University of Alberta

Genetic Invariance: A New Paradigm for Genetic Algorithm Design

by

Joseph Culberson

Technical Report TR 92–02
June 1992

DEPARTMENT OF COMPUTING SCIENCE
The University of Alberta
Edmonton, Alberta, Canada

Genetic Invariance: A New Paradigm for Genetic Algorithm Design

Joseph C. Culberson * †

June 18,1992; Revised May 2,1994

Abstract

This paper presents some experimental results and analyses of the gene invariant genetic algorithm(GIGA). Although a subclass of the class of genetic algorithms, this algorithm and its variations represent a unique approach with many interesting results. The primary distinguishing feature is that when a pair of offspring are created and chosen as worthy of membership in the population they replace their parents. With no mutation this has the effect of maintaining the original genetic material over time, although it is reorganized.

In this paper no mutation is allowed. The only genetic operator used is crossover. Several crossover operators are experimented with and analyzed. The notion of a family is introduced and different selection methods are analyzed.

Tests using simple functions, the De Jong five function test suite and several deceptive functions are reported. GIGA performs as well as traditional GAs, and sometimes better. The evidence indicates that this method makes more effective use of the crossover operator, in part because it never loses genetic material and thus has greater scope for recombination.

A new view of crossover search space structures and approaches to analysis are presented. Traditional methods of analysis for GAs do not seem to apply since GIGAs cannot be said to give increased trials to the best schemata in the usual sense. However, the analysis of crossover search space structures may have applications in traditional GA analysis.

*Supported by Natural Sciences and Engineering Research Council Grant No. OGP8053. Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, T6G 2H1. email:joe@cs.ualberta.ca

†This paper is available via ftp thorhild.cs.ualberta.ca in pub/GIGA

1 Introduction

Genetic algorithms have traditionally been designed using the guiding principle of increased allocations of trials to those members which are most fit. Analysis based on the study of *schema* (i.e. hyperplane) allocation indicates that this is a reasonable thing to do [17, 12]. However, in this paper a different approach to the use of the crossover operators within a population of individuals is taken, wherein it cannot be claimed that there are increased trials for above average individuals. In this way we introduce an approach to understanding and designing genetic algorithms based on the propagation of building blocks through a population, rather than their concentration through convergence.

For a thorough introduction to genetic algorithms the reader is referred to [4, 5, 17, 12, 20]. Here we outline the basic features of genetic algorithms (GAs) and the specifics peculiar to gene invariant genetic algorithms(GIGAs).

A genetic algorithm maintains a *population* of n strings, called *individuals* or *members*. Each string is of length l and is drawn from a set of α characters, called the *alphabet*. Often the alphabet is binary, that is $\alpha = 2$, but larger character sets are allowed, and in some applications [5] the alphabet is the set of real numbers, and thus not even finite. The population varies over time under the control of the genetic algorithm as members are replaced by new strings. These new strings are generated from previous members through the action of various genetic operators to be discussed shortly. In some cases [13] n and l may vary with time, but in this paper they are assumed to be fixed.

The members of a population are evaluated by the *environment*, which means that the program has some means of determining the value of a string. In the realm of function optimization, which will be the primary focus of this paper, we supply a routine for computing a real value from any properly defined string. These values may be modified in some way, through *scaling* for example[12]. This modified value is referred to as the *fitness* of the string.

We designate the population by the matrix notation $\mathbf{P}_{ij}^{t}$, which refers to the jth character of the ith member of the population at time step t. To refer to a particular member as a whole, we drop the second subscript, and when no confusion arises we drop the superscript indicating time.

The genetic operator most often used to produce new strings is *crossover*. Two members of the population $\mathbf{P}_1$ and $\mathbf{P}_2$ (called the *parents*) are selected, and a pair (usually) of new strings $\mathbf{C}_1$ and $\mathbf{C}_2$ (called the *children*) is formed

in which for each j either

$$\mathbf{C}_{1j} = \mathbf{P}_{1j} \text{ and } \mathbf{C}_{2j} = \mathbf{P}_{2j}$$

or

$$\mathbf{C}_{1j} = \mathbf{P}_{2j} \text{ and } \mathbf{C}_{2j} = \mathbf{P}_{1j}$$

That is, the characters of the children are the same as those of the parents, but may be switched between them. Various crossover types are described in the literature. In *one point* crossover an integer k, $1 < k < l$ is chosen, and for $j < k$ the first condition holds, while for $j \geq k$ the second condition holds. In *multipoint* crossover a number of such points are chosen and the intervals alternate between the two conditions. In *uniform* crossover, the choice between the two conditions is made independently character by character with some fixed probability p. (Because of symmetry, it does not matter whether p applies to the first or second condition.) If $p = 0.5$ then we say the crossover is *unbiased*, otherwise it is more strongly biased the further p is from one half. Additional crossover operations will be described as required.

In the GIGA implementation used for the experiments in this paper there is one point, multipoint with a user specified number of cross points and uniform crossover. Also there is the ability to specify that each crossover be chosen from several possibilities with some fixed probability. When this option is used we specify the proportional frequencies by fractions. See [3] for further details.

Other genetic operators include transpositions, reordering, reversal and mutation[12]. These are not currently available in the program described in this paper.

Important features of GAs are the methods of selection of parents and the methods of creating new populations from the children. Two combinations dominate the literature (with numerous variations);

- In *Roulette wheel* selection parents are selected all at once with a probability of being selected in proportion to their fitness, and the children are used to make up the replacement population.

- *Static* genetic algorithms select parent pairs (possibly biased by fitness) and produce child pairs. Individuals from the population are selected for replacement by the children, and this selection is usually done on the basis of fitness, where the least fit individuals have the highest probability of being replaced.

In either case, individuals may be selected several times for mating. As a result the population tends to evolve toward a set of highly similar individuals. In particular, the probability that $\mathbf{P}^t_{ij} = \mathbf{P}^t_{kj}$ for any two individuals i and k increases rapidly with t. We refer to this as *convergence* of the population. To offset this convergence, most GAs rely on *mutation*, and other operators and conditions [12].

Gene invariant algorithms (GIGAs), the subject of this paper, form a subclass of genetic algorithms, principally distinguished by the notion that the total genetic makeup of the population does not change with time. In particular, the multiset of characters of any column of the population $\mathbf{P}^t_{*j}$ *does not change with time.*[1] To maintain this invariance, it is only necessary that a pair of children produced by the crossover operation replace their parents in the population. The invariance rule then follows trivially, provided there is no mutation or other genetic operator being used.

A *family* is a set of pairs produced by a set of crossover operations performed on a single pair of parents. In GIGA when a pair of parents is selected, they are used to generate a family. The number of pairs s is called the family size. The best pair is selected from the family and replaces the parents. If *elitism* is invoked, the selection of the best pair includes the parents as part of the family. Many different notions of what constitutes a "best" pair can be defined. Some that have been tried are: the pair with the largest maximum value (or the smallest minimum if we are minimizing), the pair with largest difference in value, or even the pair with the smallest difference. Usually we will use the first definition, unless otherwise noted, with the maximum or minimum being understood in context.

The sequence of a selection of parents, production of a family and replacement of parents is called a *mating cycle* or *mating*. The program terminates after a specified number of matings m have been performed. The number of evaluations performed by the environment is no more than $2sm + n$. The number may be less because there is an equality test on parents and if they are found equal, then no family is produced. See [3] for details.

When replacing parents, we always put the child with the larger value in the row of higher index. The effect, for well behaved functions, is that the population will tend to become sorted by value. The efficacy of the sorting (and consequently the search) will depend on the selection criteria

[1] We will undoubtedly wish to relax this rule in future research, for example by adding operators such as mutation and transposition. But in this paper, the rule will be adhered to rigorously.

for the parents. Several mechanisms are available in the program, others are suggested in [3] and readers are encouraged to develop their own.

