performance of those genetic algorithms that employ a single mapping of the search space (i.e., ESGA and GENITOR). The performance of the delta coding and CHC algorithms in this test environment also illustrates the effectiveness of periodic population reinitialization in genetic search.

Although random bit climbers and algorithms using only mutation have been previously observed to perform well on some of the test functions employed here (Davis, 1991; Eshelman, 1993) the most surprising result of these comparisons is the superior performance

of random mutation hill-climbing (RMHC) when Gray coding is used. Many of these test functions were designed to be difficult for traditional search algorithms. However, when Gray coding is used the random mutation hill-climber consistently located the optimal solution on all but two of the test functions and exhibited the best performance on half of the test functions.

The use of Gray coding to transform functions found in standard genetic algorithm test suites has been shown to enhance the performance of genetic search in some cases (Hollstien, 1971; Caruana & Schaffer, 1988; Schaffer, Caruana, Eshelman, & Das, 1989). As a result, genetic algorithm performance comparisons are often accomplished using Gray coding (Eshelman, 1991; Gordon & Whitley, 1993; Schaffer et al., 1989; Schraudolph & Belew, 1992). However, Gray coding transforms a function so that it represents a different search space. Gray coding binary space not only eliminates Hamming cliffs but also has the potential to significantly alter the number of local optima in the search space as well as the size of the basins of attraction. Gray coding also induces a new set of hyperplane relationships, thus changing the schema competitions during genetic search.

Random mutation hill-climbing performs significantly worse when Gray coding is not used to transform the functions. In fact, without Gray coding, it consistently solves only a single test function. The CHC genetic algorithm performance is also degraded when Gray coding is not used, although it is able to consistently locate the optimal solution for all but two of the test functions. The performance of delta coding, however, was relatively unaffected and in some cases improved. When Gray coding is not used delta coding performs better than the other algorithms tested on all but one test function. This suggests that delta coding is able to adaptively exploit mappings of the search space that are "easier" for the genetic algorithm to solve.

2. The Delta Coding Algorithm

Our delta coding implementation uses GENITOR as the basic engine for genetic search. Delta coding evaluates the problem in a normal fashion on the initial iteration except that the population diversity is monitored. This is done by comparing the best and worst strings in the current *persisting* population,[1] where the *persisting* population is defined as the best $N - 1$ strings in the population structure of size N. If the two strings are identical or vary by a single bit in the least significant position of any one parameter, search is temporarily suspended and the best solution is saved as the *interim solution*. Mutation is not used in delta coding as described here; later in the paper we explore the use of delta coding with mutation.

A new population is then created randomly and GENITOR is restarted. This is the beginning of the *delta iterations*. Each parameter, when decoded for fitness evaluation during a delta iteration, is applied as a *delta value* ($\pm\delta$) to the interim solution saved from the previous iteration. This method of applying the new strings to the interim solution, in effect, searches a new hypercube with the interim solution at its origin. The process of selecting strings, applying crossover, and inserting offspring continues while the population diversity is monitored. At the end of each delta iteration, except the first, the numerical range for each parameter in the string is adjusted to allow the algorithm to search different subpartitions of hyperspace. This is done by altering the number of bits used to represent each parameter, based on the interim solution found in consecutive iterations.

1 The worst string in the population is not considered as part of the persisting population since it is always displaced with the insertion of the new offspring.

```
NORMAL GENITOR PHASE:
  While (Diversity Metric > 1)
  {
    Apply Recombination
    Evaluate Fitness and Insert Offspring
    IF (Fitness < Threshold) THEN
      HALT
  }

TRANSITION PHASE:
  Save Best Solution in Population as INTERIM SOLUTION
  Reinitialize Population
  Encode Parameters Using (X - 1) Bits, Use Extra Bit as Sign

DELTA ITERATION PHASE:  /* Apply GENITOR in Delta Mode */
  While ((Trials < MAX_TRIALS) AND (Fitness > Threshold))
  {
    While (Diversity Metric > 1)
    {
      Apply Recombination
      Decode All Parameter String Values
      Add All Decoded String Values to INTERIM SOLUTION Parameters
      Evaluate Fitness and Insert Offspring
    }
    Save Best New Solution as INTERIM SOLUTION
    Reinitialize Population
    IF (Delta Values EQ 0) THEN
      IF (Parameter Length < Original Length) THEN
        Encode Parameters Using 1 Additional Bit
    ELSE
      IF (Parameter Length > Lower Bound) THEN
        Encode Parameters Using 1 Less Bit
  }
```

Figure 1. The *delta coding* algorithm (assuming minimization).

The cycle of converging to a new interim solution, altering the parameter bit representations, and restarting the search is repeated until an acceptable solution is found or until a maximum number of trials has been executed. However, a delta iteration is always allowed to complete, even at the expense of a few additional trials. The delta coding algorithm is outlined in Figure 1.

Table 1. Delta coding numeric shift (remapping) example.

numeric parameters	0	1	2	3	4	5	6	7
binary coding	000	001	010	011	100	101	110	111
numeric shifts	0	1	2	3	−3	−2	−1	−0
simple delta coding	000	001	010	011	111	110	101	100

2.1 Subpartition Sampling and Delta Parameter Decoding

While the delta coding algorithm is conceptually simple, the delta parameter decoding mechanism can be complex. During the delta iterations, each delta parameter is decoded and applied to its corresponding interim solution parameter by the following method: if the sign bit of the delta value is zero then directly add the delta value to the interim solution parameter; if the sign bit of the delta value is one, complement all of the other bits in the parameter and add the result to the interim solution parameter. All addition is performed mod 2^b, where b is the number of bits in the original encoding for each parameter. Complementing and adding bits to the interim solution is equivalent to subtracting their numeric value from the interim solution.

Application of the delta parameters in this fashion provides a new mapping of the search space with each iteration. Each new mapping places the new interim solution at the origin of the hypercube and forms a new set of hyperplane competitions based on the numerically adjacent values reachable from that interim solution. Each mapping is a function of the interim solution and the current parameter string length.

Table 1 shows the delta coding for a three-bit space where the genetic algorithm has previously converged to an interim solution at 000. Delta coding interprets the binary string as a signed magnitude number. Note that the numeric parameter 7 is represented by 100 in the delta coding, which is adjacent in Hamming space to 000. In general, *if the interim solution is located at the origin of the hypercube before remapping*, then delta coding folds the space around the sign bit so that the complement of the interim solution in the original binary encoding is placed adjacent to the interim solution in Hamming space when the search switches to delta coding.

As described earlier, delta coding alters the length of the string representing each parameter at the end of each delta iteration, except the first. The remapping that occurs immediately after the first iteration of genetic search is unique in that the number of bits used to encode the problem is not altered. Only the mapping is changed, using the first bit in each parameter as a sign bit and the remaining bits as the delta value. This allows the same search space volume to be explored with a new mapping. Alteration of the parameter string length effectively changes the size of the hypercube. This allows the genetic algorithm to search subpartitions of the original search space. The area searched is defined as an interval around the parameters of the interim solution and is defined by the reach dictated by the parameter string length. The algorithm uses information about consecutive interim solutions to trigger operations that expand and contract the solution space that is currently being explored. This is done by varying the number of bits used to represent each parameter in one of two ways.

First, if the new best solution is different from the previous interim solution, the number of bits used to encode each delta value (parameter) is reduced by one bit. This effectively

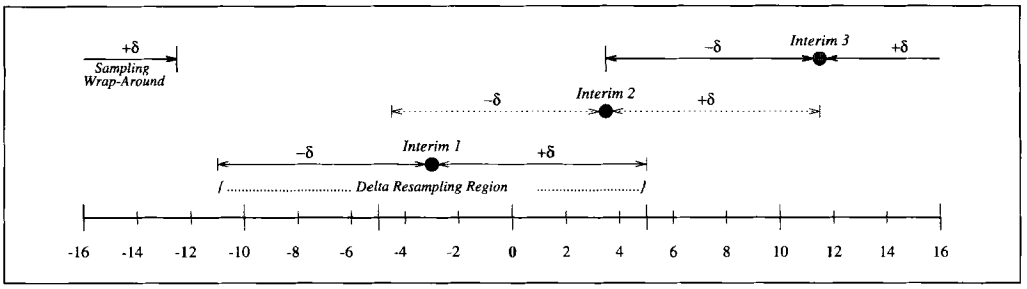

Figure 2. Points sampled in a one-dimensional numeric space using delta coding.

shrinks the hypercube. Reductions are limited to some minimal number of bits for the delta values to prevent the space being searched from becoming too small. The new interim solution need not have a higher fitness than the last interim solution, *only different.* This might seem in conflict with the elitist nature of the search. However, it is consistent with the notion that the search should not be driven by the single best solution.

The remapping of the interim solution to the string of all 0's, coupled with the application of parameters as delta values to the interim solution, determines not only how the search space is mapped but also what new subpartition of the search space is sampled. This phenomenon is of particular interest when the delta parameter encodings have been reduced so that only part of the search space is being sampled in a delta iteration.

Application of delta parameters as delta values to the interim solution guarantees that the points adjacent to the interim solution in numeric space will be contained in the subpartition that is sampled for that delta iteration (Figure 2). There is one exception to this subpartition sampling behavior. If the interim solution is located in close proximity to any boundary of the search space the sampling will be discontinued at that boundary and resumed on the opposite boundary of the hypercube extending back into the search space (see Interim 3, Figure 2).

