A Genetic Algorithm Tutorial

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

Abstract

This tutorial covers the canonical genetic algorithm as well as more experimental forms of genetic algorithms, including parallel island models and parallel cellular genetic algorithms. The tutorial also illustrates genetic search by hyperplane sampling. The theoretical foundations of genetic algorithms are reviewed, include the schema theorem as well as recently developed exact models of the canonical genetic algorithm.

Keywords: Genetic Algorithms, Search, Parallel Algorithms

1 Introduction

Genetic Algorithms are a family of computational models inspired by evolution. These algorithms encode a potential solution to a specific problem on a simple chromosome-like data structure and apply recombination operators to these structures so as to preserve critical information. Genetic algorithms are often viewed as function optimizers, although the range of problems to which genetic algorithms have been applied is quite broad.

An implementation of a genetic algorithm begins with a population of (typically random) chromosomes. One then evaluates these structures and allocates reproductive opportunities in such a way that those chromosomes which represent a better solution to the target problem are given more chances to "reproduce" than those chromosomes which are poorer solutions. The "goodness" of a solution is typically defined with respect to the current population.

This particular description of a genetic algorithm is intentionally abstract because in some sense, the term *genetic algorithm* has two meanings. In a strict interpretation, *the genetic algorithm* refers to a model introduced and investigated by John Holland (1975) and by students of Holland (e.g., DeJong, 1975). It is still the case that most of the existing theory for genetic algorithms applies either solely or primarily to the model introduced by Holland, as well as variations on what will be referred to in this paper as *the canonical genetic algorithm*. Recent theoretical advances in modeling genetic algorithms also apply primarily to the canonical genetic algorithm (Vose, 1993).

In a broader usage of the term, a genetic algorithm is any population-based model that uses selection and recombination operators to generate new sample points in a search space. Many genetic algorithm models have been introduced by researchers largely working from an experimental perspective. Many of these researchers are application oriented and are typically interested in genetic algorithms as optimization tools.

The goal of this tutorial is to present genetic algorithms in such a way that students new to this field can grasp the basic concepts behind genetic algorithms as they work through the tutorial. It should allow the more sophisticated reader to absorb this material with relative ease. The tutorial also covers topics, such as inversion, which have sometimes been misunderstood and misused by researchers new to the field.

The tutorial begins with a very low level discussion of optimization to both introduce basic ideas in optimization as well as basic concepts that relate to genetic algorithms. In section 2 a canonical genetic algorithm is reviewed. In section 3 the principle of hyperplane sampling is explored and some basic crossover operators are introduced. In section 4 various versions of the schema theorem are developed in a step by step fashion and other crossover operators are discussed. In section 5 binary alphabets and their effects on hyperplane sampling are considered. In section 6 a brief criticism of the schema theorem is considered and in section 7 an exact model of the genetic algorithm is developed. The last three sections of the tutorial cover alternative forms of genetic algorithms and evolutionary computational models, including specialized parallel implementations.

1.1 Encodings and Optimization Problems

Usually there are only two main components of most genetic algorithms that are problem dependent: the problem encoding and the evaluation function.

Consider a parameter optimization problem where we must optimize a set of variables either to maximize some target such as profit, or to minimize cost or some measure of error. We might view such a problem as a black box with a series of control dials representing different parameters; the only output of the black box is a value returned by an evaluation function indicating how well a particular combination of parameter settings solves the optimization problem. The goal is to set the various parameters so as to optimize some output. In more traditional terms, we wish to minimize (or maximize) some function $F(X_1, X_2, ..., X_M)$.

Most users of genetic algorithms typically are concerned with problems that are nonlinear. This also often implies that it is not possible to treat each parameter as an independent variable which can be solved in isolation from the other variables. There are interactions such that the combined effects of the parameters must be considered in order to maximize or minimize the output of the black box. In the genetic algorithm community, the interaction between variables is sometimes referred to as epistasis.

The first assumption that is typically made is that the variables representing parameters can be represented by bit strings. This means that the variables are discretized in an a priori fashion, and that the range of the discretization corresponds to some power of 2. For example, with 10 bits per parameter, we obtain a range with 1024 discrete values. If the parameters are actually continuous then this discretization is not a particular problem. This assumes, of course, that the discretization provides enough resolution to make it possible to adjust the output with the desired level of precision. It also assumes that the discretization is in some sense representative of the underlying function.

If some parameter can only take on an exact finite set of values then the coding issue becomes more difficult. For example, what if there are exactly 1200 discrete values which can be assigned to some variable X_i. We need at least 11 bits to cover this range, but this codes for a total of 2048 discrete values. The 848 unnecessary bit patterns may result in no evaluation, a default worst possible evaluation, or some parameter settings may be represented twice so that all binary strings result in a legal set of parameter values. Solving such coding problems is usually considered to be part of the design of the evaluation function.

Aside from the coding issue, the evaluation function is usually given as part of the problem description. On the other hand, developing an evaluation function can sometimes involve developing a simulation. In other cases, the evaluation may be performance based and may represent only an approximate or partial evaluation. For example, consider a control application where the system can be in any one of an exponentially large number of possible states. Assume a genetic algorithm is used to optimize some form of control strategy. In such cases, the state space must be sampled in a limited fashion and the resulting evaluation of control strategies is approximate and noisy (c.f., Fitzpatrick and Grefenstette, 1988).

The evaluation function must also be relatively fast. This is typically true for any optimization method, but it may particularly pose an issue for genetic algorithms. Since a genetic algorithm works with a population of potential solutions, it incurs the cost of evaluating this population. Furthermore, the population is replaced (all or in part) on a generational basis. The members of the population reproduce, and their offspring must then be evaluated. If it takes 1 hour to do an evaluation, then it takes over 1 year to do 10,000 evaluations. This would be approximately 50 generations for a population of only 200 strings.

1.2 How Hard is Hard?

Assuming the interaction between parameters is nonlinear, the size of the search space is related to the number of bits used in the problem encoding. For a bit string encoding of length L, the size of the search space is 2^L and forms a hypercube. The genetic algorithm samples the corners of this L-dimensional hypercube.

Generally, most test functions are at least 30 bits in length and most researchers would probably agree that larger test functions are needed. Anything much smaller represents a space which can be enumerated. (Considering for a moment that the national debt of the United States in 1993 is approximately 2^{42} dollars, 2^{30} does not sound quite so large.) Of course, the expression 2^L grows exponentially with respect to L. Consider a problem with an encoding of 400 bits. How big is the associated search space? A classic introductory textbook on Artificial Intelligence gives one characterization of a space of this size. Winston (1992:102) points out that 2^{400} is a good approximation of the effective size of the search space of possible board configurations in chess. (This assumes the effective branching factor at each possible move to be 16 and that a game is made up of 100 moves; $16^{100} = (2^4)^{100} = 2^{400}$). Winston states that this is "a ridiculously large number. In fact, if all the atoms in the universe had been computing chess moves at picosecond rates since the big bang (if any), the analysis would be just getting started."