For many problems, crossing parents which differ widely in value is likely to produce offspring of intermediate value, and so no progress will be made either in the minimum or maximum values. This observation is based on the assumption that similar values are reflective of string similarities, an assumption we must make if there is to be any use made of crossover.[2] Assuming we want to maximize or minimize some function, we are more likely to make local progress in terms of increased fitness if we mate strings of similar value. At the other extreme, crossing identical strings, or strings with Hamming distance less than two will produce strings identical to their parents.

In this implementation, we always select parents in adjacent rows of the population. The selection varies over the population, so that $\mathbf{P}_1^0, \mathbf{P}_2^0$ are the first to mate, then $\mathbf{P}_2^1, \mathbf{P}_3^1$, and so on up to $\mathbf{P}_{n-1}^{n-2}, \mathbf{P}_n^{n-2}$. Then over the next $n-2$ matings the pairs are chosen from the top to the bottom. The selection continues to alternate over the population. We refer to a set of $n-2$ matings as a *pass* meaning that we have passed once over the population. Other selection orderings can be specified if desired [3].

To further improve the efficiency of the program, the user can specify that the population be sorted and maintained in sorted order. Populations may initially be seeded by a random rotation. This means that a member of the population is chosen at random, and then $\alpha - 1$ further members are chosen so that the characters in each position rotate through the alphabet. A discussion of program options and operation together with suggestions for future enhancements is presented in [3]. The source code can be found in [2].

Michael Lewchuk in his master's thesis [18] investigated a special case of GIGA with analysis on the one max function and simulations on De Jong's test suite. In his method parent selection is restricted to that pair in the population that is closest in value, each mating produces a family of size one, and there is no elitism. That is, a single pair is produced and always replaces the parents. The approach is interesting because nowhere is there an explicit selection for optimal (or even superior) values. Although these restrictions appear severe and defy almost all of the principles usually used

[2] In light of the results in section 3 similarities may mean the absence of certain characters or patterns, not just the presence of them. This discussion is necessarily vague until we can determine formal and encompassing definitions for these concepts.

in GA design, the program nevertheless performs quite well on several of the test functions, outperforming the simple GA in some cases. His simulations were run on a program of his own, and this technique is not available at this time in the online program.

In this paper, we examine three sets of experiments. These are available as part of the programming system [2]. It is hoped that the reader will make use of this program to verify the claims made in this paper. Note that the use of a different random number generator may cause some of the specific results to vary.

Section 2 introduces the reader to the principle of propagation of values through the population by the action of crossover. Simple functions with single runs and frequent output of the population help illustrate the mechanisms used by GIGA to search functions for optimal values.

In section 3 we look at various deceptive functions. The well known deceptive functions are easily solved by GIGA. Some seemingly much more difficult functions are devised on which GIGA exhibits a complex search behavior that would be difficult to mimic with a TGA. A provably difficult function is devised that is effective in deceiving the program. Analysis here also illustrates the benefit of more than one type of crossover being made available to GIGA.

In section 4 GIGA is tested on De Jong's five function test suite [8]. This test suite has recently received criticism[11, 6, 9, 21, 22, 7] because tests have shown that naive evolution(NE) (i.e. a GA using only mutation – which means it is essentially a stochastic Hamming hill climber) often outperforms traditional GAs(TGAs). We show that GIGA using only crossover and no mutation, often equals or out performs a TGA and sometimes equals NE.

Finally in section 5 a summary of claimed results and possibilities is given. It is hoped the reader will be challenged to prove or disprove the claims made throughout this paper, and extend the research in new directions.

2 Some Simple Test Functions

The experiments in this section are on simple functions. They illustrate some of the properties of GIGA and emphasize the differences between crossover operators. The reader is encouraged to run the program on these functions with the population printing turned on, and the sorting facilities turned off. This will illustrate how progress is achieved by GIGA. The effects of

	One Max Experiments			
	1	1a	2	3
Maximum	6586*	5239	1337	3347
Minimum	5708*	5577	1391	3225

Table 1: Average evaluations on the one-max function.

changing various parameters can also be tested. The experiments in this section have all been run with elitism and sorting turned on, and using the random rotation method of initializing the population. When one point crossover is used, the different pairs of offspring in a family are guaranteed to be generated by different crossover points.

The first set of experiments is on the well known *one max* or *ones counting* function, wherein the value of a string is the number of ones in it. This provides an opportunity to discuss the search space structures generated by various crossover operators, and how GIGA exploits them. We then briefly discuss the *majority function*, computed as the absolute value of the number of ones minus the number of zeroes. This illustrates the power of GIGA to search severely bimodal functions. Finally we try GIGA on the simple binary number function, which shows the sorting nature of GIGA. Additional simple functions are included in the program [2].

The experiments on the one max function have the parameters $l = 60, \alpha = 2, s = 2$ and averages are taken over 50 runs. Four experiments are discussed, with input files **onemax.1, onemax.1a, onemax.2** and **onemax.3** available with the program [2]. Experiments **1** and **1a** use one point crossover with populations of size 10 and 20 respectively. Experiments **2** and **3** use uniform crossover with $p = 0.50$ and $p = 0.05$ respectively. The population is of size $n = 10$. The pair with the largest maximum was chosen to replace the parents.

The average number of evaluations to find the maximum and minimum values (note they are both found in the same search) for each experiment can be found in table 1. For **1** the program failed to find the minimum once and the maximum twice in 5000 mating cycles(starred entries). The averages are taken over the successful runs in this case. In all other experiments all trials were successful.

Clearly the unbiased uniform crossover is superior on this function. One point crossover benefits from an increase in population size but is still unable

to compete with even the biased uniform crossover. On the other hand, not evident from the table, the biased uniform is the only one capable of completely sorting the population. That is, given sufficient mating cycles the population will eventually be sorted into two sets of strings, those with value 0 and those with value 60.

A study of why these results occur illustrates both properties of the various crossover types and the mechanism by which optimization is carried out by GIGA. Because the population is sorted, at time $t = 0$ adjacent strings in the population will contain nearly equal numbers of ones. An example is shown in figure 1 where the difference between adjacent strings is always three or less.

```
23  001000111110000001000000001100000100111011011011101000010100
24  000010100001100010100100101101001000110100011000010001111011
26  010100100100010001111011000010100110011110000100100111100100
29  000000110101110001100111110011011100001100100101101101101000
30  011110001011111001000011100101000000000100111011110101011101
30  100001110100000110111100011010111111111011000100001010100010
31  111111001010001110011000001100100011110011011010010010010111
34  101011011011101110000100111101011001100001111011011000011011
36  111101011110011101011011010010110111001011100111101110000100
37  110111000001111110111111110011111011000100100100010111101011
```

Figure 1: An initial sorted population for one max

Using unbiased uniform crossover, the ones in the positions in which the two parents differ will be binomially distributed over the two children in each crossover operation. The variance is large and so initially ones are quickly removed from the low end of the population, and zeroes from the top. (For a complete formal analysis, one must take into account the population size when more than one mating is considered. Such an analysis is beyond the scope of this paper.)

With uniform crossover biased at $p = 0.05$, the children will differ on average by only 1.2 characters from their parents even if we assume the parents are complements! With only one or two of the ones being switched per mating, it takes longer for the optimal to be found. The faster production of the maximum and minimum by unbiased uniform crossover is due to the

higher variance it has.

However, when the population is nearly sorted, with only one or two 1's in a string of 0's for example, the probability that exactly those two characters will be chosen by unbiased crossover is very small. Such a population, produced using the biased uniform crossover, is shown in figure 2. Unbiased crossover is unlikely to obtain even this much separation of zeroes and ones in a reasonable time. Out of the 2^{60} equally probable crossover possibilities (using unbiased crossover) on the strings valued 3 and 57, only 14 will yield an improvement.