The second event causing the number of bits in the parameter encoding to be altered occurs when a delta iteration converges to exactly the same solution (i.e., delta values of all zeros). When the genetic algorithm converges to the same interim solution, the search has effectively stalled. To prevent this, the number of bits used to encode the delta values ($\pm\delta$) is increased by one bit, expanding the hypercube. This allows the genetic algorithm to search a larger partition of the search space with a mapping that may be different from the one used previously. This would be the case if the current interim solution were different from the interim solution two or more iterations previous. The expansion mechanism is bounded by the initial parameter encoding length.

This method of controlling the size of the subpartition of hyperspace being searched differs from the Dynamic Parameter Encoding (DPE) algorithm (Schraudolph & Belew, 1992). While the DPE algorithm attempts to focus genetic search in subpartitions of hyperspace that appear more promising, it does not allow partitions of the search space that have been excluded to be reexplored later in the search. If the DPE algorithm focuses its search in a partition of hyperspace that does not contain the global optimum the search will not locate the global optimum.

Another adaptive genetic search strategy, ARGOT (Adaptive Representation Genetic Optimizer Technique) (Shaefer, 1987) also modifies the representation of the search space; however, the methods employed to change the representation are complex. Several en-

vironmentally triggered operators alter intermediate mappings that are used to map the binary strings representing the function parameters onto the search space. The intermediate mappings are based on internal measurements such as parameter convergence, parameter variance, and parameter "location" within the range of possible parameter values.

3. Advantages of Delta Coding

The delta coding algorithm has several advantages. First, it is conceptually simple to understand and implement. Second, it does not rely on disruptive mechanisms to sustain diversity during genetic search. Delta coding searches until the diversity of the current population is exhausted. It then preserves the progress made by saving the best known solution parameters at that point and reinitializes the population. This provides a new and diverse population with which to resume the search. This resurgence of diversity for each delta iteration allows the search to continue to locate new solutions, avoiding the asymptotic behavior experienced when diversity is significantly reduced.

Additionally, the criteria for adjusting the size of the hypercube imposed on the solution space are a function of the current population diversity and the previous interim solution. No complex mechanisms are required for these adjustment functions, although additional parameters are used. The use of shorter strings to perform search in a reduced hypercube allows the employment of smaller populations. A smaller population decreases the execution time for GENITOR (Starkweather et al., 1990). (While we have used smaller populations, we have not tried to change population size dynamically; this is possible, however, since string length is dynamically being altered.)

3.1 Advantages of Remapping Hyperspace

The new mapping imposed on the search space with each delta iteration may also be advantageous in that we need not locate the "easiest" mapping of the function, only a mapping that is easier than those explored earlier. Analysis of the remapping strategy in delta coding reveals that hyperspace is transformed on each iteration based on a numeric shift relative to the current interim solution. This is a result of the method in which the delta values are applied to the interim solution parameters. No attempt is made to preserve previous relationships in Hamming space.

The effects of this remapping can be viewed best by examining the fitness space of a function shown in relative Hamming order. Figure 3(a) illustrates a four-bit fully deceptive function, referred to here as function *df0*. The fitness of the strings is represented on the y-axis. All order-1, order-2 and order-3 hyperplanes lead the search toward the *deceptive attractor* (Whitley, 1991), represented by the string 0000. The global optimum is located at 1111. The strings in Figure 3(a) are organized according to Hamming distance with respect to 0000 and then ascending numeric value. This means that the strings are sorted first according to the number of 1 bits they contain, and then according to their decoded numeric value.

Applying the folding principles illustrated in Table 1 to function *df0* produces the delta coded version of the function shown in Figure 3(b). Figure 3(b) is sorted using the Hamming distance/numeric sorting described previously. While a one-dimensional ordering of strings does not represent the extent of the remapping, it does provide insight into the way the space has been remapped. There are still two local optima in Hamming space: one located at 1000 and another at 1111. However, the string 0000 no longer represents a local optimum. Most lower-order hyperplanes now lead toward the global optimum and analytical results based

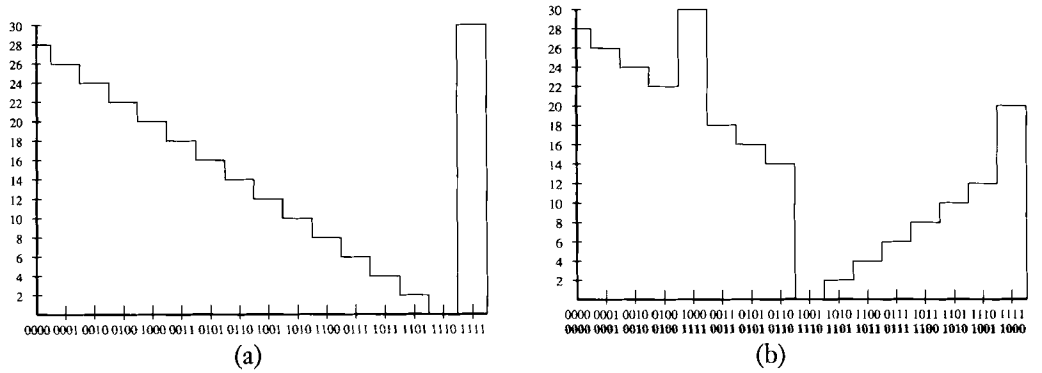

Figure 3. Remapping search space with delta coding; interim solution 0000. Figure (b) includes the delta encoding (row 1) and the original encoding (row 2).

on the work of Vose and Liepins (1991) and Whitley (1993) indicate that a traditional simple genetic algorithm will now converge on the global optimum.

3.2 Advantages of Relocating the Subpartition Sample

The reduced hypercube induced by delta coding produces a more focused search. It also introduces the possibility of excluding the global optimum from the reduced search space. However, parts of the solution space that are excluded from the reduced search space in one delta iteration can be reintroduced later in the search. Each new interim solution repositions and remaps the hypercube being searched, defining a new version of the search space. This behavior allows the algorithm to effectively "crawl" along a rough fitness surface in search of better solutions. Figure 4 shows a one-dimensional conceptual view of this behavior when minimizing a function. The sampling window has been reduced to some lower bound to avoid confusion between the movement of the sampling window and its expansion/contraction. The genetic algorithm converges on new solutions in the search space within the sampling window, allowing relocation of the hypercube to search new portions of the problem space.

The parameters in a delta iteration are decoded as delta values ($\pm\delta$) and added to the interim solution. The resultant parameter values must be contained within the original search space. Therefore, a new hypercube imposed around an interim solution close to boundaries in the solution space will wrap around on the opposite edges. This is also illustrated in Figure 4.

An additional advantage of delta coding is the ability to add bits back into the delta parameter representations. If the parameter reduction mechanism has reduced the sampling window such that the search cannot converge on a new solution, the algorithm will not be able to escape that subpartition of hyperspace. This effectively stalls the search. Adding bits back into the delta parameter representation enlarges the sampling window. This allows the algorithm to correct for aggressively focusing the search in smaller subpartitions of hyperspace (Figure 5).

4. Algorithm Descriptions and Performance Comparisons

In these experiments we have compared the performance of random mutation hill-climbing and three genetic search algorithms to that of delta coding.

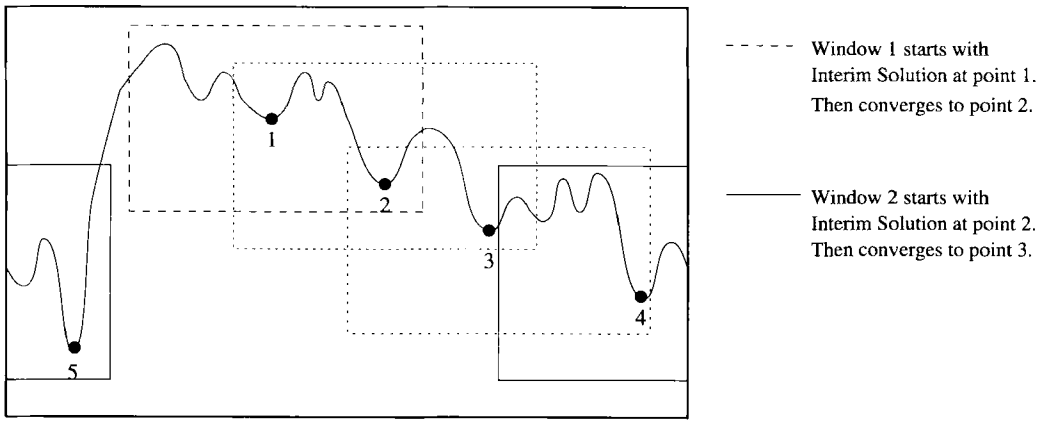

Figure 4. Crawling along a rough fitness surface. Assuming the search has converged to the interim solution at point 1, the subpartition of hyperspace defined by window 1 is searched. This allows the search to converge to point 2. The subpartition of hyperspace surrounding point 2 is then searched, converging on point 3. The search eventually converges on point 5.