The point is that as long as the number of "good solutions" to a problem are sparse with respect to the size of the search space, then random search or search by enumeration of a large search space is not a practical form of problem solving. On the other hand, any search other than random search imposes some bias in terms of how it looks for better solutions and where it looks in the search space. Genetic algorithms indeed introduce a particular bias in terms of what new points in the space will be sampled. Nevertheless, a genetic algorithm belongs to the class of methods known as "weak methods" in the Artificial Intelligence community because it makes relatively few assumptions about the problem that is being solved.

Of course, there are many optimization methods that have been developed in mathematics and operations research. What role do genetic algorithms play as an optimization tool? Genetic algorithms are often described as a global search method that does not use gradient information. Thus, nondifferentiable functions as well as functions with multiple local optima represent classes of problems to which genetic algorithms might be applied. Genetic algorithms, as a weak method, are robust but very general. If there exists a good specialized optimization method for a specific problem, then genetic algorithm may not be the best optimization tool for that application. On the other hand, some researchers work with hybrid algorithms that combine existing methods with genetic algorithms.

2 The Canonical Genetic Algorithm

The first step in the implementation of any genetic algorithm is to generate an initial population. In the canonical genetic algorithm each member of this population will be a binary string of length L which corresponds to the problem encoding. Each string is sometimes referred to as a "genotype" (Holland, 1975) or, alternatively, a "chromosome" (Schaffer, 1987). In most cases the initial population is generated randomly. After creating an initial population, each string is then evaluated and assigned a *fitness* value.

The notion of *evaluation* and *fitness* are sometimes used interchangeably. However, it is useful to distinguish between the *evaluation function* and the *fitness function* used by a genetic algorithm. In this tutorial, the *evaluation function,* or *objective function,* provides a measure of performance with respect to a particular set of parameters. The *fitness function* transforms that measure of performance into an allocation of reproductive opportunities. The evaluation of a string representing a set of parameters is independent of the evaluation of any other string. The fitness of that string, however, is always defined with respect to other members of the current population.

In the canonical genetic algorithm, fitness is defined by: $f_i/\bar{f}$ where f_i is the evaluation associated with string i and $\bar{f}$ is the average evaluation of all the strings in the population. Fitness can also be assigned based on a string's rank in the population (Baker, 1985; Whitley, 1989) or by sampling methods, such as tournament selection (Goldberg, 1990).

It is helpful to view the execution of the genetic algorithm as a two stage process. It starts with the *current population.* Selection is applied to the current population to create an *intermediate population.* Then recombination and mutation are applied to the intermediate population to create the *next population.* The process of going from the current population to the next population constitutes one generation in the execution of a genetic algorithm. Goldberg (1989) refers to this basic implementation as a Simple Genetic Algorithm (SGA).

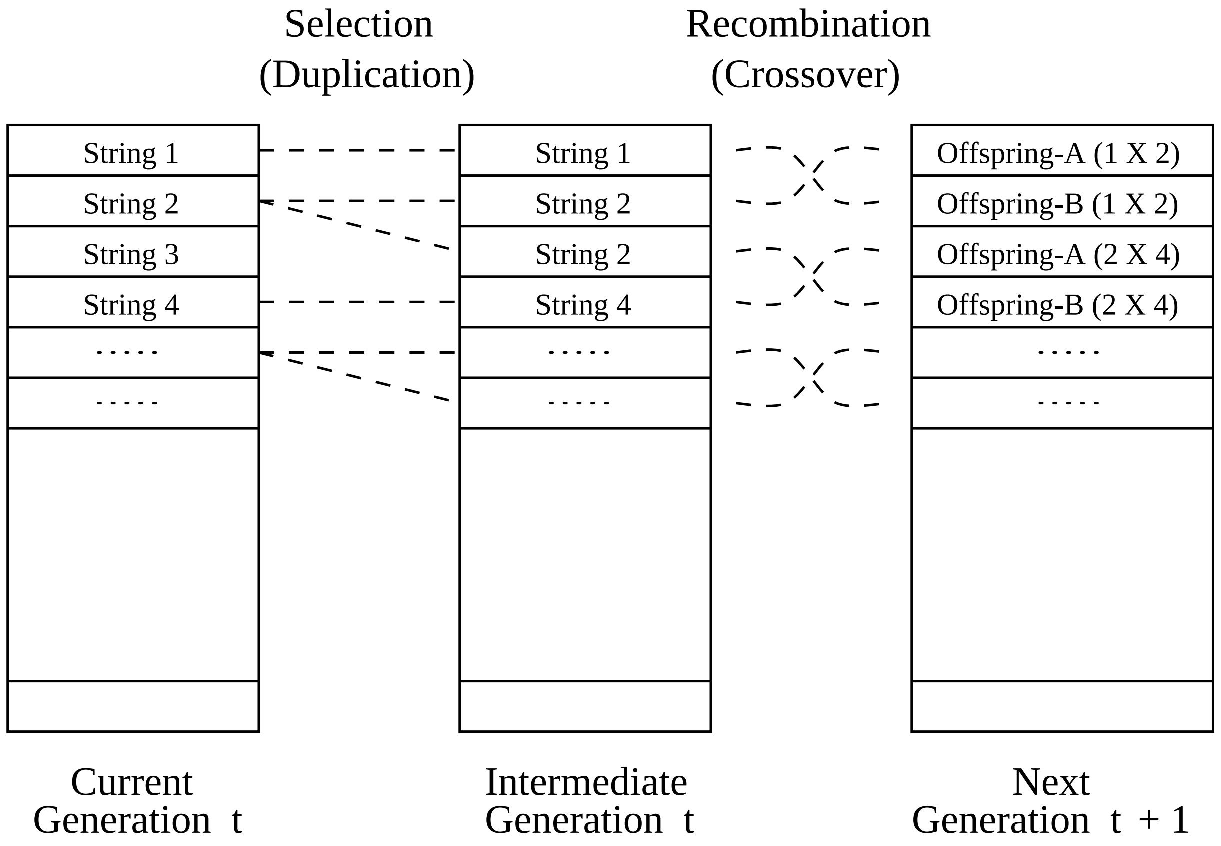

Figure 1: *One generation is broken down into a selection phase and recombination phase. This figure shows strings being assigned into adjacent slots during selection. In fact, they can be assigned slots randomly in order to shuffle the intermediate population. Mutation (not shown) can be applied after crossover.*

We will first consider the construction of the intermediate population from the current population. In the first generation the current population is also the initial population. After calculating $f_i/\bar{f}$ for all the strings in the current population, selection is carried out. In the canonical genetic algorithm the probability that strings in the current population are copied (i.e., duplicated) and placed in the intermediate generation is proportion to their fitness.