Biased uniform on the other hand has a reasonable probability of selecting only one character as a crossover, and so one by one the remaining ones are sorted from the strings of zeroes over a few thousand evaluations. For optimization it is not reasonable to require the complete sorting of the population. Nevertheless, these arguments indicate that biased crossover may work better with very small populations.

```
0    000000000000000000000000000000000000000000000000000000000000
0    000000000000000000000000000000000000000000000000000000000000
0    000000000000000000000000000000000000000000000000000000000000
0    000000000000000000000000000000000000000000000000000000000000
3    000001000000010000000000000010000000000000000000000000000000
57   111110111111101111111111111101111111111111111111111111111111
60   111111111111111111111111111111111111111111111111111111111111
60   111111111111111111111111111111111111111111111111111111111111
60   111111111111111111111111111111111111111111111111111111111111
60   111111111111111111111111111111111111111111111111111111111111
```

Figure 2: A sorted population for one max after 3000 matings by biased uniform

Notice that GIGA works by moving the ones through the population over time, instead of evolving toward populations containing many ones.

To understand the difficulties with one point crossover, we need to look at the space over which the algorithm is searching. The *search space of the problem* is the set of strings and their values that can be formed from the alphabet. Notice that we are assuming a fixed length string. The values are assigned by the black box environment. The encoding then is part of the problem space definition.

The *search space structure* (SSS) imposed by an *algorithm* is a directed graph where the vertices represent state information and an arc (x, y) means there is some value that can be assigned to the vertices x and y such that there is a possibility of the algorithm changing to state y from state x. For GAs, the vertices are populations, and two populations are connected if there is some genetic operation that will produce the second population from the first. The precise definition will depend upon the specific GA being implemented. For GIGA as described in this paper, there will be an arc if exactly two members differ, and the pair in y can be produced by a crossover operation on the pair in x. Again the precise definition will depend on the options chosen for the program.

The *search space configuration* (SSC) imposed by an algorithm for a specific problem will be the substructure allowed by the algorithm for the specific values of the function. For example, using GIGA, once values are assigned to the strings only those strings which are adjacent when sorted (assuming sorting is maintained) will be able to participate in crossover. Furthermore, depending on family size and whether elitism is selected etc. it might not be possible to select certain child pairs. To be completely precise, we would also need to take into account state information indicating which pair in the population is next slated for mating. We will not formalize these notions in this paper.

Arcs will have some probability (weight) associated with them indicating the probability of the particular transition, and vertices will take on the value of the maximum (or best) string in the population. The weights may depend on the algorithm and the problem values.

We restrict our attention to a population of size two. The motivation is that the local search space structure surrounding a pair of parents is in fact a search space on a population of size two.

We define the *Hamming closure* by saying $\mathbf{C}$ is in the Hamming closure of $\mathbf{P}_1$ and $\mathbf{P}_2$ if for each $1 \leq j \leq l$ either $\mathbf{C}_j = \mathbf{P}_{1j}$ or $\mathbf{C}_j = \mathbf{P}_{2j}$. If the Hamming distance between parents is k, then the Hamming closure has 2^k strings and the search space has 2^{k-1} vertices. These numbers are correct even if $\alpha > 2$ for the local space of two parents. The vertices are pairs of strings complementary with respect to the Hamming closure of the parents, and thus we will consider only complementary strings. (This ignores the probability effects induced by variations in the number of characters between positions of differing character values. For example, if $l > k$, then there may be several cut points between some of the characters, and only one between others. This could have probabilistic consequences if we are using one point

crossover.)

Uniform crossover forms a complete graph on the vertices. If it is unbiased then every point is equally likely to be generated, resulting in a random search. Otherwise the points at smaller Hamming distance from the parents have higher probability of being generated, and thus the search is biased.

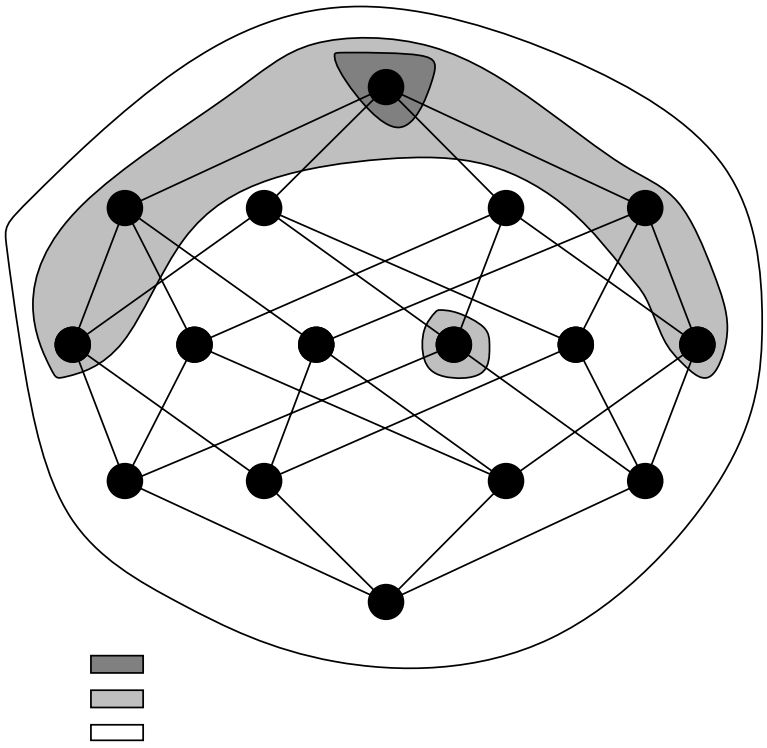

Figure 3: One Point Crossover Search Space on Ones Counting

One point crossover can only select from a number of points equal to the Hamming distance between the parents. The local SSS is a hypercube of dimension $k-1$, but the connections for the most part do not correspond to the Hamming hypercube.[3]

[3]A simple isomorphism demonstrating the hypercube can be constructed; select any $k-1$ bit string to map to an arbitrary point in the crossover space. For a neighbor generated by a crossover at position i, flip the ith bit of the string. This suggests a simple

Figure 3 illustrates the SSS for a pair of strings with $k = 5$ under one point crossover. Since crossover is symmetric, we show undirected edges. The contours group vertices of equal value under the ones counting function. Notice that for the point 00100/11011, which is of value 4, all edges lead to lower valued points. As a result, using elitism and mating repeatedly on this pair would never lead to an optimal value. For larger k, such false peaks become much more numerous, and dominate the probable termination of the process. For $l = 60$ and $n = 2$ it is extremely unlikely we would find the optimal string.

When n is large enough, these false peaks in crossover space may be eliminated. As the population becomes large, sorting the strings will lead to adjacent strings having greater similarity. The similarity increases as the ones are sorted from the zeroes under sequences of matings. This explains why the larger population made it easier for GIGA to find optimal values using one point crossover in experiment **1a**.

However, if we wish to completely separate the zeroes from the ones even a larger population does not help. Eventually we must sort the characters from the last two strings that are not pure zeroes or ones (if we do not stop at a peak with even more strings). For example, our population could look like that in figure 2. No one point crossover can improve that population. For $l = 60$ the probability is high that GIGA with elitism will never be able to complete the separation using one point crossover.

Several points should be made about this analysis. First, although it may seem that one max is a trivial function, it in fact models very nicely the requirements of propagating short building blocks of nearly equal value. When one point crossover is used on functions which have such building blocks, crossover points that break the blocks lead to low valued offspring that are eliminated by the replacement selection process. Ignoring these useless crossover attempts, the blocks are recombined by one point crossover in the same way as characters are in the one max function. Similar false peaks can occur. One point crossover may be superior to uniform when we are dealing with problems involving combination of long substrings. An analysis of such a case is presented in section 3.3.