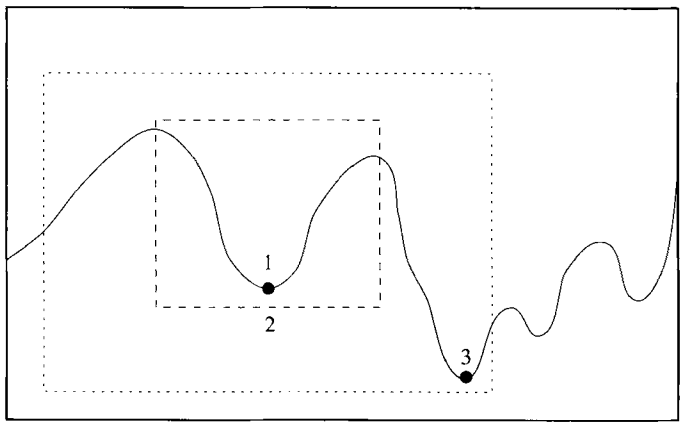

Figure 5. Expanding the delta coding search window after converging to the same interim solution on consecutive iterations using a reduced search space string representation.

4.1 Algorithm Descriptions

The random mutation hill-climber randomly generates a binary string. Then search is performed by applying mutation to that string. The resulting changes are kept whenever the fitness of the resulting string is better than or equal to the string before mutation. This process is repeated until the optimal solution is found or until some maximum number of trials is executed.

The simple genetic algorithm (SGA) (Goldberg, 1989a) that we used for comparison purposes is an elitist form of the simple genetic algorithm (ESGA) where the best individual in the population is always copied into the next generation. Selection is performed using Baker's stochastic universal sampling algorithm (Baker, 1987), and crossover is applied according to a user defined probability value (P_c).

The GENITOR algorithm (Whitley & Kauth, 1988) is not a generational algorithm. Instead GENITOR maintains a ranked population whereby individuals are selected for reproductive opportunity according to a linear bias mechanism. Those individuals with a higher fitness are selected for reproduction more often. Crossover is always applied to those individuals selected. One of the resulting offspring is chosen randomly for evaluation and insertion into the population in its ranked position. The worst individual in the population is displaced. The total reproduction process constitutes a *trial.*

The CHC Adaptive Search Algorithm (Eshelman, 1991), is a generational genetic search model. However, this algorithm differs from the standard genetic algorithm in several ways and seems to be intricately designed whereby all of its subcomponents are necessary to achieve the best performance. First of all, the CHC algorithm employs "heterogeneous recombination" as a method of "incest prevention" (Eshelman, 1991). If the number of differing bits exceeds some threshold, uniform crossover is applied to half of the differing positions, where the positions are chosen randomly. The operator designed for performing uniform crossover over half of the differing bits is HUX. HUX is a very disruptive crossover operator designed to sample new points in the search space by ensuring that parents paired for recombination have at least some appreciable differences. Additionally, the CHC algorithm does not bias the selection of strings for reproduction in favor of those strings with a higher fitness. Instead the algorithm employs a cross-generational selection/competition mechanism by randomly and uniformly choosing strings for recombination from the parent population. Offspring are held in a temporary population. Then the best N strings from the parent and offspring populations are selected to form the population for the new generation, where N is the population size. No mutation is applied during the recombination phase of the algorithm.

Like delta coding, CHC uses a form of restart to provide new diversity in the population to continue genetic search. CHC's restart mechanism is known as *cataclysmic mutation.* When no offspring can be inserted into the population of a succeeding generation, the best individual in the population is used as a template to reinitialize the population. The new population includes one copy of the template string. The remainder of the strings in the new population are formed by repeatedly mutating some percentage of bits in the template string and copying the result into the new population. This creates a population that is biased toward a good solution but with new diversity. This mechanism preserves the progress made so far and sustains further search but does not change the mapping of the strings with respect to one another in Hamming space.

4.2 The Test Suite

The DeJong test suite has become one of the standards by which the effectiveness of genetic algorithms are measured (DeJong, 1975; Goldberg, 1989a). These functions are described by the following equations:

$$F1 : f(x_i \mid_{i=1,3}) = \sum_{i=1}^{3} x_i^2 \qquad x_i \in [-5.12, 5.11],\ \Delta x_i = 0.01$$

$$F2 : f(x_i \mid_{i=1,2}) = 100(x_1^2 - x_2)^2 + (1 - x_1)^2 \qquad x_i \in [-2.048, 2.047],\ \Delta x_i = 0.001$$

$$F3 : f(x_i \mid_{i=1,5}) = \sum_{i=1}^{5} \lfloor x_i \rfloor \qquad x_i \in [-5.12, 5.11],\ \Delta x_i = 0.01$$

$$F4 : f(x_i \mid_{i=1,30}) = \left[\sum_{i=1}^{30} i x_i^4\right] + Gauss(0,1) \qquad x_i \in [-1.28, 1.27],\ \Delta x_i = 0.01$$

$$F5 : f(x_i \mid_{i=1,2}) = \left[0.002 + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^{2} (x_i - a_{ij})^6}\right]^{-1}$$

$$x_i \in [-65.536, 65.535],\ \Delta x_i = 0.001$$

The following functions have also been included in other genetic algorithm comparative works (Mühlenbein et al., 1991) and are included here as an initial step toward further evaluation and comparison of these genetic algorithms on larger and more difficult functions.

$$Rastrigin : f(x_i \mid_{i=1,n}) = (n * 10) + \left[\sum_{i=1}^{n} \left(x_i^2 - 10\cos(2\pi x_i)\right)\right]$$

$$x_i \in [-5.12, 5.11],\ \Delta x_i = 0.01, n = 20$$

$$Schwefel : f(x_i \mid_{i=1,10}) = \sum_{i=1}^{10} -x_i \sin(\sqrt{|x_i|}) \qquad x_i \in [-512, 511],\ \Delta x_i = 1.0$$

$$Griewank : f(x_i \mid_{i=1,10}) = 1 + \sum_{i=1}^{10} \frac{x_i^2}{4000} - \prod_{i=1}^{10} (\cos(x_i/\sqrt{i})) \quad x_i \in [-512, 511],\ \Delta x_i = 1.0$$

4.3 Empirical Test Design

For the following tests, each of the parameters for the algorithms tested was adjusted to improve the performance as much as possible for that algorithm on each problem. The best performance possible was interpreted as the greatest number of runs locating the optimal solution while expending the least amount of work possible (i.e., the smallest number of recombinations). Our approach to finding the best set of parameters for a particular algorithm was to tune the parameters by judgment rather than to select the parameter combinations from a predetermined list. The latter approach might cause particularly good combinations of parameter settings to be overlooked. These tests represent the exploration of hundreds of parameter set combinations.

All of the genetic algorithms tested here used two-point reduced surrogate crossover (Booker, 1987) except for the CHC algorithm. The CHC tests were performed using the HUX crossover operator as specified by the design for that algorithm (Eshelman, 1991). The mutation rate listed in these experiments is the probability that a bit is complemented.

Review of published test results for the SGA and ESGA on these functions also reveals that the offspring produced are compared to the parents and are not evaluated if there is no difference between them. Therefore, the work expended is not counted. This method of tallying recombinations is employed here only for the ESGA.

We have also found that other genetic algorithm comparisons in the literature on the same test functions may not be comparable because the binary strings were not always mapped onto the functions consistently. All of the comparisons presented here mapped the binary strings onto the function space for a given problem in the same way.

Previous genetic algorithm performance comparisons using functions included in this test suite report the use of Gray coding (Caruana & Schaffer, 1988; Schaffer et al., 1989;

Eshelman, 1991; Gordon & Whitley, 1993; Schraudolph & Belew, 1992). Gray coding is a general method for transforming one binary mapping into another such that the resulting consecutive binary representations differ in a single bit position. There are many Gray codes (Gilbert, 1958; Hamming, 1980) for a binary space. However, *the* Gray code most common in genetic algorithm testing and used for all of the experiments here is the *Binary Reflected Gray Code* (Bitner, Ehrlich, & Reingold, 1976; Hamming, 1980). Although there are several methods for generating the binary reflected Gray code (Bitner et al., 1976; Chen & Shin, 1987; Mathias & Whitley, 1994), the algorithm for this code can be represented as:

```
gray[0] = binary[0]
k = 1
WHILE (k < string_length)
{
   IF (binary[k-1] == 0) THEN  gray[k] = binary[k]
   ELSE  gray[k] = COMPLEMENT(binary[k]);
   k = k + 1
}
```

Performing genetic search on a function that has been transformed using Gray coding solves a *different mapping* of the function and incurs a time penalty for de-Graying the string. Here, we have compared the performance of algorithms on these test functions with and without Gray coding.

Finally, several earlier tests report the results for DeJong's F4 noisy function using different methods for determining when the algorithm has solved the problem to optimality. Some tests use an evaluation value of -2.0 to determine when optimality has been reached. Other tests compare the offspring to the optimal string to determine optimality but use the noisy evaluation value for genetic algorithm feedback. In the tests here we have used an evaluation value threshold of -2.5 to test for optimality for all algorithms except for the ESGA, where a value of -2.0 was used to make it easier for the ESGA to solve the problem and for historical comparison purposes.

4.4 Empirical Test Results Using Gray Coding

Table 2 lists the parameter settings that yield the best performance results for the ESGA and GENITOR algorithms for the functions tested here. *Percent solved* refers to the percentage of those runs that found the optimal solution. *Average trials* indicates the number of recombinations necessary to locate the optimal solution averaged over those runs that were successful. The *average best* and *maximum trials* values are recorded only when the optimal solution is not found 100% of the time. The average best value is the average fitness of the best individual in the population for all 30 runs after the maximum number of recombinations have been executed. Gray coding was applied to all of the test functions.

Our ESGA results differ slightly from other published results in some instances. Several differences in the tests may account for these discrepancies. First, the performance of the ESGA could sometimes be improved using parameter settings not included in other tests; in particular, the ESGA often performed best using no crossover. Second, the methods used to test for optimality for the F4 function were not consistent.