There are a number of ways to do selection. We might view the population as mapping onto a roulette wheel, where each individual is represented by a space that proportionally corresponds to its fitness. By repeatedly spinning the roulette wheel, individuals are chosen using "stochastic sampling with replacement" to fill the intermediate population.

A selection process that will more closely match the expected fitness values is "remainder stochastic sampling." For each string i where $f_i/\bar{f}$ is greater than 1.0, the integer portion of this number indicates how many copies of that string are directly placed in the intermediate population. All strings (including those with $f_i/\bar{f}$ less than 1.0) then place additional copies in the intermediate population with a probability corresponding to the fractional portion of $f_i/\bar{f}$. For example, a string with $f_i/\bar{f} = 1.36$ places 1 copy in the intermediate population, and then receives a 0.36 chance of placing a second copy. A string with a fitness of $f_i/\bar{f} = 0.54$ has a 0.54 chance of placing one string in the intermediate population.

"Remainder stochastic sampling" is most efficiently implemented using a method known as *Stochastic Universal Sampling*. Assume that the population is laid out in random order as in a pie graph, where each individual is assigned space on the pie graph in proportion to fitness. Next an outer roulette wheel is placed around the pie with N equally spaced pointers. A single spin of the roulette wheel will now simultaneously pick all N members of the intermediate population. The resulting selection is also unbiased (Baker, 1987).

After selection has been carried out the construction of the intermediate population is complete and recombination can occur. This can be viewed as creating the *next population* from the intermediate population. Crossover is applied to randomly paired strings with a probability denoted p_c. (The population should already be sufficiently shuffled by the random selection process.) Pick a pair of strings. With probability p_c "recombine" these strings to form two new strings that are inserted into the next population.

Consider the following binary string: 1101001100101101. The string would represent a possible solution to some parameter optimization problem. New sample points in the space are generated by recombining two parent strings. Consider the string 1101001100101101 and another binary string, yxyyxyxxyyyxyxxy, in which the values 0 and 1 are denoted by x and y. Using a single randomly chosen recombination point, 1-point crossover occurs as follows.

```
11010 \/ 01100101101
yxyyx /\ yxxyyyxyxxy
```

Swapping the fragments between the two parents produces the following offspring.

```
11010yxxyyyxyxxy          and          yxyyx01100101101
```

After recombination, we can apply a mutation operator. For each bit in the population, mutate with some low probability p_m. Typically the mutation rate is applied with less than 1% probability. In some cases, mutation is interpreted as randomly generating a new bit, in which case, only 50% of the time will the "mutation" actually change the bit value. In other cases, mutation is interpreted to mean actually flipping the bit. The difference is no more than an implementation detail as long as the user/reader is aware of the difference and understands that the first form of mutation produces a change in bit values only half as often as the second, and that one version of mutation is just a scaled version of the other.

After the process of selection, recombination and mutation is complete, the next population can be evaluated. The process of evaluation, selection, recombination and mutation forms one *generation* in the execution of a genetic algorithm.

2.1 Why does it work? Search Spaces as Hypercubes.

The question that most people who are new to the field of genetic algorithms ask at this point is why such a process should do anything useful. Why should one believe that this is going to result in an effective form of search or optimization?

The answer which is most widely given to explain the computational behavior of genetic algorithms came out of John Holland's work. In his classic 1975 book, *Adaptation in Natural and Artificial Systems*, Holland develops several arguments designed to explain how a "genetic plan" or "genetic algorithm" can result in complex and robust search by implicitly sampling hyperplane partitions of a search space.

Perhaps the best way to understand how a genetic algorithm can sample hyperplane partitions is to consider a simple 3-dimensional space (see Figure 2). Assume we have a problem encoded with just 3 bits; this can be represented as a simple cube with the string 000 at the origin. The corners in this cube are numbered by bit strings and all adjacent corners are labelled by bit strings that differ by exactly 1-bit. An example is given in the top of Figure 2. The front plane of the cube contains all the points that begin with 0. If "*" is used as a "don't care" or wild card match symbol, then this plane can also be represented by the special string 0**. Strings that contain * are referred to as schemata; each schema corresponds to a hyperplane in the search space. The "order" of a hyperplane refers to the number of actual bit values that appear in its schema. Thus, 1** is order-1 while 1**1******0** would be of order-3.

The bottom of Figure 2 illustrates a 4-dimensional space represented by a cube "hanging" inside another cube. The points can be labeled as follows. Label the points in the inner cube and outer cube exactly as they are labeled in the top 3-dimensional space. Next, prefix each inner cube labeling with a 1 bit and each outer cube labeling with a 0 bit. This creates an assignment to the points in hyperspace that gives the proper adjacency in the space between strings that are 1 bit different. The inner cube now corresponds to the hyperplane 1*** while the outer cube corresponds to 0***. It is also rather easy to see that *0** corresponds to the subset of points that corresponds to the fronts of both cubes. The order-2 hyperplane 10** corresponds to the front of the inner cube.

A bit string matches a particular schemata if that bit string can be constructed from the schemata by replacing the "*" symbol with the appropriate bit value. In general, all bit strings that match a particular schemata are contained in the hyperplane partition represented by that particular schemata. Every binary encoding is a "chromosome" which corresponds to a corner in the hypercube and is a member of $2^L - 1$ different hyperplanes, where L is the length of the binary encoding. (The string of all * symbols corresponds to the space itself and is not counted as a partition of the space (Holland 1975:72)). This can be shown by taking a bit string and looking at all the possible ways that any subset of bits can be replaced by "*" symbols. In other words, there are L positions in the bit string and each position can be either the bit value contained in the string or the "*" symbol.

It is also relatively easy to see that $3^L - 1$ hyperplane partitions can be defined over the entire search space. For each of the L positions in the bit string we can have either the value *, 1 or 0 which results in 3^L combinations.