Second, the search structure is independent of the value assignment. Thus, the search will succeed or not depending on whether values form too

way of converting deceptive functions for Hamming hill climbers into ones for one point crossover systems. Similarly, easy functions for hill climbing can be converted into easy functions for crossover.

	Majority Experiments			
	1	1a	2	3
Maximum	10516*	16331*	5001	4958

Table 2: Average evaluations on the majority function.

many false peaks in this structuring of the space. There are many functions which could be defined to create false peaks on one point crossover.

Third, using a two point crossover (divide the strings into three segments, and swap the middle segment) the one max function is no longer deceptive, because we can always select out a single bit for crossover if necessary. This suggests that two point crossover might be better for the combination of building blocks when there are many of them in a string. However, if all blocks do not have equal values, then there will exist functions for which 2-point crossover is also deceived.

Fourth, unbiased uniform crossover on a population of size two cannot have false peaks, since it is a random search. However, when the population is enlarged, the strings being mated are not complementary pairs. As a result, the search will tend to search some regions with higher probability than others. Thus, deception should be possible here as well. A full analysis is needed.

⋆ ⋆ ⋆ ⋆ ⋆

Experiments **major.1, major.1a, major.2** and **major.3** optimize the majority function, using one point crossover with populations of size 10 and 20, unbiased uniform and uniform at $p = 0.05$ respectively. Other parameters including string length are as in the one max experiments. Finding the minimum, which occurs for equal numbers of 0's and 1's in the string, is trivial for GIGA. The number of evaluations required to find the maximum value are shown for the three experiments in table 2.

It must be pointed out that one point crossover failed 45 out of 50 times to find the maximum on population size 10 and 31 times out of 50 with population of 20 (starred entries). Unbiased uniform does not seem to have any advantage over the biased operator. The reader is encouraged to determine why these results should occur.

For linear functions and many others replacing the parents with the pair which has the maximal maximum, minimal minimum or maximum difference in value yield equivalent behavior using GIGA [3]. Notice that for the majority function, the three definitions are not equivalent. What happens when we use the maximum difference definition? The answer may surprise you!

For the final simple function, we consider the binary number function; that is the function which treats the string as a binary number. In this function different positions contribute widely different values. Initially, the leading bits will have the most effect on sorting, sometimes carrying low order bits to the opposite end of the population from where they are required. The low order bits will then be found and moved through the population after the high order bits have been selected. The opposite effect occurs at the low end of the population. The visual effect seen on running the experiment **binary** is the formation of triangular regions in the population. At the low end we have leading zeroes, while at the high end we have leading ones. The number of identical leading characters diminishes towards the center. The experiment uses one point crossover, but the effects are very similar using other crossover operators. The family size should be kept small to maximize the delay of the effect.

A sample population taken after 1100 matings is shown in figure 4 for those who do not have access to the program.

A deterministic search on binary numbers can be set up using the non-repeating one point crossover. Setting the population size to two (unlike one max, the binary number function has no false peaks under one point crossover), and family size to $l-1$ no more than $l-1$ matings will be required to optimize an l bit string, yielding both minimum and maximum.

3 Deception

Goldberg [12] and others have designed functions that are deceptive to various GAs. The principle reason we might be interested in such functions is to further illuminate the means by which the algorithms succeed or fail. In this section, we examine some of the deceptive functions of the literature, and some new ones. The results show some interesting insights into the nature of GIGA and of crossover search space structures in general.

```
000000000000000000000001
000000000000000000100010
000000000000000001100000001
000000000100101110000100
000000110110011000010001
000000111010100011001001
000001000000011111000001
000001101001110000010110
000110011011111011100110
000110100101000000101100
111000001110000000111000
111001110010011101001101
111110001111001011011110
111111001011101011001110
111111010011111100011011
111111110100100101111011
111111111101001011111010
111111111011101111111101
111111111111111011111111
111111111111111111111110
```

Figure 4: A sample population from the binary number function

3.1 Standard Deceptive Functions

Experiment **goldberg.1** uses 10 replications of Goldberg's [13] 3-bit deception on 50 runs. It uses a population size of 10, family size of 5, one point crossover and succeeds 50 out of 50 times with an average of 2269 evaluations. This compares to 20960 as the best result from [10]. Note this is the tightly coupled version of the deceptive problem. In this version, one point crossover can recombine building blocks and fairly quickly they migrate to the top.

However, if the loose version is used (where the first set of 3 bits occupy positions 0,10,20 etc.) or uniform crossover is used, then it is very difficult to propagate the substrings that are required. With populations of size 75-100 and long runs (from 40,000 to 160,000 evaluations) the problem can usually be solved, but this is not very competitive. Some improvement can be had by using mixed crossover with about 2/3 uniform at $p = 0.50$ and 1/3 of 5-point crossover. The 5-point crossover has some ability to pick out small subsets of bits that are not adjacent, but the probability is quite low. Experiment **goldberg.2** solves 10 out of 10 cases, but requires an average of 107,070 evaluations.

The problem is that the loosely ordered form is less helpful to crossover search. Recombining building blocks is difficult with the usual crossover operators if the blocks consist of non-contiguous characters. Perhaps we could design crossover operators that assume non-contiguous sequences as counterparts to one point crossover. These might be able to take advantage of well defined properties that the usual crossover operators cannot.

Experiment **liepin.1** uses 10 replications of Liepin's [19, 10] 5-bit deception (also tightly coupled) on 50 runs. It uses a population of 10, family size of 10, one point crossover and succeeds 50 out of 50 times with an average of 6842 evaluations. This compares to 11230 as the best result reported by Eshelman [10]. Since this does not seem too difficult, experiment **liepin.2** uses a 10-bit version of Liepin's function. This experiment uses a mixed crossover operator. This succeeds on 49 out of 50 trials, using an average of 24854 evaluations on the successes.

It should be noted that neither of these functions was designed to be deceptive to GIGA. However, they are designed based on ideas of what should be deceptive to a GA, and GIGAs are very special GAs. One thing that makes them susceptible to GIGA is that selecting the characters needed for the suboptimal point leaves exactly those characters required for the optimal point. Thus, the building blocks are created towards the low valued

end of the population, then propagated up to the top.

3.2 Deceiving GIGA

So how do we build a deceptive function for GIGA? Clearly, if we create a single spike on one point in the problem space and assign every other substring a value of zero, then neither GIGA nor any other randomized algorithm will be able to find the point easily. Similarly, on a randomly generated function, even if all strings have distinct values, the best we can do is a random search taking exponential time on average. However, such functions are of little interest in determining the boundaries of applicability of a technique.

We want a function that exhibits a high degree of structure; one that does not rely on shuffling bits so that crossover is unable to build blocks, but rather uses contiguous blocks so that if blocks become available they can be combined. We want a function which definitely misleads GIGA, rather than one that just loses it in a sea of randomness.

The usual deceptive functions presented and tested in the preceding subsection have the property that the optimal point is diametrically opposite the suboptimal deceptive point in Hamming space. GIGA keeps all of the original characters, and every character is guaranteed to appear in every column of $\mathbf{P}_{ij}$. The ones collect at the high end of the population and the zeroes are left to form optimal building blocks elsewhere. Once formed, GIGA then propagates them upwards through the population.

Several more severe functions of this type were created for GIGA using multi-character alphabets and longer substrings. We refer to these functions as f_{d1} through f_{d4}. None of these proved particularly deceptive in practice, but observing GIGA's performance illustrates some interesting properties of the program.