In all of the test cases where the ESGA performs better than GENITOR, no crossover is employed by the ESGA. Search is performed using only mutation. It is also interesting to note that the optimal ESGA performance on all but three of the test functions is observed when crossover is not used. The superior performance of mutation without crossover in the

Table 2. Performance comparisons for the ESGA and GENITOR using Gray coding.

Problem	F1	F2	F3	F4	F5	F6	F7	F8
Number of Parameters	3	2	5	30	2	20	10	10
Total Bits	30	24	50	240	34	200	100	100
				ESGA Algorithm				
Percent Solved	100%	100%	100%	77%	100%	100%	100%	13%
Population Size	10	10	50	50	10	10	50	50
Average Trials	703	7528	2590	28664	569	112714	86624	173009
Maximum Trials				100000				500000
Average Best				−2.11				0.075
Crossover Rate	0.00	0.00	0.60	0.80	0.00	0.00	0.60	0.90
Mutation Rate	0.02	0.02	0.01	0.005	0.02	0.005	0.01	0.001
				GENITOR Algorithm				
Percent Solved	100%	100%	100%	100%	100%	100%	100%	93%
Population Size	15	500	25	100	75	800	300	800
Average Trials	831	67461	997	18501	787	151207	16347	133733
Maximum Trials								500000
Average Best								0.002
Linear Selection Bias	1.25	1.50	1.25	1.25	2.00	1.25	1.25	1.25
Mutation Rate	0.04	0.04	0.04	0.01	0.04	0.01	0.01	0.02

ESGA has been observed previously for some of the test functions used here (Davis, 1991; Eshelman, 1991).

GENITOR consistently finds the optimal solution for seven of the eight test functions and finds the optimal solution 93% of the time for F8. GENITOR performed best when some mutation was added. The improved performance included fewer trials and a reduction in the population size needed to solve the test functions consistently. A decrease in the population size for GENITOR is significant in that less overall work is necessary to insert offspring into the population (Starkweather et al., 1990).

Table 3 shows the performance results for random mutation hill-climbing (RMHC) when Gray coding is applied to all of the test functions. Surprisingly, RMHC consistently finds the optimal solution for six of the eight test functions, performing better than the ESGA and GENITOR in four of those six cases.

Table 4 shows the performance for the CHC and delta coding algorithms. Gray coding was used for all test functions. However, Gray coding was applied only during the initial iteration for the delta coding algorithm tests. All subsequent iterations used the signed binary

Table 3. Performance of random mutation hill-climbing using Gray coding.

Problem	F1	F2	F3	F4	F5	F6	F7	F8
Percent Solved	100%	100%	100%	20%	100%	100%	100%	3%
Average Trials	507	19591	621	68851	481	104844	115267	227701
Maximum Trials				100000				500000
Average Best				−1.750				0.099
Mutation Rate	0.04	0.15	0.04	0.02	0.07	0.02	0.05	0.02

Table 4. Delta coding and CHC performance results using Gray coding.

Problem	F1	F2	F3	F4	F5	F6	F7	F8
				CHC Algorithm				
Percent Solved	100%	100%	100%	100%	100%	100%	100%	100%
Population Size	50	50	50	50	50	50	50	50
Average Trials	1126	9455	1265	18745	733	158839	9803	51015
Cataclysmic Mutation	35%	35%	35%	35%	35%	35%	35%	35%
				Delta Coding Algorithm				
Percent Solved	100%	100%	100%	100%	100%	100%	100%	100%
Population Size	15	25	15	25	25	800	100	200
Average Trials	585	3548	995	4883	490	258135	16957	53264
Linear Selection Bias	2.00	2.00	2.00	2.00	2.00	2.00	1.90	2.00
Mutation Rate	0.05	0.02	0.06	0.03	0.07	0.01	0.02	0.02

decoding described in Section 2.1. All delta coding experiments used a parameter reduction scheme where each parameter was reduced by one bit per delta iteration with a lower bound of four bits per parameter. The parameter enlargement scheme increased each parameter by one bit.

The cataclysmic mutation rate, population size, and HUX crossover threshold parameters for the CHC algorithm were set as suggested by Eshelman (1991). The CHC algorithm performed quite well using these parameters on most of the test functions, and additional tuning did not significantly impact the results. The CHC algorithm even solved the F7 and F8 functions using fewer recombinations than any of the other algorithms tested here when Gray coding was applied and was the only algorithm besides delta coding to solve all of the test functions consistently.

Table 5. Number of local optima in Hamming space.

Function	F6	F6	F7	F7	F8	F8
Dimension	1	2	1	2	1	2
Binary	19	361	12	144	18	627
Gray	5	25	5	25	22	639

Of those algorithms tested here, delta coding performed the best or second best on all functions except F6 and F7. Delta coding performed especially well on test functions F2, F4, and F8, which proved to be very difficult for most of the other algorithms tested here. Very little parameter tuning was necessary to solve all of the functions consistently. And although only the best parameter sets are reported here, it is significant to note that delta coding was able to solve most of the functions consistently over a wide range of parameter settings. Additionally, the delta coding algorithm was able to solve most of these functions consistently with smaller populations than GENITOR.

The original delta coding algorithm did not incorporate mutation since the iterative restarts inherently maintained a high level of population diversity. However, it was discovered that the addition of mutation improved the performance of delta coding in most of the test cases recorded here. In some cases, however, the improvement was minimal.

4.5 Should Gray Coding Be Used?

4.5.1 Gray Coding and Local Optima Gray coding is known to eliminate Hamming cliffs that exist in binary function spaces. A Hamming cliff occurs when two consecutive numbers have complementary binary representations. For example, the binary representations for the numbers 7 and 8 are complements of each other (i.e., 0111 and 1000). Hamming cliffs may account for the inability of optimization methods that use binary encodings to locate the optimal solution of a function. The elimination of Hamming cliffs is a result of the more general property that all Gray codes transform complementary binary strings into representations that are adjacent in Gray Hamming space. For example, the Gray coded representations for the binary strings 000000 and 111111 are 000000 and 100000. This relationship can be generalized for any complementary pair of strings in binary space by applying exclusive-or ($\oplus$) over the strings in the canonic example (Mathias & Whitley, 1993). Thus, two local optima located at complementary string positions in binary space will be in the same attraction basin in Gray coded space.

Gray coding may potentially increase or decrease the number of local optima for any function as compared with the standard binary encoding of that function. Table 5 shows the number of local optima in binary and Gray coded Hamming space for one- and two-dimensional versions of the Rastrigin (F6), Schwefel (F7), and Griewank (F8) functions using 10-bit encodings. While there are fewer local optima in Gray coded Hamming space than in binary space for F6 and F7, F8 has approximately the same number in both spaces.

Table 5 also shows how local optima grow with respect to multidimensional functions. The number of local optima in multidimensional functions that are a linear combination of a nonlinear function (i.e., F6/F7), grow according to l^d, where l is the number of local optima in the one-dimensional function and d is the dimension. Therefore, if more local

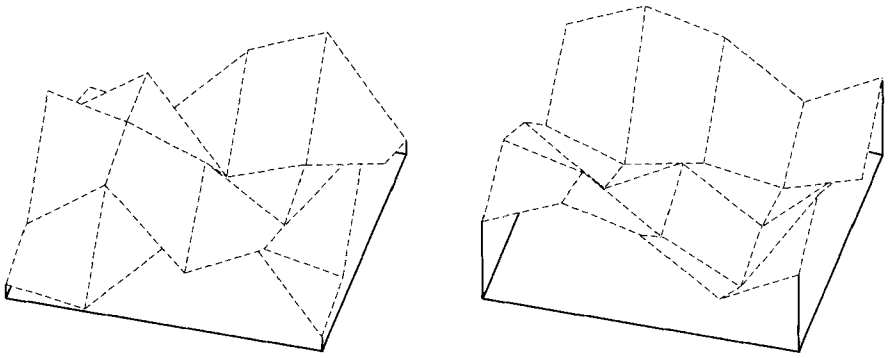

Figure 6. Three-dimensional view of binary and Gray coded Hamming space for a four-bit version of the Griewank function. The Gray coded space (b) is noticeably simpler than the binary space (a).

optima exist in binary space for the one-dimensional function than in Gray space, then as the dimensionality increases the number of local optima in binary space grow exponentially faster than in Gray space. Thus Gray coding solves a simpler function mapping. This behavior is not observed for F8, where parameter interactions are multiplicative. As the dimensionality of F8 increases the number of local optima increase more rapidly than predicted by the l^d growth expression.

Figure 6 provides a three-dimensional representation of the binary (a) and Gray coded (b) Hamming spaces for a four-bit Griewank function. Fitness is represented as elevation, and Hamming space is arranged according to the following neighborhood grid pattern:

```
1111 - 1110 - 1010 - 1011 - 1111
 |      |      |      |      |
1101 - 1100 - 1000 - 1001 - 1101
 |      |      |      |      |
0101 - 0100 - 0000 - 0001 - 0101
 |      |      |      |      |
0111 - 0110 - 0010 - 0011 - 0111
 |      |      |      |      |
1111 - 1110 - 1010 - 1011 - 1111
```

The neighbors to the north, south, east, and west of each string differ by one bit. The points along the edges are duplicated so that neighborhoods are completed with minimal wraparound. Local minima in binary Hamming space are located at the points 0001, 0010, and 1110. The global minimum is located at 1000. Local minima in Gray coded Hamming space are located at the points 0011 and 1001. The global minimum is represented by the string 1100.