Establishing that each string is a member of $2^L - 1$ hyperplane partitions doesn't provide very much information if each point in the search space is examined in isolation. This is why the notion of a population based search is critical to genetic algorithms. A population of sample points provides information about numerous hyperplanes; furthermore, low order hyperplanes should be sampled by numerous points in the population. (This issue is reexamined in more detail in subsequent sections of this paper.) A key part of a genetic algorithm's *intrinsic* or *implicit parallelism* is derived from the fact that many hyperplanes are sampled when a population of strings is evaluated (Holland 1975); in fact, it can be argued that far more hyperplanes are sampled than the number of strings contained in the population. Many

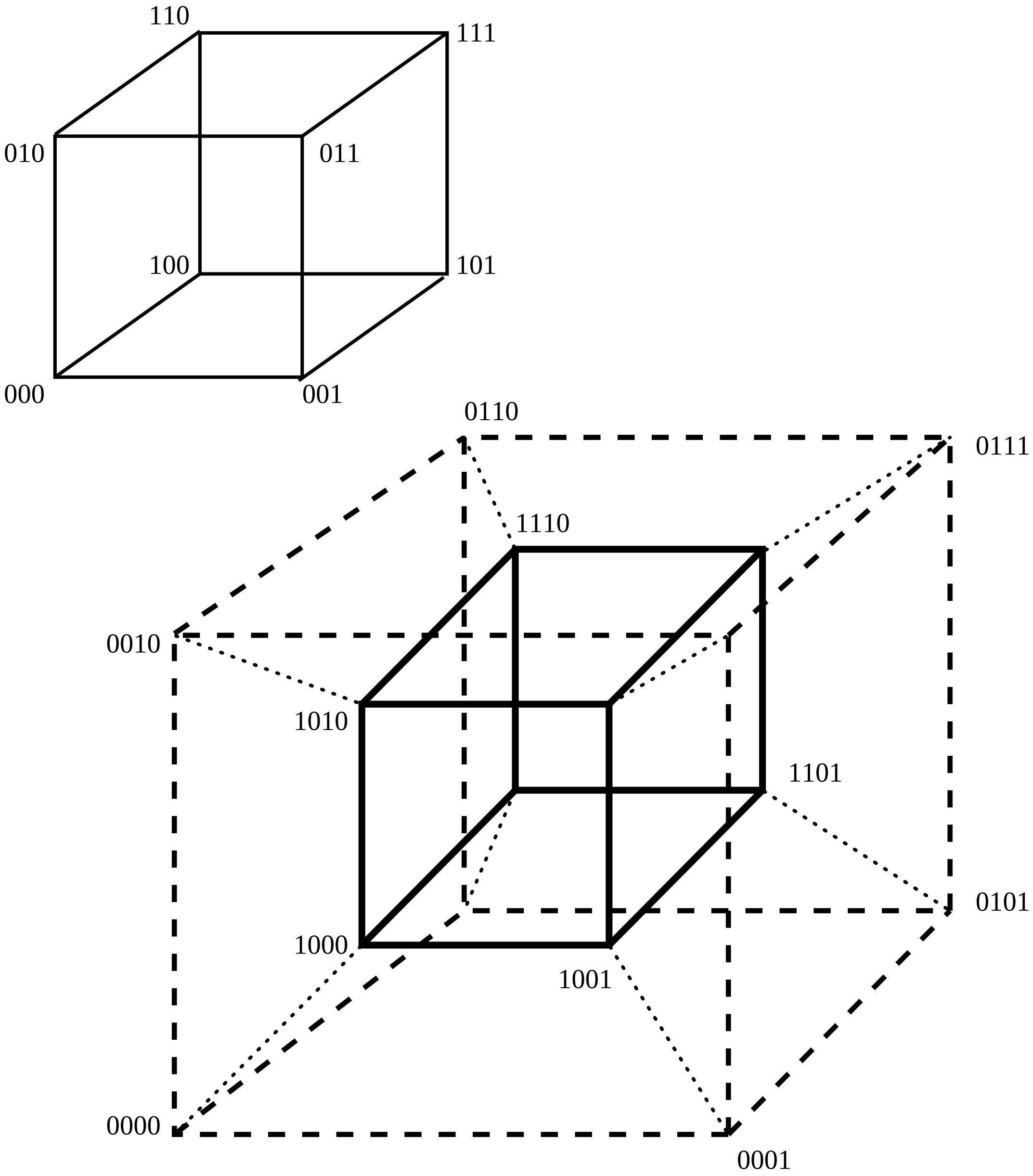

Figure 2: *A 3-dimensional cube and a 4-dimensional hypercube. The corners of the inner cube and outer cube in the bottom 4-D example are numbered in the same way as in the upper 3-D cube, except a 1 is added as a prefix to the labels of inner cube and a 0 is added as a prefix to the labels of the outer cube. Only select points are labeled in the 4-D hypercube.*

different hyperplanes are evaluated in an *implicitly parallel* fashion each time a single string is evaluated (Holland 1975:74); but it is the cumulative effects of evaluating a population of points that provides statistical information about any particular subset of hyperplanes.[1]

Implicit parallelism implies that many hyperplane competitions are simultaneously solved in parallel. The theory suggests that through the process of reproduction and recombination, the schemata of competing hyperplanes increase or decrease their representation in the population according to the relative fitness of the strings that lie in those hyperplane partitions. Because genetic algorithms operate on populations of strings, one can track the proportional representation of a single schema representing a particular hyperplane in a population and indicate whether that hyperplane will increase or decrease its representation in the population over time when fitness based selection is combined with crossover to produce offspring from existing strings in the population.

3 Two Views of Hyperplane Sampling

Another way of looking at hyperplane partitions is presented in Figure 3. A function over a single variable is plotted as a one-dimensional space, with function maximization as a goal. The hyperplane 0****...** spans the first half of the space and 1****...** spans the second half of the space. Since the strings in the 0****...** partition are on average better than those in the 1****...** partition, we would like the search to be proportionally biased toward this partition. In the second graph the portion of the space corresponding to **1**...** is shaded, which also highlights the intersection of 0****...** and **1**...**, namely, 0*1*...**. Finally, in the third graph, 0*10**...** is highlighted.

One of the points of Figure 3 is that the sampling of hyperplane partitions is not really effected by local optima. At the same time, increasing the sampling rate of partitions that are above average compared to other competing partitions does not guarantee convergence to a global optimum. The global optimum could be a relatively isolated peak, for example. Nevertheless, good solutions that are globally competitive should be found.

It is also a useful exercise to look at an example of a simple genetic algorithm in action. In Table 1, the first 3 bits of each string are given explicitly while the remainder of the bit positions are unspecified. The goal is to look at only those hyperplanes defined over the first 3 bit positions in order to see what actually happens during the selection phase when strings are duplicated according to fitness. The theory behind genetic algorithms suggests that the new distribution of points in each hyperplane should change according to the average fitness of the strings in the population that are contained in the corresponding hyperplane partition. Thus, even though a genetic algorithm never explicitly evaluates any particular hyperplane partition, it should change the distribution of string copies as if it had.

[1] Holland initially used the term *intrinsic parallelism* in his 1975 monograph, then decided to switch to *implicit parallelism* to avoid confusion with terminology in parallel computing. Unfortunately, the term *implicit parallelism* in the parallel computing community refers to parallelism which is extracted from code written in functional languages that have no explicit parallel constructs. *Implicit parallelism* does not refer to the potential for running genetic algorithms on parallel hardware, although genetic algorithms are generally viewed as highly parallelizable algorithms.