The functions each divide the string into substrings of size α; i.e. each substring has length equal to the alphabet size. The substring of all zeroes is assigned a value of $\alpha+1$. Any other substring is assigned a value according to how many non-zero characters it has. Let t be the number of zero characters, i.e. $t = \|\{x_i = 0\}\|$, where x is the substring of characters. Then

$$f_{d1}(x) = \begin{cases} \alpha + 1 & t = \alpha \\ \alpha - t & t < \alpha/2 \\ 0 & \text{otherwise} \end{cases}$$

Suppose we visualize the Hamming space of this function over an alphabet of size α by pretending the space has been flattened into concentric rings, with the optimal point at center, and points at Hamming distance i in the ith ring.[4] We let height above the circular display represent the value of the substring. Then the problem space is a bowl with stepped sides and a single spike at the center. The more nearly correct the string is, the further down the slope towards the center it lies. If more than 1/2 of the characters are zeroes, the value is zero. The idea was to create a large area near the optimal point where GIGA must do a random search.

For our experiments we used a concatenation of five subproblems. The five character version required an average of 7117 evaluations averaged over ten runs. The 10 character version required 66585 evaluations averaged over 50 runs with no failures to find the maximum. The input for $\alpha = 10$ is in the file **decept.d1**. An experiment using $\alpha = 15$ can be found in the file **decept.d1.15**. The population size is 75, $l = 75$ so that there are five substrings, family size is five and one point crossover is used. The initial population has a minimum value of 65 and a maximum value of 75, which means that at least one string has all 15 characters non-zero in all five substrings. This latter would occur with probability of 0.35 if all characters were chosen independently although the random rotation used would alter this probability a bit. The population is kept sorted by value.

We trace the search process here, but the reader is encouraged to run the experiment and print every 1000th population to obtain a complete picture of the process. The first string value of zero occurs after 16145 evaluations, which means that every substring in some string has at least 7 zero characters. This string is produced as a result of the concentration of non-zero characters in the higher valued part of the population. As the search continues more zero valued strings are produced, while the great majority of strings obtain values of 75. After some time the first substring of all zeroes is produced from the mating of two strings with at least $\alpha/2$ zeroes. When a zero substring combines with a string with no zeroes so that the substring is preserved, its value increases and so the string is sorted to the high end of the population. Additional substrings of zeroes are formed

[4] To be representative, the rings would have to increase rapidly in size. Suppose the center point is a circle one millimeter in radius. For $\alpha = 10$ the distance five ring would be about 2.8 meters wide, and the outer ring would have to be about 19.2 meters wide, with outer boundary at 100 meters in order for the *areas* to be roughly representative. For $\alpha = 15$ the outer boundary would be 661.7 kilometers away from the center and the outer ring would be 130.3 kilometers wide.

fairly quickly. The maximum string value increases by one after 111145, 139515, 143895, 244635 and 326395 evaluations. Each of these increases means that one more substring of pure zeroes has been combined into the top valued string. The last increase achieves the optimal value.

If we let `N` represent a substring of non-zero characters, and `Z` represent a substring of zeroes, then at one point in this experiment the top three strings look like

```
ZNNNZ
ZZNZN
ZZZNZ
```

The next lower string has no zero substring. The value of `N` is 15 and that of `Z` is 16. Any cross that mixes a pair of substrings cannot lead to an improved value. Thus, using one point crossover only crossover points between substrings can improve the result. But, except for the probability of selecting the appropriate crossover point, this looks like one point crossover on the one max function. We see that no crossover on the last pair of strings will lead to a higher value, although there are two points that would yield equal value. The lower valued pair (at the top) could be combined to produce the string `ZZNZZ` and then the top two could be combined to produce the optimal. In fact, more zero strings are actually produced before the program finally produces an optimal string in this run. As the top valued string obtains more zero substrings, the difficulty of the recombination of these blocks increases. Two point crossover would likely be more efficient than one point if there were more replications of the problem.

Notice that uniform crossover would be extremely unlikely to produce the desired recombination effects regardless of the bias. This low probability of combining building blocks is usually referred to as its disruptive effect. Uniform crossover might however produce the first zero substrings more quickly. Considerations such as this were the reason for allowing more than one type of crossover to be used in the program.

Clearly, a Hamming hill climber will fail miserably on this function, since almost all changes will lead toward the plateau of maximum error. Only substrings within Hamming distance strictly less than $\alpha/2$ of the spike have any chance of being converted to the optimal. A GA should also have difficulty, because the likelyhood is great that the optimal substrings will not be in the initial population, or even constructible from it after only a few generations unless very high rates of mutation are used. After a few generations the population will be converging around one or more points

on the false plateau, and then the probability of getting off of this plateau, across the depression and hitting the spike is very small.

The reader is encouraged to turn off the sorting feature, and print out intermediate populations to see where and how the substrings are formed, and to watch their progress and combinations through the population. The user will notice that elitism is very important to maintenance of these optimal blocks. Also, experimenting with uniform crossover will quickly show it to be significantly inferior on this problem, because it finds it very difficult to propagate and combine these long substrings as units in the population.

Function f_{d2} was an attempt to make the problem harder for GIGA. In this case, substrings with more than 3/4 of their characters correct (i.e. zeroes) were assigned increasing values as the number of zero characters increased. That is,

$$f_{d2}(x) = \begin{cases} \alpha + 1 & t = \alpha \\ \alpha - t & t < \alpha/2 \\ t - \frac{3}{4}\alpha & \frac{3}{4}\alpha < t < \alpha \\ 0 & \text{otherwise} \end{cases}$$

The idea was that these strings would then move up from the bottom of the population, where they would tend to cross with the median value strings which have more non-zero characters. This might make it harder for GIGA to chance upon optimal substrings. Instead, this function is if anything more readily solved by GIGA, taking an average of only 60421 evaluations. Because they get values intermediate to the outer ring and the bottom, the strings containing these nearly optimal substrings cluster together in the middle of the population. Since GIGA mates all adjacent pairs equally often, putting the strings with higher proportions of zero characters in the middle does not hide them. In fact, it separates them from the strings with as few as $\alpha/2$ zero characters, and tends to more quickly concentrate zeroes. The higher proportion of zero characters in these matings yield higher rates of optimal substrings.

Function f_{d3} is the same as f_{d1}, except that zero is assigned to any string with up to two thirds of its characters non-zero. This means that there will be many more substrings with erroneous characters at the bottom of the population. Again, GIGA was able to defeat the supposed deceptive nature of the function handily, although the search time did increase somewhat.

At this time frustration (of sorts) led to the creation of function f_{d4}. In this function, only the optimal substring and the substrings in which all characters are non-zero get non-zero values. This puts the majority of

strings on the plane with value zero. It would seem this very large plane should cause GIGA to degenerate at best to something akin to random search. Although it does slow down somewhat, GIGA can still find the optimal strings at $\alpha = 10$ using an average of 244440 evaluations.

How does it do it?

Notice that we are using random rotation to initialize the population. Suppose for the sake of analysis, $\alpha = 10$, $l = 10$ and population size $n = 10$. Now think of the population as a (dynamically changing) 10 by 10 matrix, where each row is one string member of the population. Random rotation ensures that each character appears once in each column of the matrix, and genetic invariance ensures this remains true throughout the program run. When the population is initialized, 9/10 of the characters in any string (on average) are non-zero. With just a few crossovers, we will with high probability get a string with all characters non-zero. Due to elitism, there is a sense in which this string can never be lost. Namely, there will always be a completely non-zero string in the population after one is first found.

Thus, each column has effectively been reduced from 9 non-zero characters and one zero, to 8 non-zero characters and one zero. Notice we do not care which 8 non-zero characters are in the remaining strings at any time, only the number of them. The probability of selecting a non-zero character is now 8/9, which is smaller than before, but still with recombination we quickly get another non-zero string. Now the population will always have (at least) two completely non-zero strings. This reduces the ratio of available characters to 7/8. Repeating this we eventually have just two strings that are not completely non-zero. Since there are only two characters per position that can be recombined (the other characters are locked up in the higher valued completely wrong strings) there are only $2^{10} = 1024$ strings to search through for the one which yields the last non-zero string and perforce at the same stroke the optimal string. Of course, the search through this subspace is essentially random, but it is small enough that is not a serious impediment to the programs average behavior. We would need to mate the bottom strings on average about 1024 times(varying slightly according to what crossover operator we are using) until we form the optimal string. Since we have 9 pairs in the population to mate, we allocate about 1/9 of the matings to the bottom pair, and so we would require 9200 matings before we find the optimal on average, after reaching the state where we have two zero-valued strings.