4.5.2 Gray Coding and Steepest Descent Search Altering the number of local optima by application of Gray coding implies that local search might behave differently in the Gray coded version of the search space than in the standard binary version of the search space. We examined the effects of Gray coding on the search space of the same test functions using a steepest descent search. The steepest descent algorithm used here begins at a random point in Hamming space and then evaluates the fitness of its l nearest neighbors, where l is the

Table 6. Steepest descent performance. All comparisons are over 30 independent runs.

Coding	Func.(Dimension)	F1 (3)	F2 (2)	F3 (5)	F4 (30)	F5 (2)
Binary	Number Solved	30	9	30	3	30
	Average Restarts	8.3	483.7	79.7	610.0	17.4
	Average Best		0.000009		−1.68	
Gray	Number Solved	30	15	28	0	30
	Average Restarts	1	466.9	264.6		2.2
	Average Best		0.000001	−29.93	−0.744	

Coding	Func.(Dimension)	F6 (20)	F7 (10)	F7 (20)	F8 (10)	F8 (20)
Binary	Number Solved	0	0	0	2	0
	Average Restarts				468.0	
	Average Best	14.44	−4014.6	−7641.2	0.069	0.145
Gray	Number Solved	0	3	0	30	30
	Average Restarts		514.0		37.5	5.5
	Average Best	17.39	−4032.2	−7626.6		

length of the string. The algorithm moves to the neighbor with the smallest fitness value if it is less than or equal to the fitness of the current point. This process is repeated until a better solution cannot be found. If the optimum solution is not reached a new random starting point is generated and the search is continued.

Steepest descent search was applied to each of the test functions 30 independent times. A maximum of 1000 random starting points was allowed for each run. The results are shown in Table 6. *Number solved* indicates the number of independent runs in which the optimal solution was located. *Average restarts* indicates the number of random start points necessary to locate the optimum solution, averaged over those runs that were successful. *Average best* reflects the best solution found, averaged over all 30 runs.

The results in Table 6 can also be used to gain some indication as to the number of restarts necessary to find the optimal solution. This can be accomplished by entering the table values for the number solved (*Slv*), average restarts (*Avg*), number of independent runs (*N*), and maximum restarts allowed in an independent run (*Max*) into the following equation:

$$\frac{(Slv * Avg) + Max(N - Slv)}{Slv}$$

Table 6 indicates that Gray coding significantly enhances the performance of steepest descent search for most of the test functions here. Steepest descent search solves the function

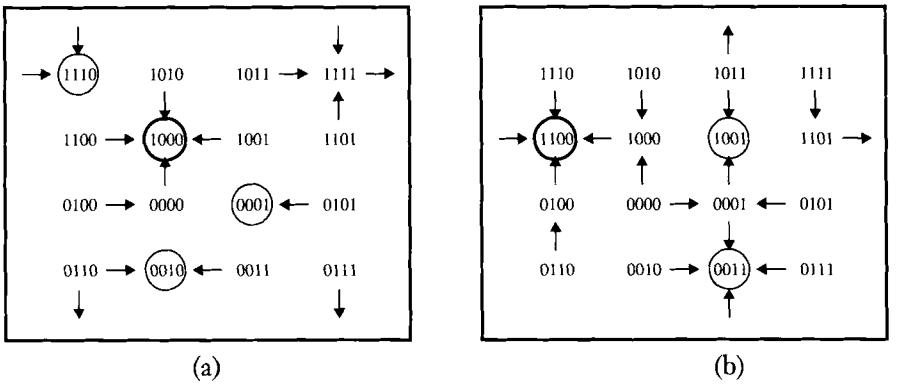

Figure 7. Binary and Gray coded Hamming space with respect to steepest ascent.

more often, consistently solves the function using fewer average restarts, or finds a better average solution when the function is Gray coded. However, this behavior is particularly surprising for F8. Based on the growth rate of local minima for F8, given in Table 5, the binary and Gray codings should prove equally difficult. *Further analysis of smaller versions of F8 suggests that the percentage of points in the attraction basin of the global optimum increases dramatically when Gray coding is applied.*

This phenomenon can be illustrated by using directional descent diagrams to represent the four-bit Griewank function shown in Figure 6. Directional descent diagrams indicate the path followed by steepest descent search with arrows. There are four attraction basins in binary Hamming space (Figure 7(a)) with optima at 0001, 0010, 1110, and 1000. There are two, three, six, and six points in each respective attraction basin. The attraction basins in Gray coded Hamming space (Figure 7(b)) whose optima are located at 0011, 1001, and 1100 have sizes seven, five, and nine, respectively. The global attraction basin in Gray space includes 43% of the points as compared to 35% in binary space. Analysis based on larger versions of the Griewank function indicate that *the size of the attraction basin containing the global optimum increases relative to other basins of attraction as the dimensionality is increased.* One implication of this phenomenon is that any study concerning the scalability of genetic algorithms should consider how scaling a test function impacts the function's topology and the size of the attraction basin containing the global optimum.

4.5.3 Gray Coding and Genetic Search Gray coding a function remaps the hyperplane relationships within the search space. This alters the schemata competitions that occur during genetic search. This is illustrated using the function given in Table 7 and the executable model of the simple genetic algorithm developed by Whitley (1993; 1994). This executable model allows us to track the representation of the strings in the population over time for a given problem. The model assumes an infinite population and does not model mutation. The model also requires that the fitness of all points in the search space be known. Each generation, it calculates the proportional representation for each string in the population, based on each string's relative fitness and current representation in the population.

Based on the executable model output for the binary search space, Figure 8(a) shows that the proportional representation of the optimal string 0000 monotonically increases over time. By generation 20 the SGA has begun to converge on the optimal string, and by generation 40 almost no other strings are represented in the population. However, the executable model

Table 7. Four-bit function.

Binary	Gray	Fitness	Binary	Gray	Fitness
0000	0000	30	0100	0110	12
0001	0001	23	0101	0111	20
0010	0011	8	0110	0101	10
0011	0010	4	0111	0100	2
1000	1100	18	1100	1010	16
1001	1101	24	1101	1011	22
1010	1111	28	1110	1001	14
1011	1110	26	1111	1000	0

output shown in Figure 8(b) shows that when the Gray coding representation of the problem is used, the SGA rapidly converges on the string 1111 (1010, binary). This behavior is due to the remapping of the original hyperplane relationships, producing schemata competitions that mislead the SGA. This example not only illustrates how Gray coding remaps hyperspace but also how Gray coding can produce a function that is more difficult for the SGA to solve.

While Gray coding usually makes this particular set of test functions easy to hill-climb, this would not necessarily be the case for arbitrarily chosen functions. The invertible transformation between Gray and binary space guarantees that any mapping between points and strings that exists in one space also exists in the other. Since the sets of mappings from points to strings are equivalent, search is not inherently easier in one space or the other.

Nevertheless, any mapping that makes a test function easy to hill-climb should not be used for genetic algorithm comparative studies. One can argue that any representation that allows the function to be solved by simple stochastic hill-climbing compromises comparative studies, since those algorithms that exploit hill-climbing have an advantage. We would also argue that genetic search is most needed on problems where random mutation hill-climbing fails. Thus, comparisons between genetic algorithms should utilize more difficult test functions.

4.6 Empirical Test Results Without Gray Coding

The fact that random mutation hill-climbing is shown here to solve almost half of the functions in the test set more quickly than any of the genetic algorithms tested (see Table 3) indicates that some of these test functions are relatively easy to solve when Gray coding is used. In fact, for those functions where random mutation hill-climbing performs best and Gray coding is used, the best set of guiding parameters for the ESGA use a value of 0.0 for P_C (i.e., no crossover). Therefore, we compared the performance of the random mutation hill-climber with that of the CHC and delta coding genetic algorithms on the same set of test functions without Gray coding that were previously compared using Gray coding (Table 8).

When Gray coding is not applied to the functions in the test set, random mutation hill-climbing can solve only one of the test functions consistently (F3). Even the CHC genetic

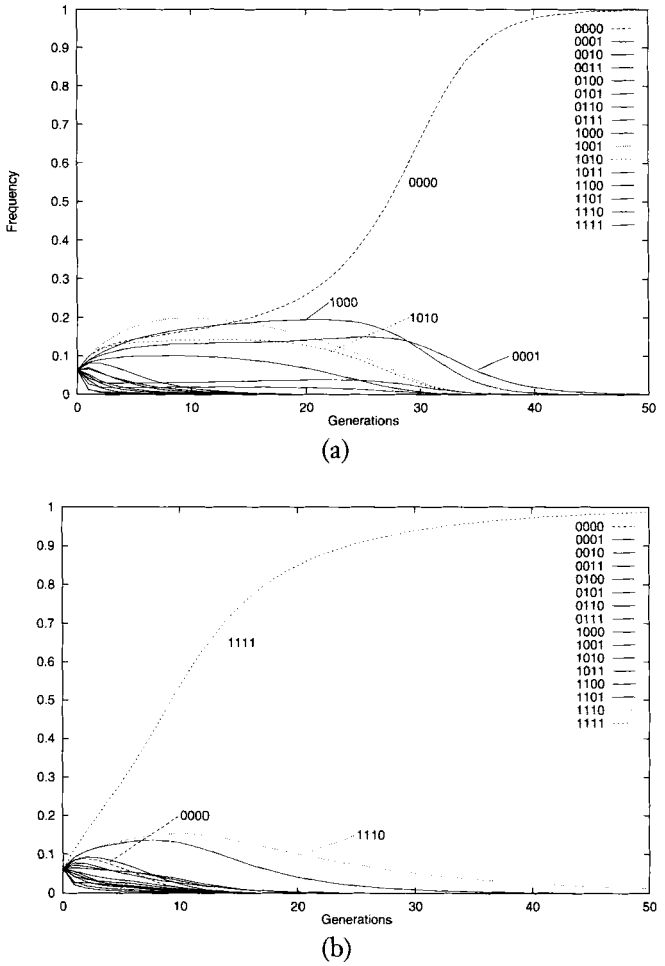

Figure 8. String competitions for binary and Gray coding of four-bit problem.

algorithm cannot consistently solve the F6 and F8 test functions without the use of Gray coding. Thus, when a binary encoding is used, these problems are much more difficult to solve.