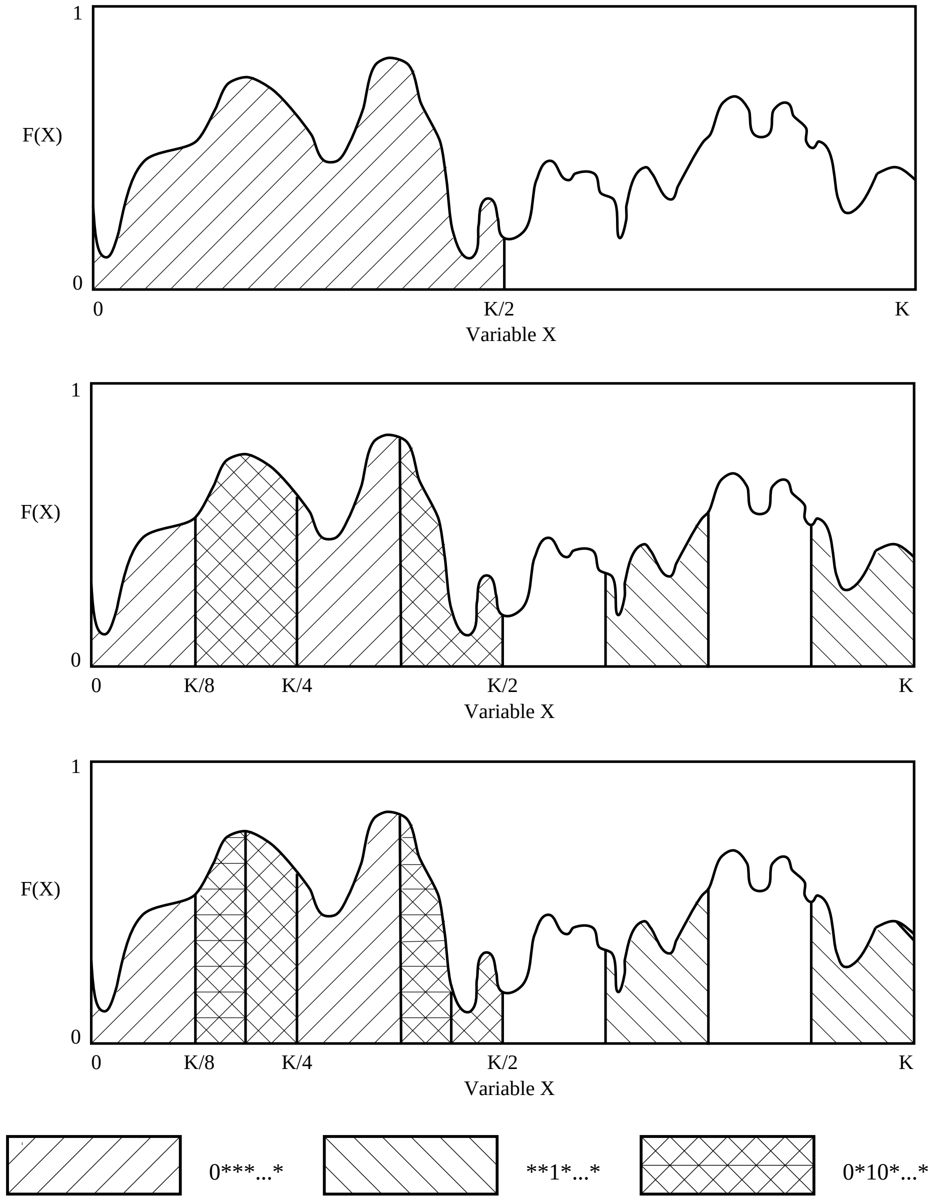

Figure 3: A function and various partitions of hyperspace. Fitness is scaled to a 0 to 1 range in this diagram.

String	Fitness	Random	Copies		String	Fitness	Random	Copies
$001b_{1,4}...b_{1,L}$	2.0	–	2		$011b_{12,4}...b_{12,L}$	0.9	0.28	1
$101b_{2,4}...b_{2,L}$	1.9	0.93	2		$000b_{13,4}...b_{13,L}$	0.8	0.13	0
$111b_{3,4}...b_{3,L}$	1.8	0.65	2		$110b_{14,4}...b_{14,L}$	0.7	0.70	1
$010b_{4,4}...b_{4,L}$	1.7	0.02	1		$110b_{15,4}...b_{15,L}$	0.6	0.80	1
$111b_{5,4}...b_{5,L}$	1.6	0.51	2		$100b_{16,4}...b_{16,L}$	0.5	0.51	1
$101b_{6,4}...b_{6,L}$	1.5	0.20	1		$011b_{17,4}...b_{17,L}$	0.4	0.76	1
$011b_{7,4}...b_{7,L}$	1.4	0.93	2		$000b_{18,4}...b_{18,L}$	0.3	0.45	0
$001b_{8,4}...b_{8,L}$	1.3	0.20	1		$001b_{19,4}...b_{19,L}$	0.2	0.61	0
$000b_{9,4}...b_{9,L}$	1.2	0.37	1		$100b_{20,4}...b_{20,L}$	0.1	0.07	0
$100b_{10,4}...b_{10,L}$	1.1	0.79	1		$010b_{21,4}...b_{21,L}$	0.0	–	0
$010b_{11,4}...b_{11,L}$	1.0	–	1					

Table 1: *A population with fitness assigned to strings according to rank.* **Random** *is a random number which determines whether or not a copy of a string is awarded for the fractional remainder of the fitness.*

The example population in Table 1 contains only 21 (partially specified) strings. Since we are not particularly concerned with the exact evaluation of these strings, the fitness values will be assigned according to rank. (The notion of assigning fitness by rank rather than by fitness proportional representation has not been discussed in detail, but the current example relates to change in representation due to fitness and not how that fitness is assigned.) The table includes information on the fitness of each string and the number of copies to be placed in the intermediate population. In this example, the number of copies produced during selection is determined by automatically assigning the integer part, then assigning the fractional part by generating a random value between 0.0 and 1.0 (a form of remainder stochastic sampling). If the random value is greater than $(1-remainder)$, then an additional copy is awarded to the corresponding individual.

Genetic algorithms appear to process many hyperplanes implicitly in parallel when selection acts on the population. Table 2 enumerates the 27 hyperplanes (3^3) that can be defined over the first three bits of the strings in the population and *explicitly* calculates the fitness associated with the corresponding hyperplane partition. The true fitness of the hyperplane partition corresponds to the average fitness of all strings that lie in that hyperplane partition. The genetic algorithm uses the population as a sample for estimating the fitness of that hyperplane partition. Of course, the only time the sample is random is during the first generation. After this, the sample of new strings should be biased toward regions that have previously contained strings that were above average with respect to previous populations.