Increasing the population and string size do not substantially change the nature of this analysis, other than the statistical values and the need to

combine substrings into the optimal string. Increasing the alphabet size (and at the same time the block size) rapidly increases the difficulty. For example, at $k = 15$, the last string pair entails a search space of 65536 combinations, to be searched at random. Much longer search times are required.

Function f_{d5} is the first attempt at a function which has its suboptimal point not at the opposite pole in Hamming space from the optimal. Again letting t be the number of zero characters

$$f_{d5}(x) = \begin{cases} \alpha + 1 & t = \alpha \\ t & t < \alpha/2 \\ \alpha - t & \alpha/2 \leq t < \alpha \end{cases}$$

As GIGA runs on this function, the best valued strings tend to have $\alpha/2$ zero characters. These half-correct strings are concentrated at the top of the population. Suppose we have $n = \alpha$. Then mating the top two strings yields the optimal string with probability of 0.5^{α}, assuming unbiased uniform crossover and that each character occurrs once in each column of the population. This means the function is not completely deceptive, and experiments bear this out, although it is quite difficult and one point crossover is not adequate. Turning elitism off seems to help form blocks faster, but then they are lost before they can be combined into an optimal string. A full analysis needs to be done on GIGAs behavior on this function.

If we choose instead to ramp up to $\alpha/3$, then the algorithm will be led towards strings with 1/3 zero characters. Function f_{d6} implements this

$$f_{d6}(x) = \begin{cases} \alpha + 1 & t = \alpha \\ t & t < \alpha/3 \\ \alpha - t & \alpha/3 \leq t < \alpha \end{cases}$$

If all of the top strings have only approximately one third zero characters, and all of the zero characters in the population tend to gather in the top strings, then no mating can produce the optimal string. This suggests this function is unlikely to be solved by GIGA. Experiments confirm this analysis.

3.3 Ancillary Analysis

For the deceptive functions f_{d1} to f_{d4} notice that if we had been using a binary code for the characters, and bitwise crossover, then it is likely that we could have destroyed the capability of the algorithm to find the optimal. The reason is that by rearranging the bits, we would have created more

non-zero characters, and destroyed the zeroes. We then would have had all strings situated on the plateau and no further progress would have been possible. This argues in favor of more natural representations instead of binary encodings.

Selecting for minimal values, instead of maximal, would likely have severely hampered the algorithm on this function, even though we know the maximum point is located in the center of a depressed plane. It was only by removing the non-zero characters, accomplished by selecting for maximal values, that the search space was constricted enough to have any hope of locating the maximum. It appears the best search is one that examines interesting features. If we concentrated entirely on the lower plane, we are left wandering at random. If we allow the search to examine and select for the suboptimal points, then eventually the remaining strings form a small enough subspace to be interesting. Searching this space finds interesting points in it. One wonders whether it would be possible to have more than two levels of such emergent "interestingness" and developing behavioral changes over time.

If we were using uniform crossover, then it is possible that after achieving the reduction to two zero-valued strings we would generate an optimal substring more quickly. The reason is that given a pair of strings, we would generate each point in the space with equal probability on every attempt. In particular, we search all of the substrings at the same time with uniform crossover, while one point can only mix at most one substring from the parents. The point is that at this stage we want to move characters around at random, because *we do not yet have the building blocks.* Without building blocks, or indications of ascent, we can do no more than a random search, and uniform crossover gives us such a search.

On the other hand once an optimal substring is found, it moves to the top of the population where it must combine with other substrings from other strings. Here, one point (or perhaps two or three point) crossover is clearly superior to uniform. Uniform (at $p = 0.5$) would correctly combine two ten character blocks into one string with probability of 2^{-20}, while one point only needs to select the cut point between them. This has probability $1/l$ if the two blocks are in adjacent positions, and much larger if the blocks are separated.

Similar comments apply to many other functions, including some of the De Jong functions. At certain times, we wish to recombine strings to form building blocks, and at other times we need to combine the blocks into strings. Building the blocks is essentially a random search (although the

probabilities may be strongly biased as when we first start on function f_{d4} to build suboptimal non-zero blocks) while putting the blocks together requires that certain points be made more accessible to the search than others.

These arguments strongly suggest the need for research into the possibility of fluctuating crossover operations, that sometimes use one point and sometimes uniform or biased uniform or multipoint crossover. It was after observing the results from this function that the mixed crossover option was installed. This particular implementation is ad hoc, but the following shows just how effective it can be sometimes. For a variant of function f_{d4} in which optimal blocks are assigned a value of 1000 with $\alpha = 10$, the first optimal block was found after 19330 evaluations using 1/2 unbiased uniform and 1/2 one point crossover, while one point alone required 68230 evaluations for the same experimental setup. The complete solution required 171439 evaluations using mixed crossover while 249180 were required by one point.

4 De Jong Test Suite

In this section we test GIGA on the De Jong test suite functions f_1 to f_5 [8, 12]. One of the hazards in making comparisons between different algorithm design paradigms is that in the desire to make our favorite paradigm win, we may add so many target specific options and variations that the results are not valid as support for the superiority of the general design. On the other hand, if we compare a purely basic system to results that have had years of tuning and additions of specialized variations, the results are likely to look so bad in comparison that they may not reflect the potential of the approach.

In this section, I try to take a balanced approach. In some instances, considerable tuning has been done to get excellent results. Alternatively, some of the tests have been performed using a single parameter setting across all five functions, and the results compared to the default settings (and some improved settings) of the well known GENESIS system[15]. Some enhancements, such as Gray coding, have been tried and commented on. Others, such as mutation which would violate the invariance property, have been deliberately avoided, although it is likely they would afford further improvement.

One purpose of this section is to show that the basic design of GIGA has at least as much potential as the traditional GA design. Additional

Function	TGA	NE	CHC
f_1	2323	805	1089
f_2	16394	9201	9065
f_3	2381	1270	1169
f_5	4379	1719	1396

Table 3: Evaluations to Optimization from Eshelman and Schaffer

observations will be made as the experiments are described.

The De Jong test suite has been used as a basis for comparison of GA techniques and other techniques for several years. It has recently been the center of some controversy because it seems that stochastic hill climbing, naive evolution (i.e. a GA with no crossover) and other techniques may out perform the TGA on most of these functions[11].

We cite in table 3 results for functions f_1, f_2, f_3, f_5 from Eshelman and Schaffer [11] for two genetic algorithms and naive evolution. TGA used proportional selection and the individual elitist strategy, a population size of 30, mutation rate of 0.01 and two point crossover at a rate of 0.95. It represents a high dependence on crossover in the traditional GA vein. CHC was a non-traditional GA that used cross generational elitist selection, uniform crossover and restarts on population convergence [10]. NE uses a traditional GA, but no crossover, a population of size 10 and mutation rate of 0.023. Thus, it represents the other end of the scale, reliance on stochastic hill climbing. Table 3 provides the number of evaluations required to find an optimal solution.

One of the concerns raised by the numbers in table 3 is that crossover may be outperformed by other search operators. Another is a concern that GAs may not be the superior method they are often considered to be by the GA community.

We will use results from GIGA to argue that for f_1 and f_3, crossover is in fact very effective since we will do slightly better than TGA on f_1 and as well as CHC on f_3 using *only* crossover as a genetic operator. We note that TGA uses a mutation rate of 0.01 and the NE results indicate that without mutation TGA would have even poorer performance. Thus our use of crossover in GIGA seems more effective than its use by TGA on these two functions.