5. Analysis of Delta Coding

5.1 Oscillation in Delta Coding

While delta coding was able to consistently solve most of the test functions used here over a wide variety of parameter settings, the Rastrigin (F6) and Schwefel (F7) functions were significantly more challenging for the algorithm. Not only was it more difficult to find parameters for the delta coding algorithm that consistently solved the F6 and F7 functions, but optimizing its performance on the functions proved to be difficult. Analysis of the delta coding performance data for these functions indicated that the algorithm often exhibited an *oscillation* condition.

Oscillation occurs when the number of bits representing each parameter has been re-

Table 8. The performance of random mutation hill-climbing (RMHC) and the CHC and delta coding genetic algorithms without Gray coding.

	RMHC Algorithm							
Problem	F1	F2	F3	F4	F5	F6	F7	F8
Percent Solved	20%	23%	100%	20%	40%	00%	40%	00%
Average Trials	384	27672	544	44967	884		346667	
Maximum Trials	200000	100000		100000	50000	500000	500000	500000
Average Best	0.0001	0.0002		−1.73	1.16	6.71	−4170.72	0.182
Mutation Rate	0.05	0.09	0.04	0.02	0.08	0.03	0.05	0.03
	CHC Algorithm							
Percent Solved	100%	100%	100%	100%	100%	00%	100%	23%
Population Size	50	50	50	50	50	50	50	50
Average Trials	56892	37737	687	17017	4443		15230	345242
Maximum Trials						500000		500000
Average Best						0.183		0.038
	Delta Coding Algorithm							
Percent Solved	100%	100%	100%	100%	100%	100%	100%	100%
Population Size	15	25	15	25	50	800	100	200
Average Trials	674	2365	521	5027	1204	250005	22852	48806
Linear Bias Selection	2.00	2.00	2.00	2.00	2.00	1.90	1.90	2.00
Mutation Rate	0.04	0.02	0.05	0.03	0.06	0.01	0.02	0.005

duced to a lower limit such that the parameter representations cannot be reduced further. The algorithm converges to some solution, A. In the next iteration the algorithm converges on a new solution, B. The solution is different, and so the number of bits used to encode the parameters is not altered. The algorithm converges back to solution A on the next iteration. No alteration of the parameter bit representations is engaged as none of the conditions to enlarge the space are met and the parameter representations have been reduced to their lower limit. On the next iteration the algorithm converges back to solution B. Thus, the algorithm continues to oscillate between the two solutions.

This phenomenon can be illustrated using a subpartition of a function space as shown in Figure 9. Assume that the delta coding algorithm has converged on the string 0101 in the four-bit subspace shown and that the representation for the parameter is then reduced to three bits. This remaps the hyperspace subpartition being searched to the *Interim 1 Subpartition* labeled in Figure 9. The string, 0101 in the four-bit subpartition, is remapped

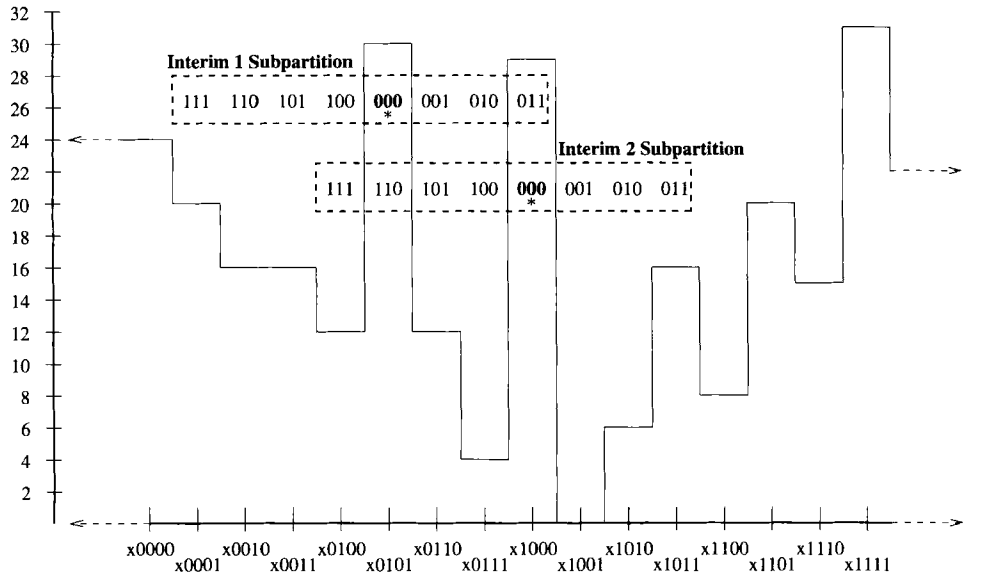

Figure 9. Oscillation problem search space with interim encodings. Note that the x prefacing each of the four-bit strings indicates that this subspace is a part of a larger search space.

to 000. We can use the output of Whitley's executable model of a simple genetic algorithm without mutation (Whitley, 1993) to track the proportional representation of the strings expected in the SGA population when searching the Interim 1 Subpartition. (Although an SGA was not used by delta coding in the empirical experiments, the point here is to illustrate the oscillation problem.) Figure 10(a) shows that by the twentieth generation the population has converged on the string 011, which is equivalent to 1000 in the four-bit subpartition.

The interim solution 011 (i.e., 1000 in the four-bit subpartition) is then mapped to the string 000 in the subspace defined as *Interim 2 Subpartition* in Figure 9. The executable model shows that the SGA then converges on the string 110 when the Interim 2 Subpartition is searched (Figure 10(b)) using this new mapping. This is equivalent to the string 0101 in the original four-bit subpartition. Thus, the search returns to the Interim 1 Subpartition. And since the algorithm will search the same subpartition and employ the same mapping used when previously searching this subpartition, the search will continue to oscillate between the two interim solutions represented by the strings 0101 and 1000 in the original four-bit subpartition. This phenomenon was frequently observed in performance tests on the Rastrigin (F6) and Schwefel (F7) functions.

One possible method for dealing with the problem of oscillation in the delta coding algorithm is to save the most recent interim solutions for several previous iterations of genetic search. Each new interim solution could then be compared with previous interim solutions to identify the occurrence of oscillation. After detecting oscillation several mechanisms might be helpful for ensuring that the search does not continue to oscillate. The simplest option available is to enlarge the search window by increasing the number of bits used to represent each of the function parameters. The parameter bit representations could be altered in a number of ways including resetting them to their original lengths or just increasing each by

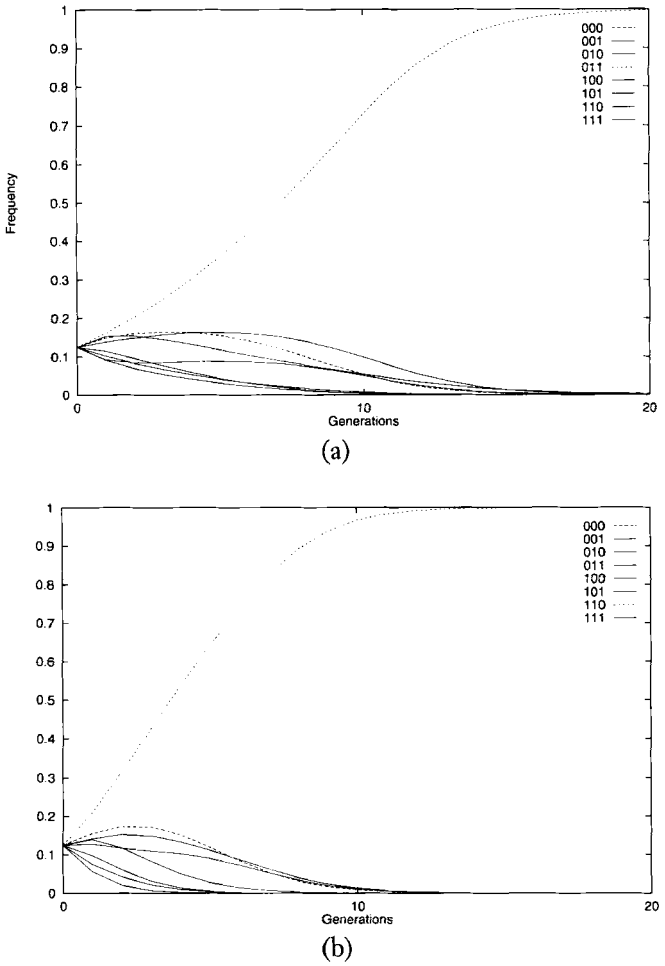

Figure 10. String competitions for interim encodings.

a single bit as if the algorithm had reconverged to the same interim solution on consecutive iterations.