If the genetic algorithm works as advertised, the number of copies of strings that actually fall in a particular hyperplane partition after selection should approximate the expected number of copies that should fall in that partition.

Schemata and Fitness Values										
Schema	Mean	Count	Expect	Obs		Schema	Mean	Count	Expect	Obs
101*...*	1.70	2	3.4	3		*0**...*	0.991	11	10.9	9
111*...*	1.70	2	3.4	4		00**...*	0.967	6	5.8	4
1*1*...*	1.70	4	6.8	7		0***...*	0.933	12	11.2	10
01...*	1.38	5	6.9	6		011*...*	0.900	3	2.7	4
**1*...*	1.30	10	13.0	14		010*...*	0.900	3	2.7	2
11...*	1.22	5	6.1	8		01**...*	0.900	6	5.4	6
11**...*	1.175	4	4.7	6		0*0*...*	0.833	6	5.0	3
001*...*	1.166	3	3.5	3		*10*...*	0.800	5	4.0	4
1***...*	1.089	9	9.8	11		000*...*	0.767	3	2.3	1
0*1*...*	1.033	6	6.2	7		**0*...*	0.727	11	8.0	7
10**...*	1.020	5	5.1	5		*00*...*	0.667	6	4.0	3
*1**...*	1.010	10	10.1	12		110*...*	0.650	2	1.3	2
****...*	1.000	21	21.0	21		1*0*...*	0.600	5	3.0	4
						100*...*	0.566	3	1.70	2

Table 2: *The average fitnesses (*Mean*) associated with the samples from the 27 hyperplanes defined over the first three bit positions are explicitly calculated. The Expected representation (*Expect*) and Observed representation (*Obs*) are shown.* Count *refers to the number of strings in hyperplane H before selection.*

In Table 2, the *expected* number of strings sampling a hyperplane partition after selection can be calculated by multiplying the number of hyperplane samples in the current population before selection by the average fitness of the strings in the population that fall in that partition. The *observed* number of copies actually allocated by selection is also given. In most cases the match between expected and observed sampling rate is fairly good: the error is a result of sampling error due to the small population size.

It is useful to begin formalizing the idea of tracking the potential sampling rate of a hyperplane, H. Let $M(H,t)$ be the number of strings sampling H at the current generation t in some population. Let $(t + intermediate)$ index the generation t after selection (but before crossover and mutation), and $f(H,t)$ be the average evaluation of the sample of strings in partition H in the current population. Formally, the change in representation according to fitness associated with the strings that are drawn from hyperplane H is expressed by:

$$M(H, t + intermediate) = M(H,t)\frac{f(H,t)}{\bar{f}}.$$

Of course, when strings are merely duplicated no new sampling of hyperplanes is actually occurring since no new samples are generated. Theoretically, we would like to have a sample of *new* points with this same distribution. In practice, this is generally not possible. Recombination and mutation, however, provides a means of generating new sample points while partially preserving distribution of strings across hyperplanes that is observed in the intermediate population.

3.1 Crossover Operators and Schemata

The *observed* representation of hyperplanes in Table 2 corresponds to the representation in the intermediate population after selection but before recombination. What does recombination do to the observed string distributions? Clearly, order-1 hyperplane samples are not affected by recombination, since the single critical bit is always inherited by one of the offspring. However, the observed distribution of potential samples from hyperplane partitions of order-2 and higher can be affected by crossover. Furthermore, all hyperplanes of the same order are not necessarily affected with the same probability. Consider 1-point crossover. This recombination operator is nice because it is relatively easy to quantify its effects on different schemata representing hyperplanes. To keep things simple, assume we are are working with a string encoded with just 12 bits. Now consider the following two schemata.

```
11**********          and          1**********1
```

The probability that the bits in the first schema will be separated during 1-point crossover is only $1/L-1$, since in general there are $L-1$ crossover points in a string of length L. The probability that the bits in the second rightmost schema are disrupted by 1-point crossover however is $(L-1)/(L-1)$, or 1.0, since each of the L-1 crossover points separates the bits in the schema. This leads to a general observation: when using 1-point crossover the positions of the bits in the schema are important in determining the likelihood that those bits will remain together during crossover.

3.1.1 2-point Crossover

What happens if a 2-point crossover operator is used? A 2-point crossover operator uses two randomly chosen crossover points. Strings exchange the segment that falls between these two points. Ken DeJong first observed (1975) that 2-point crossover treats strings and schemata as if they form a ring, which can be illustrated as follows:

```
   b7 b6 b5               *  *  *
 b8        b4           *         *
b9          b3        *             *
 b10       b2           *         *
   b11 b12 b1             *  1  1
```

where `b1` to `b12` represents bits 1 to 12. When viewed in this way, 1-point crossover is a special case of 2-point crossover where one of the crossover points always occurs at the wrap-around position between the first and last bit. Maximum disruptions for order-2 schemata now occur when the 2 bits are at complementary positions on this ring.

For 1-point and 2-point crossover it is clear that schemata which have bits that are close together on the string encoding (or ring) are less likely to be disrupted by crossover. More precisely, hyperplanes represented by schemata with more compact representations should be sampled at rates that are closer to those potential sampling distribution targets achieved under selection alone. For current purposes a *compact representation* with respect to schemata is one that minimizes the probability of disruption during crossover. Note that this definition is operator dependent, since both of the two order-2 schemata examined in section 3.1 are equally and maximally compact with respect to 2-point crossover, but are maximally different with respect to 1-point crossover.

3.1.2 Linkage and Defining Length

Linkage refers to the phenomenon whereby a set of bits act as "coadapted alleles" that tend to be inherited together as a group. In this case an *allele* would correspond to a particular bit value in a specific position on the chromosome. Of course, linkage can be seen as a generalization of the notion of a *compact representation with respect to schema.* Linkage is is sometimed defined by physical adjacency of bits in a string encoding; this implicitly assumes that 1-point crossover is the operator being used. Linkage under 2-point crossover is different and must be defined with respect to distance on the chromosome when treated as a ring. Nevertheless, linkage usually is equated with physical adjacency on a string, as measured by *defining length.*

The *defining length* of a schemata is based on the distance between the first and last bits in the schema with value either 0 or 1 (i.e., not a * symbol). Given that each position in a schema can be 0, 1 or *, then scanning left to right, if I_x is the index of the position of the rightmost occurrence of either a 0 or 1 and I_y is the index of the leftmost occurrence of either a 0 or 1, then the defining length is merely $I_x - I_y$. Thus, the defining length of ****1**0**10** is $12 - 5 = 7$. The defining length of a schema representing a hyperplane H is denoted here by $\Delta(H)$. The defining length is a direct measure of how many possible crossover points fall within the significant portion of a schemata. If 1-point crossover is used, then $\Delta(H)/L - 1$ is also a direct measure of how likely crossover is to fall within the significant portion of a schemata during crossover.