First we consider f_1. The GIGA experiment, averaged over fifty runs,

uses a population of size 6, a mixed crossover strategy of 3/113 uniform crossover biased at $p = 0.04$, 80/113 two point crossover, and 30/113 one point crossover. The setup can be found in file **dejong.sp1**. It is highly tuned, and is quite sensitive to population size, at least up to a few hundred evaluations on average. It succeeded 50 out of 50 times with an average of 2074 evaluations to find the minimum compared to 2323 for TGA.

The GIGA experiment for function f_3 is in the file **dejong.sp3** and uses a population of size 15 with unbiased uniform crossover. It solves the problem 50 out of 50 times, using an average of 1155 evaluations. This problem seemed so easy that I did not waste much time trying to optimize further. This is already sufficient to support the claim stated above.

For the remainder of our comparison, we will use the GENESIS system with the default setup parameters, plus some modified parameters we use when the GENESIS defaults seem particularly bad. In files **dejong.1** through **dejong.5** [2] are experiments over ten runs each for each of the five De Jong functions. Except for string length, which is defined by the functions, all five experiments use the same parameter settings. In each case a population of size 20, family size of 2, elitism and mixed crossover with 4/10 unbiased uniform, 4/10 one point and 2/10 two point crossover was used. Little tuning was done in selecting these parameters, and that almost exclusively on function f_1. In each case the results nearly equal or exceed the results of GENESIS using the default setup parameters, where the comparison is based on the values achieved after an equal number of evaluations.

Of course, the default parameters GENESIS uses are not always particularly good, and so for some of the functions other parameter settings were tried. For function f_2 population was set to 30, crossover rate to 11.4, and mutation to 0.01, with the other parameters left at the default values. Initially the GA seemed to be a bit faster than GIGA, but after 4000 evaluations GIGA using the standard setup above had a value comparable to that of the GA (0.0051 compared to 0.0057 for the GA). The GA then leveled off while GIGA continued to decline to 0.0027 at 20000 evaluations. Note that the default mutation for the GA was 0.002 and thus under the assumptions based on table 3 we have given a significant benefit to the GA. Tuning GIGA a little bit, as in file **dejong.sp2** (the settings are described on the following page) we can get averages over 50 runs as low as 0.001 after 20000 evaluations, with continuing declines to 0.000089. However, even with these settings the optimal value is achieved only twelve times out of fifty in 25000 matings.

Similarly, for function f_4, changing the default settings for GENESIS to a population of size 30 (from 30,000), with crossover rate of 24 and mutation rate of 0.01 gives much better results, which are comparable to those of the standard GIGA experiment.

The standard GIGA input on f_5 reached a value of 0.998004 (which is as small as it gets using only six digits of output accuracy) in 3220 evaluations, while the default GENESIS had reached only 1.1 (averaged over 10 trials) and never went below 1.0 in 10000 trials. Changing the population to 30 from 200 and increasing the mutation rate did not improve the results for the GA. Little further research has been done on this function as of this writing.

Some of the numbers from the experiments (using the improved GENESIS settings) are presented in table 4. All experiments were over 10 runs. The values cited for GENESIS are taken (from the column headed "Best") from the nearest number of trials to the cited number of evaluations, as GENESIS prints out values at irregular intervals. The number of evaluations from GIGA are computed by the formula given in section 1, i.e. $2sm + n$. This is a tight upper bound on the actual number of evaluations used to produce the average values cited.

When we consider the preliminary nature of the research into GIGA, these results are quite intriguing. As we suggest throughout this paper, and in the accompanying document [3], there are many possible improvements that could be made to GIGA.

For the De Jong functions, one claimed improvement in GAs is the use of Gray codes. Gray codes reduce the effect of Hamming cliffs for GAs. GIGA must move material upward (or downward) over a population, and so we would intuitively expect that GIGA would be even more susceptible to Hamming cliffs than the GA, *if the cliffs tend to thwart crossover*. The supposition is that if two adjacent values in the population have widely different character strings, then crossing them is unlikely to allow any improvement. This is not strictly true of course, and one need only consider the effects of crossing 1000 and 0111 when evaluated as binary numbers to see that such cliffs need not be a barrier to crossover in either GIGA or GAs. Notice that for mutation the point 0111 definitely acts as a barrier to further progress, in that only mutating the lead bit can yield improvement. On the other hand 1000 responds favorably to mutation almost everywhere.

We now examine a sample population of GIGA from f_2 after it has nearly found an optimal string. In figure 5 we print a partial population and its evaluation, and have separated the two substrings used in the function

Function	Evaluations	GIGA	GENESIS
f_1	2020	0.00357	0.00407
	4020	0.00031	0.00021
	6020	0.00005	0.00002
	8020	0.00001	0
	10020	0	0
f_2	4020	0.0051	0.0057
	8020	0.0036	0.00568
	12020	0.0029	0.00568
	16020	0.00277	0.00568
	20020	0.00269	0.00568
f_3	2020	0.4	0.1
	2820	0.0	0.0
f_4	4820	3.305	0.835
	10020	-0.066	-0.231
	14820	-0.711	-0.785
	20020	-1.443	-0.964
f_5	2020	0.998021	1.30190
	4020	0.998004	1.10097

Table 4: De Jong function average values using GIGA and GENESIS

10000 Matings

```
0.000001        101111100111 101111100110
0.000004        101111101010 101111101100
0.000247        101111011101 101111010001
0.000261        101111110110 110000000101
0.000596        101111010100 101110111111
0.000961        101111001001 101110101011
0.001346        110000001100 110000110010
```

15000 Matings

```
0.000001        101111100111 101111100110
0.000004        101111101010 101111101100
0.000217        101111011100 101111010001
0.000285        101111110111 110000000111
0.000596        101111010100 101110111111
0.001053        101111001001 101110101010
0.001346        110000001100 110000110010
```

20000 Matings

```
0.000000        101111101000 101111101000
0.000001        101111100111 101111100110
0.000165        101111011111 101111010111
0.000261        101111110110 110000000101
0.000596        101111010100 101110111111
0.000782        101111001101 101110110010
0.001119        110000001000 110000101010
```

Figure 5: Partial Population for f_2

definition for the readers convenience. Let the two substrings have values y_0 and y_1, where the value of a substring x is $b(x)/1000 - 2.048$, and $b(x)$ is the value of x as a binary number. Function f_2 is defined as $100.0(y_0^2 - y_1)^2 + (1.0 - y_0)^2$.

The members shown are the lowest valued members taken 5000 matings apart. The population in this case has 250 members, and so each adjacent pair has received 20 or more matings between printings. The family size is 5, so this implies 100 crossover operations per pair between printings. The crossover is a mixture of 30/95 uniform with $p = 0.04$, 30/95 one point, 10/95 two point, 5/95 5 point, and 20/95 10 point[5]. A version making 50 runs can be found in **dejong.sp2**. The populations come from printing out the population of the first run of this experiment at the indicated times.

If we compare the strings with function values 0.000001 and 0.0, we see that there is a small Hamming cliff, namely the last four bits of the first substring are complements in the two values, and three bits are complements in the second substring. This suggests that the Hamming cliff argument may have some merit. However, there is an even greater difficulty. If we look at the the first two strings after either 10000 or 15000 matings, which have values 0.000001 and 0.000004, then we see that no crossover of these strings could produce the optimal string. The required bits are simply not available.

The Hamming distance between the second and third strings of the population is 10 after 10000 matings and 9 after 15000 matings. In each case, exactly two nonadjacent bits must be exchanged to produce the optimal string, while any other cross would produce a pair of strings with a higher minimal value. If we were using uniform crossover, this would imply that the expected number of evaluations would be from 512 to 1024 before the optimal string was produced. In fact, it appears that it was a mating between the second and third strings that produced the optimal.