Another method for preventing further oscillation once it has been detected is to test combinations of the parameter values contained in the most recent previous interim solutions. For example, if there were four function parameters for a problem, combining the first and second parameters from one interim solution with the third and fourth parameters from another interim solution would result in a new set of input parameters for evaluation. While the number of possible combinations would grow rather quickly as the number of parameters and interim solutions increased, analysis of the interim solutions involved in oscillation conditions for the Rastrigin and Schwefel function tests showed that this particular method would have been very effective for avoiding oscillation and would have resulted in much faster convergence to the global optimum.

Another way of effectively testing these combinations without explicitly evaluating all of the parameter combinations could be accomplished by seeding the population with strings that represented those parameters that were observed in previous interim solutions when

decoded with respect to the current interim solution. The current interim solution and the most recent interim solution are always located within the current search window; however, it may be necessary to increase the size of the search window if one wishes to seed the population with interim solutions other than the current and the most recent interim solutions. By seeding the population using multiple previous interim solutions the binary representations for the various parameter values contained in the previous interim solutions would bias the samples contained in the current population, thereby influencing the next iteration of search by potentially exploiting highly fit building blocks.

Another weakness of the delta coding algorithm noted from the analysis of its performance data on several of the test functions is that it often searches subpartitions of the fitness landscape using mappings whereby all offspring created during recombination have fitness values that are worse than the current members of the population. Thus, offspring are never inserted into the ranked population structure used by the GENITOR search mechanism. This has the side effect of stalling the delta coding search. When offspring are not inserted into the population the two strings used to measure population diversity never change. Therefore, the criteria are never met to terminate the delta iteration. This led to the insertion of a protection mechanism whereby a particular delta iteration was always limited to some maximum number of trials. Tuning of this parameter was critical to the empirical performance of delta coding on some of the functions in this test suite (i.e., F6 and F7) and often led to performance improvements on many of the test functions.

5.2 Overhead of the Delta Coding Algorithm

The measurement of delta coding performance against other algorithms using the number of recombinations as the performance criteria overlooks an important component of the different algorithms. All of the algorithms have some computational overhead inherent in the design of the search method. However, there is a relatively high amount of computational overhead inherent in the delta coding algorithm. This overhead is primarily the result of two mechanisms in the algorithm: (1) testing the Hamming distance between population members to check for adequate population diversity to continue the search and (2) the total reinitialization of the population for each delta iteration. Although every algorithm has some associated overhead, delta coding is perhaps one of the most expensive. Thus, testing the performance of these algorithms using a time measurement would yield a better indication of how effectively each algorithm is able to solve these test functions.

Performance times for random mutation hill-climbing and the CHC and delta coding algorithms are illustrated in Table 9. Table 9 shows the average time required to solve the respective functions in the test suite to optimality when the algorithms employed Gray coding. Table 10 shows the average time to solve the same functions to optimality when Gray coding is not applied. The standard deviation (σ) for the average times is also listed along with the percentage of runs (out of 30 independent experiments) in which the algorithms were successful. The times are recorded in seconds as measured on a SparcStation II. The average time is computed over all 30 runs whether the algorithm was successful in locating the optimal solution or not. The algorithms were all allowed to run for some maximum number of seconds so that in no case was an algorithm that was not able to consistently locate the optimal solution allowed less overall time to run than those algorithms that were successful.

For each algorithm, *significance* in Table 9 indicates which of the other two algorithms were determined to perform significantly worse than the algorithm in question with a 99% confidence level. When two algorithms are compared that both solve a test function consis-

Table 9. CPU seconds to solve test suite functions for random mutation hill-climbing (RMHC), CHC, and delta coding algorithms using Gray coding.

Problem	F1	F2	F3	F4	F5	F6	F7	F8
				RMHC Algorithm				
% Solved	100%	100%	100%	37%	100%	100%	100%	3%
Avg. Time	0.15	5.63	0.36	458.00	0.29	254.76	180.47	573.19
σ	0.06	3.77	0.02	146.85	0.21	123.96	79.46	43.02
Avg. Best				-1.92				0.099
Significance	> *both*		> *both*		> *chc*	> *δ coding*		
				CHC Algorithm				
% Solved	100%	100%	100%	100%	100%	100%	100%	100%
Avg. Time	0.57	1.91	0.76	36.45	0.46	284.44	11.51	50.49
σ	0.19	0.85	0.24	16.49	0.21	128.94	3.31	40.21
Significance		> *rmhc*		> *rmhc*		> *δ coding*	> *both*	> *rmhc*
				Delta Coding Algorithm				
% Solved	100%	100%	100%	100%	100%	100%	100%	100%
Avg. Time	0.31	1.40	0.82	12.45	0.34	804.29	25.75	65.64
σ	0.10	0.63	0.40	4.05	0.12	65.71	10.25	28.84
Significance	> *chc*	> *both*		> *both*	> *chc*		> *rmhc*	> *rmhc*

tently, statistical significance is determined by applying a two-tailed t-test using the average CPU time taken to solve the function. When both of the algorithms do not consistently solve the test function they are compared using a χ^2 test using the number of times out of 30 that the respective algorithms are able to solve the function. In the event that neither algorithm is able to solve the function the average best solutions are compared using a two-tailed t-test. All of the parameters used for obtaining the times are the same as those shown in Tables 2 through 4 except for the maximum trials allowed. These tests were constrained directly by CPU time rather than trials.

The results in Table 9 show that when a problem is easily hill-climbed random mutation hill-climbing is the most efficient algorithm (in terms of time) of those tested here. The overhead in delta coding appears to make the problem much more difficult than necessary in these cases. This is evident when the number of recombinations used to consistently solve the function by the delta coding and random mutation hill-climbing (RMHC) are comparable. For example, RMHC on the average takes only slightly more recombinations to locate the optimal solution for F1 when Gray coding is applied than does the delta coding algorithm. However, the execution time for the delta coding search is approximately twice

Table 10. CPU seconds to solve test suite functions for random mutation hill-climbing (RMHC), CHC, and delta coding algorithms *without* Gray coding.

Problem	F1	F2	F3	F4	F5	F6	F7	F8
				RMHC Algorithm				
% Solved	20%	30%	100%	27%	40%	0%	23%	0%
Avg. Time	138.51	107.48	0.30	295.18	17.46	1615.50	676.25	745.73
σ	70.38	50.54	0.18	84.63	14.26	7.19	106.81	5.49
Avg. Best	0.0001	0.0002		−1.96	1.16	5.89	−4109.43	0.182
Significance			> *chc*					
				CHC Algorithm				
% Solved	100%	100%	100%	100%	100%	0%	100%	43%
Avg. Time	20.21	7.18	0.51	27.02	1.91	1525.86	16.50	704.31
σ	20.48	9.62	0.07	11.24	1.96	16.10	4.66	210.02
Avg. Best						0.175		0.020
Significance	> *rmhc*	> *rmhc*		> *rmhc*	> *rmhc*	> *rmhc*	> *both*	> *rmhc*
				Delta Coding Algorithm				
% Solved	100%	100%	100%	100%	100%	100%	100%	100%
Avg. Time	0.35	1.03	0.41	12.86	0.87	754.63	34.10	60.16
σ	0.14	0.65	0.29	6.01	0.56	52.08	8.67	19.19
Significance	> *both*	> *both*		> *both*	> *both*	> *both*	> *rmhc*	> *both*

that for RMHC. However, when a problem is more difficult and is not as easily hill-climbed (Table 10), delta coding is able to solve the problems to optimality much faster than RMHC.

Delta coding also appears to have more overhead than the CHC adaptive search algorithm. This is illustrated by noting that the CHC algorithm requires a third more recombinations on the average to solve the F3 function to optimality than delta coding. However, the CHC algorithm executes in slightly less time than the delta coding algorithm when searching for the optimal solution to the F3 function. The dramatic difference in overhead is directly related to the number of times that the delta coding reinitializes the population to sustain genetic search.

6. Conclusions and Future Work

CHC performance on the test suite used here confirms old results (Eshelman, 1991) and offers new evidence that the CHC algorithm is very competitive with other genetic algorithms. In fact, under certain conditions it performs better than all of the other algorithms.

These results also show that the CHC algorithm is very robust in the sense that little, if any, parameter tuning is necessary to achieve very good results.

The performance of the CHC and delta coding algorithms also suggests that restarts during genetic search are an effective method for maintaining population diversity. DeJong and Spears (1989) and Krishnakumar (1989) have also previously used genetic search with restarts. Restarts sustain search and preclude having to use a large initial population or high rate of mutation. The ESGA and GENITOR algorithms are able to use small populations and still locate the optimum for some of these test functions. However, this strategy only works well on functions where random mutation hill-climbing performs well. This might indicate that the populations in these algorithms converge quickly and depend on mutation to search out the solution.

These results represent the first standard benchmark comparison of delta coding with other genetic algorithms. Here, delta coding consistently performs well on this suite of test problems with or without Gray coding. In fact, the performance of delta coding is often enhanced when Gray coding is not employed. When Gray coding is not used, delta coding is the only algorithm able to solve all of these test functions consistently and, in all but a single case, delta coding performs better than the other algorithms against which it has been compared.