3.1.3 Linkage and Inversion

Along with mutation and crossover, *inversion* is often considered to be a basic genetic operator. Inversion can change the linkage of bits on the chromosome such that bits with greater nonlinear interactions can potentially be moved closer together on the chromosome.

Typically, inversion is implemented by reversing a random segment of the chromosome. However, before one can start moving bits around on the chromosome to improve linkage, the bits must have a position independent decoding. A common error that some researchers make when first implementing inversion is to reverse bit segments of a directly encoded chromosome. But just reversing some random segment of bits is nothing more than large scale mutation if the mapping from bits to parameters is position dependent.

A position independent encoding requires that each bit be tagged in some way. For example, consider the following encoding composed of pairs where the first number is a bit tag which indexes the bit and the second represents the bit value.

$$((9\ 0)\ (6\ 0)\ (2\ 1)\ (7\ 1)\ (5\ 1)\ (8\ 1)\ (3\ 0)\ (1\ 0)\ (4\ 0))$$

The linkage can now be changed by moving around the tag-bit pairs, but the string remains the same when decoded: 010010110. One must now also consider how recombination is to be implemented. Goldberg and Bridges (1990), Whitley (1991) as well as Holland (1975) discuss the problems of exploiting linkage and the recombination of tagged representations.

4 The Schema Theorem

A foundation has been laid to now develop the fundamental theorem of genetic algorithms. The *schema theorem* (Holland, 1975) provides a lower bound on the change in the sampling rate for a single hyperplane from generation t to generation $t+1$.

Consider again what happens to a particular hyperplane, H when only selection occurs.

$$M(H, t + intermediate) = M(H,t)\frac{f(H,t)}{\bar{f}}.$$

To calculate M(H,t+1) we must consider the effects of crossover as the next generation is created from the intermediate generation. First we consider that crossover is applied probabilistically to a portion of the population. For that part of the population that does not undergo crossover, the representation due to selection is unchanged. When crossover does occur, then we must calculate losses due to its disruptive effects.

$$M(H,t+1) = (1-p_c)M(H,t)\frac{f(H,t)}{\bar{f}} + p_c\left[M(H,t)\frac{f(H,t)}{\bar{f}}(1 - losses) + gains\right]$$

In the derivation of the schema theorem a conservative assumption is made at this point. It is assumed that crossover within the defining length of the schema is always disruptive to the schema representing H. In fact, this is not true and an exact calculation of the effects of crossover is presented later in this paper. For example, assume we are interested in the schema 11*****. If a string such as 1110101 were recombined between the first two bits with a string such as 1000000 or 0100000, no disruption would occur in hyperplane 11***** since one of the offspring would still reside in this partition. Also, if 1000000 and 0100000 were recombined exactly between the first and second bit, a new independent offspring would sample 11*****; this is the sources of *gains* that is referred to in the above calculation. To simplify things, *gains* are ignored and the conservative assumption is made that crossover falling in the significant portion of a schema always leads to disruption. Thus,

$$M(H,t+1) \geq (1-p_c)M(H,t)\frac{f(H,t)}{\bar{f}} + p_c\left[M(H,t)\frac{f(H,t)}{\bar{f}}(1 - disruptions)\right]$$

where *disruptions* overestimates losses. We might wish to consider one exception: if two strings that both sample H are recombined, then no disruption occurs. Let $P(H,t)$ denote the proportional represention of H obtained by dividing $M(H,t)$ by the population size. The probability that a randomly chosen mate samples H is just $P(H,t)$. Recall that $\Delta(H)$ is the defining length associated with 1-point crossover. Disruption is therefore given by:

$$\frac{\Delta(H)}{L-1}(1 - P(H,t)).$$

At this point, the inequality can be simplified. Both sides can be divided by the population size to convert this into an expression for $P(H, t+1)$, the proportional representation of H at generation $t+1$. Furthermore, the expression can be rearranged with respect to p_c.

$$P(H, t+1) \geq P(H, t) \, \frac{f(H, t)}{\bar{f}} \left[1 - p_c \frac{\Delta(H)}{L-1} (1 - P(H, t)) \right]$$

We now have a useful version of the schema theorem (although it does not yet consider mutation); but it is not the only version in the literature. For example, both parents are typically chosen based on fitness. This can be added to the schema theorem by merely indicating the alternative parent is chosen from the intermediate population after selection.

$$P(H, t+1) \geq P(H, t) \, \frac{f(H, t)}{\bar{f}} \left[1 - p_c \frac{\Delta(H)}{L-1} (1 - P(H, t) \, \frac{f(H, t)}{\bar{f}}) \right]$$

Finally, mutation is included. Let $o(H)$ be a function that returns the order of the hyperplane H. The order of H exactly corresponds to a count of the number of bits in the schema representing H that have value 0 or 1. Let the mutation probability be p_m where mutation always flips the bit. Thus the probability that mutation does affect the schema representing H is $(1 - p_m)^{o(H)}$. This leads to the following expression of the schema theorem.

$$P(H, t+1) \geq P(H, t) \, \frac{f(H, t)}{\bar{f}} \left[1 - p_c \frac{\Delta(H)}{L-1} (1 - P(H, t) \, \frac{f(H, t)}{\bar{f}}) \right] (1 - p_m)^{o(H)}$$

4.1 Crossover, Mutation and Premature Convergence

Clearly the schema theorem places the greatest emphasis on the role of crossover and hyperplane sampling in genetic search. To maximize the preservation of hyperplane samples after selection, the disruptive effects of crossover and mutation should be minimized. This suggests that mutation should perhaps not be used at all, or at least used at very low levels.

The motivation for using mutation, then, is to prevent the permanent loss of any particular bit or allele. After several generations it is possible that selection will drive all the bits in some position to a single value: either 0 or 1. If this happens without the genetic algorithm converging to a satisfactory solution, then the algorithm has *prematurely converged.* This may particularly be a problem if one is working with a small population. Without a mutation operator, there is no possibility for reintroducing the missing bit value. Also, if the target function is nonstationary and the fitness landscape changes over time (which is certainly the case in real biological systems), then there needs to be some source of continuing genetic diversity. Mutation, therefore acts as a background operator, occasionally changing bit values and allowing alternative alleles (and hyperplane partitions) to be retested.

This particular interpretation of mutation ignores its potential as a hill-climbing mechanism: from the strict hyperplane sampling point of view imposed by the schema theorem mutation is a necessary evil. But this is perhaps a limited point of view. There are several experimental researchers that point out that genetic search using mutation and no crossover often produces a fairly robust search. And there is little or no theory that has addressed the interactions of hyperplane sampling and hill-climbing in genetic search.