Similar blockings occur elsewhere in the population. Notice that it is not the Hamming cliff per se that causes the problem, but rather that there are few or no bits in strings with values close to the optimal that can be exchanged to produce the optimal. On the other hand, mutation, or small stochastic hill climbing excursions might aid in generating the optimal string from the nearby strings.[6]

After this analysis, Gray code capability was added to the De Jong functions. The tests in **dejong.1** to **dejong.5** were rerun interpreting the

[5]There is nothing magic about 95, its just what the frequencies added up to.

[6]The urge to violate invariance becomes almost overpowering here.

binary strings as Gray codes. The only case in which there was improvement was in function number 2, although the test still failed to find an optimal value. In all other cases the performance degraded significantly.

These experiments and analysis give support to the suspicion that Gray coding the De Jong functions benefits the mutation operator by removing the Hamming cliffs that act as barriers to stochastic bit-wise hill climbing.

The patterns in the population above do suggest that a crossover operator that exchanges just a few bits, say one or two or three, could be useful in many situations. In fact, the use of uniform with $p = 0.04$ and the 10 point crossover are attempts to do just that. These are reasonably effective, but the number of bits in the strings makes it difficult to achieve the desired effects with these operators. However, a better crossover would focus on the positions in which the strings differ, and select exactly k bits to exchange. Crossover is a mechanism for restricting the search space to the Hamming closure of two strings. It is reasonable then to design crossover operators that focus on this space.

Tests using shifted centers [7] were also performed. Only f_1 and f_3 seemed to be significantly affected by the offsets. Function f_1 required considerably more search time, while f_3 required considerably less. The minimum value of the latter was increased due to the shift used. Little experimentation or analysis has been done as of this writing.

Finally, a test using no sorting and no elitism seemed to perform marginally better on f_2. This may indicate that it is a good idea to either mate other than adjacent pairs, or allow a bit of stochastic movement, or it may be a statistical anomaly. It showed a slight improvement throughout the search averaged over 50 runs.

5 Conclusions

The analysis in this paper has shown the need to understand not only the construction, propagation and recombination of building blocks, but also the structure of the search space constructed by the crossover operators and the mapping of function values onto that space. There seems to be little opportunity to apply the study of hyperplane sampling that underlies much of GA analysis to GIGA.

The schema theorem predicts the rapid increase of higher valued schemata under GAs. The down side of that analysis is the rapid destruction of information that might prove valuable. GIGA on the other hand has the

capability of quite bizarre behavior, that not only takes advantage of the functions in which the GA assumption holds but can also take advantage of more indirect relationships in the functions. On the down side for the current implementation of GIGA, it is sometimes hard to propagate values when the crossover space does not reflect value distribution in a helpful way.

In [14] Grefenstette extends an analysis begun in [1] and carried forward in [16]. He defines monotonic fitness functions as ones which do not change the relative orders of the elements with respect to the objective function. GIGAs as currently implemented do not use fitness functions, or rather the objective function is the fitness function, and in fact behavior is easily seen to be unaffected by such scalings. Therefore, by default GIGA uses strictly monotonic fitness.

A selection algorithm is monotonic [14] if the growth rate of a representative of the search space in the population at time t, $gr(x_i)$ is related to its value by

$$gr(x_i) \leq gr(x_j) \iff f(x_i) \leq f(x_j)$$

It is strictly monotonic if in addition

$$f(x_i) < f(x_j) \Longrightarrow gr(x_i) < gr(x_j)$$

It is easily seen that GIGA does not satisfy these definitions. In fact, it seems unlikely that any notion of increasing trials will help in the analysis of GIGA. Rather it is the recombination of material, and thereby its propagation through the population that is the secret to its success. Recombination is the key to the effective use of a population of search space instances. In this way GIGA may make better use of the recombinatorial powers of crossover than traditional GAs in many situations.

A secondary population effect is memory of previous trials, and this may be especially important if mutation is used. An argument can be made on this basis for variable sized populations, that grow as mutation is used to prevent too much loss of material. Such possibilities should be explored in future research.

It may be reasonable to combine GIGAs with GAs. Some ideas have been proposed in [18]. For example, a GIGA could be used for a while, then a GA could be used to replace individuals of lower fitness, mutation introduced, and the GIGA used for further recombination attempts. Such approaches are left for future exploration, perhaps after a better understanding of the crossover search space is obtained.

References

[1] James E. Baker. *Analysis of the Effects of Selection in Genetic Algorithms.* PhD thesis, Vanderbilt University, Department of Computer Science, 1989.

[2] Joseph Culberson. GIGA program and experiments, 1992. ftp thorhild.cs.ualberta.ca in pub/GIGA/SHAR.Z.

[3] Joseph C. Culberson. GIGA program description and operation. ftp thorhild.cs.ualberta.ca, April 1992.

[4] Lawerence Davis, editor. *Genetic Algorithms and Simulated Annealing.* Research Notes in Artificial intelligence. Morgan Kaufmann, 1987.

[5] Lawerence Davis, editor. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, 1991.

[6] Lawrence "David" Davis. Re: Gas and very fast simulated re-annealing. *GA-list: Genetic Algorithms Digest*, 5(37), December 1991. ftp ftp.aic.nrl.navy.mil.

[7] Lawrence "David" Davis. Re: Mutation, bitclimbing and test suites. *GA-list: Genetic Algorithms Digest*, 6(2), January 1992. ftp ftp.aic.nrl.navy.mil.

[8] K.A. DeJong. *Analysis of Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, The University of Michigan, 1975.

[9] Larry Eshelman. Bit-climbers and naive evolution. *GA-list: Genetic Algorithms Digest*, 5(39), December 1991. ftp ftp.aic.nrl.navy.mil.

[10] Larry J. Eshelman. The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. In Gregory J. E. Rawlins, editor, *Fondations of Genetic Algorithms (FOGA I)*, pages 265–283. Morgan Kaufmann, 1991.

[11] Larry J. Eshelman and J. David Schaffer. Re: GAs and very fast simulated re-annealing. *GA-list: Genetic Algorithms Digest*, 5(37), December 1991. ftp ftp.aic.nrl.navy.mil.

[12] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, Inc., 1989.

[13] David E. Goldberg, Bradley Korb, and Kalyanmoy Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3:493–530, 1989.

[14] John J. Grefenstette. Conditions for implicit parallelism. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms (FOGA I)*, pages 252–261. Morgan Kaufmann, 1991.

[15] John J. Grefenstette. GENESIS 1.2ucsd. Enhanced version by Nicol N. Schraudolph, 1991 Version. ftp iuvax.cs.indianna.edu in pub/alife/software/unix/GAucsd.

[16] John J. Grefenstette and James E. Baker. How genetic algorithms work: A critical look at implicit parallelism. In John J. Grefenstette, editor, *Genetic Algorithms and Their Applications: Proceedings of the Third International Conference on Genetic Algorithms*. Erlbaum, 1989.

[17] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[18] Michael Lewchuk. Genetic invariance: A new approach to genetic algorithms. Master's thesis, University of Alberta, Edmonton Alberta, April 1992. Technical Report TR 92-05 "Genetic Invariance: A New Type of Genetic Algorithm" ftp thorhild.cs.ualberta.ca.

[19] Gunar E. Liepins and Michael D. Vose. Representational issues in genetic optimization. *Journal of Experimental and Theoretical AI*, May 1991.

[20] Gregory J. E. Rawlins, editor. *Foundations of Genetic Algorithms*. Morgan Kaufmann, 1991.

[21] Bruce Rosen and Lester Ingber. Re: GAs and very fast simulated re-annealing. *GA-list: Genetic Algorithms Digest*, 5(40), December 1991. ftp ftp.aic.nrl.navy.mil.

[22] Nici Schraudolph. Re: GAs and very fast simulated re-annealing. *GA-list: Genetic Algorithms Digest*, 6(1), January 1992. ftp ftp.aic.nrl.navy.mil.