The results reported in this paper also suggest that the suite of test problems used here, although currently serving as common benchmarks, needs to be reviewed. Many of these functions were never intended to be lasting genetic algorithm benchmarks (DeJong, 1975). We have shown here that random mutation hill-climbing performs better than genetic algorithms on several of these test functions. However, simple random mutation hill-climbers do not always perform well on real-world problems, and genetic algorithms are most needed when simpler methods fail. We therefore argue that a good test suite should be resistant to simple stochastic hill-climbing algorithms.

For similar reasons, we also question the use of representations that make a function easy for hill-climbers when performing comparative tests for genetic algorithms. Problem transforms such as Gray coding dramatically alter key features of the function space such as the number of local optima, relative attraction basin sizes and hyperplane competitions for many of these test functions (Mathias & Whitley, 1994). If the goal is to optimize a given application and Gray coding or any other representation reduces the work necessary to solve the problem, then it should be employed. On the other hand, when doing comparative studies the use of problem representations that are solvable by stochastic hill-climbing may not provide enough of a challenge to adequately exercise and test the power of a genetic algorithm. After all, genetic algorithms are most useful when other simpler, less costly methods fail.

Theoretically, there is no reason to expect that Gray coding an *arbitrary* function will make it any easier to locate the optimum solution via genetic search. Nevertheless, for this particular set of test functions Gray coding has a significant impact on the performance of random mutation hill-climbing. This suggests that the continued use of Gray coding for these problems in genetic algorithm comparisons may result in the perception that those genetic algorithms with strong hill-climbing components are the superior algorithms.

Acknowledgments

We would especially like to thank Larry Eshelman for his help in guiding our CHC implementation effort and for providing useful insights into this work. The reviewers and editors of *Evolutionary Computation* also provided helpful suggestions. This work was supported in

part by NSF grant IRI-9312748 and by the Colorado Advanced Software Institute (CASI). CASI is sponsored in part by the Colorado Advanced Technology Institute for purposes of economic development.

References

Baker, J. (1987). Reducing bias and inefficiency in the selection algorithm. In J. J. Grefenstette (Ed.), *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference* (pp. 14–21). Hillsdale, NJ: Lawrence Erlbaum Associates.

Bitner, J. R., Ehrlich, G., & Reingold, E. M. (1976). Efficient generation of the binary reflected Gray code and its applications. *Communications of the ACM, 19(9)*, 517–521.

Booker, L. (1987). Improving search in genetic algorithms. In L. Davis (Ed.), *Genetic Algorithms and Simulated Annealing* (pp. 61–73). San Mateo, CA: Morgan Kaufmann.

Caruana, R., & Schaffer, J. D. (1988). Representation and hidden bias: Gray vs. binary coding for genetic algorithms. In *Proceedings of the Fifth International Conference on Machine Learning* (pp. 132–161). San Mateo, CA: Morgan Kaufmann.

Chen, M., & Shin, K. (1987). Processor allocation in an N-cube multiprocessor using Gray codes. *IEEE Transactions on Computers, C-36*, 1396–1407.

Davis, L. (1989). Adapting operator probabilities in genetic algorithms. In J. D. Schaffer (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 61–69). San Mateo, CA: Morgan Kaufmann.

Davis, L. (1991). Bit-climbing, representational bias, and test suite design. In L. Booker and R. Belew (Eds.), *Proceedings of the Fourth International Conference on Genetic Algorithms* (pp. 18–23). San Mateo, CA: Morgan Kaufmann.

DeJong, K. (1975). *An analysis of the behavior of a class of genetic adaptive systems.* Doctoral dissertation, University of Michigan, Ann Arbor, MI.

DeJong, K., & Spears, W. (1989). Using genetic algorithms to solve NP-complete problems. In J. D. Schaffer (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 124–132). San Mateo, CA: Morgan Kaufmann.

Eshelman, L. (1991). The CHC adaptive search algorithm. How to have safe search when engaging in nontraditional genetic recombination. In G. Rawlins (Ed.), *Foundations of Genetic Algorithms* (pp. 265–283). San Mateo, CA: Morgan Kaufmann.

Eshelman, L. (1993). Personal communication.

Forrest, S., & Mitchell, M. (1993). Relative building-block fitness and the building block hypothesis. In D. Whitley (Ed.), *Foundations of Genetic Algorithms - 2* (pp. 109–126). San Mateo, CA: Morgan Kaufmann.

Gilbert, E. (1958). Gray codes and paths on the n-cube. *The Bell System Technical Journal* (pp. 815–826), May.

Goldberg, D. (1989a). *Genetic algorithms in search, optimization and machine learning.* Reading, MA: Addison-Wesley.

Goldberg, D. (1989b). Sizing populations for serial and parallel genetic algorithms. In J. D. Schaffer (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 70–79). San Mateo, CA: Morgan Kaufmann.

Gordon, V., & Whitley, D. (1993). Serial and parallel genetic algorithms as function optimizers. In S. Forrest (Ed.), *Proceedings of the Fifth International Conference on Genetic Algorithms* (pp. 177–183). San Mateo, CA: Morgan Kaufmann.

Hamming, R. (1980). *Coding and information theory.* Englewood Cliffs, NJ: Prentice-Hall, Inc.

Hoffmeister, F., & Bäck, T. (1991). Genetic algorithms and evolution strategies: similarities and differences. In H. P. Schwefel and R. Männer (Eds.), *Parallel problem solving from nature* (pp. 455–469). Berlin: Springer/Verlag (Lecture Notes in Computer Science).

Hollstien, R. (1971). *Artificial genetic adaptation in computer control systems.* Doctoral dissertation, University of Michigan, Ann Arbor, MI.

Horn, J., & Goldberg, D. (1995). Genetic algorithm difficulty and the modality of the fitness landscape. In D. Whitley and M. Vose (Eds.), *Foundations of Genetic Algorithms - 3.* San Mateo, CA: Morgan Kaufmann.

Krishnakumar, K. (1989). Micro-genetic algorithms for stationary and non-stationary function optimization. In *SPIE's Intelligent Control and Adaptive Systems Conference.*

Mathias, K., & Whitley, D. (1993). Remapping hyperspace during genetic search: canonical delta folding. In D. Whitley (Ed.), *Foundations of Genetic Algorithms - 2* (pp. 167–186). San Mateo, CA: Morgan Kaufmann.

Mathias, K., & Whitley, D. (1994). Transforming the search space with Gray coding. In J. D. Schaffer (Ed.), *Proceedings of the IEEE International Conference on Evolutionary Computation* (pp. 513–518). Piscataway, NJ: IEEE Service Center.

Mühlenbein, H., Schomisch, M., & Born, J. (1991). The parallel genetic algorithm as function optimizer. In L. Booker and R. Belew (Eds.), *Proceedings of the Fourth International Conference on Genetic Algorithms* (pp. 271–278). San Mateo, CA: Morgan Kaufmann.

Schaffer, J. D., Caruana, R., Eshelman, L., & Das, R. (1989). A study of control parameters affecting online performance of genetic algorithms for function optimization. In J. D. Schaffer (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 51–60). San Mateo, CA: Morgan Kaufmann.

Schaffer, J. D., & Morishima, A. (1987). An adaptive crossover distribution mechanism for genetic algorithms. In J. J. Grefenstette (Ed.), *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference* (pp. 36–40). Hillsdale, NJ: Lawrence Erlbaum Associates.

Schaffer, J. D., & Morishima, A. (1988). Adaptive knowledge representation: A context sensitive recombination mechanism for genetic algorithms. *International Journal of Intelligent Systems, 3,* 229–246.

Schraudolph, N., & Belew, R. (1992). Dynamic parameter encoding for genetic algorithms. *Machine Learning, 9,* 9–21.

Schwefel, H. P. (1981). *Numerical optimization of computer models.* Chichester, England: John Wiley.

Schwefel, H.P. (1987). Collective phenomena in evolutionary systems. In *31st Annual Meeting of the International Society for General Systems Research* (pp. 1025–1033).

Shaefer, C. (1987). The ARGOT strategy: Adaptive representation genetic optimizer technique. In J. J. Grefenstette (Ed.), *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference* (pp. 50–58). Hillsdale, NJ: Lawrence Erlbaum Associates.

Starkweather, T., Whitley, D., & Mathias, K. (1990). Optimization using distributed genetic algorithms. In H. P. Schwefel and R. Männer (Eds.), *Parallel problem solving from nature* (pp. 176–185). Berlin: Springer/Verlag (Lecture Notes in Computer Science).

Vose, M., & Liepins, G. (1991). Punctuated equilibria in genetic search. *Complex Systems, 5,* 31–44.

Whitley, D. (1991). Fundamental principles of deception in genetic search. In G. Rawlins (Ed.), *Foundations of Genetic Algorithms* (pp. 221–241). San Mateo, CA: Morgan Kaufmann.

Whitley, D. (1993). An executable model of the simple genetic algorithm. In D. Whitley (Ed.), *Foundations of Genetic Algorithms - 2* (pp. 45–62). San Mateo, CA: Morgan Kaufmann.

Whitley, D. (1994). A genetic algorithm tutorial. *Statistics and Computing, 4,* 65–85.

Whitley, D., & Kauth, J. (1988). GENITOR: A different genetic algorithm. In *Proceedings of the 1988 Rocky Mountain Conference on Artificial Intelligence*.

Whitley, D., Mathias, K., & Fitzhorn, P. (1991). Delta coding: An iterative strategy for genetic algorithms. In L. Booker and R. Belew (Eds.), *Proceedings of the Fourth International Conference on Genetic Algorithms* (pp. 77–84). San Mateo, CA: Morgan Kaufmann.