Another problem related to premature convergence is the need for *scaling* the population fitness. As the average evaluation of the strings in the population increases, the variance in fitness decreases in the population. There may be little difference between the best and worst individual in the population after several generations, and the selective pressure based on fitness is correspondingly reduced. This problem can partially be addressed by using some form of fitness scaling (Grefenstette, 1986; Goldberg, 1989). In the simplest case, one can subtract the evaluation of the worst string in the population from the evaluations of all strings in the population. One can now compute the average string evaluation as well as fitness values using this adjusted evaluation, which will increase the resulting selective pressure. Alternatively, one can use a rank based form of selection.

4.2 How Recombination Moves Through a Hypercube

The nice thing about 1-point crossover is that it is easy to model analytically. But it is also easy to show analytically that if one is interested in minimizing schema disruption, then 2-point crossover is better. But operators that use many crossover points should be avoided because of extreme disruption to schemata. This is again a point of view imposed by a strict interpretation of the schema theorem. On the other hand, disruption may not be the only factor affecting the performance of a genetic algorithm.

4.2.1 Uniform Crossover

The operator that has received the most attention in recent years is *uniform crossover*. Uniform crossover was studied in some detail by Ackley (1987) and popularized by Syswerda (1989). Uniform crossover works as follows: for each bit position 1 to L, randomly pick each bit from either of the two parent strings. This means that each bit is inherited independently from any other bit and that there is, in fact, no linkage between bits. It also means that uniform crossover is unbiased with respect to defining length. In general the probability of disruption is $1 - (1/2)^{o(H)-1}$, where o(H) is the order of the schema we are interested in. (It doesn't matter which offspring inherits the first critical bit, but all other bits must be inherited by that same offspring. This is also a worst case probability of disruption which assumes no alleles found in the schema of interest are shared by the parents.) Thus, for any order-3 schemata the probability of uniform crossover separating the critical bits is always $1 - (1/2)^2 = 0.75$. Consider for a moment a string of 9 bits. The defining length of a schema must equal 6 before the disruptive probabilities of 1-point crossover match those associated with uniform crossover ($6/8 = .75$). We can define 84 different order-3 schemata over any particular string of 9 bits (i.e., 9 choose 3). Of these schemata, only 19 of the 84 order-2 schemata have a disruption rate higher than 0.75 under 1-point crossover. Another 15 have exactly the same disruption rate, and 50 of the 84 order-2 schemata have a lower disruption rate. It is relative easy to show that, while uniform crossover is unbiased with respect to defining length, it is also generally more disruptive than 1-point crossover. Spears and DeJong (1991) have shown that uniform crossover is in every case more disruptive than 2-point crossover for order-3 schemata for all defining lengths.

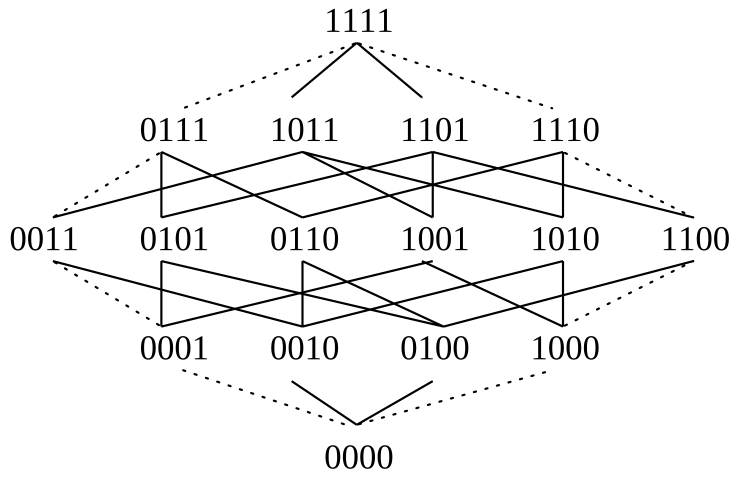

Figure 4: *This graph illustrates paths though 4-D space. A 1-point crossover of 1111 and 0000 can only generate offspring that reside along the dashed paths at the edges of this graph.*

Despite these analytical results, several researchers have suggested that uniform crossover is sometimes a better recombination operator. One can point to its lack of representational bias with respect to schema disruption as a possible explanation, but this is unlikely since uniform crossover is uniformly worse than 2-point crossover. Spears and DeJong (1991:314) speculate that, "With small populations, more disruptive crossover operators such as uniform or n-point ($n \gg 2$) may yield better results because they help overcome the limited information capacity of smaller populations and the tendency for more homogeneity." Eshelman (1991) has made similar arguments outlining the advantages of disruptive operators.

There is another sense in which uniform crossover is unbiased. Assume we wish to recombine the bits string 0000 and 1111. We can conveniently lay out the 4-dimensional hypercube as shown in Figure 4. We can also view these strings as being connected by a set of minimal paths through the hypercube; pick one parent string as the origin and the other as the destination. Now change a single bit in the binary representation corresponding to the point of origin. Any such move will reach a point that is one move closer to the destination. In Figure 4 it is easy to see that changing a single bit is a move up or down in the graph.

All of the points between 0000 and 1111 are reachable by some single application of uniform crossover. However, 1-point crossover only generates strings that lie along two complementary paths (in the figure, the leftmost and rightmost paths) through this 4-dimensional hypercube. In general, uniform crossover will draw a complementary pair of sample points with equal probability from all points that lie along *any* complementary minimal paths in the hypercube between the two parents, while 1-point crossover samples points from only two specific complementary minimal paths between the two parent strings. It is also easy to see that 2-point crossover is less restrictive than 1-point crossover. Note that the number of bits that are different between two strings is just the Hamming distance, $\mathcal{H}$. Not including the original parent strings, uniform crossover can generate $2^{\mathcal{H}} - 2$ different strings, while 1-point crossover can generate $2(\mathcal{H} - 1)$ different strings since there are $\mathcal{H}$ crossover points that produce unique offspring (see the discussion in the next section) and each crossover produces 2 offspring. The 2-point crossover operator can generate $2\binom{\mathcal{H}}{2} = \mathcal{H}^2 - \mathcal{H}$ different offspring since there are $\mathcal{H}$ choose 2 different crossover points that will result in offspring that are not copies of the parents and each pair of crossover points generates 2 strings.

4.3 Reduced Surrogates

Consider crossing the following two strings and a "reduced" version of the same strings, where the bits the strings share in common have been removed.

```
0001111011010011          ----11---1-----1
0001001010010010          ----00---0-----0
```

Both strings lie in the hyperplane 0001**101*01001*. The flip side of this observation is that crossover is really restricted to a subcube defined over the bit positions that are different. We can isolate this subcube by removing all of the bits that are equivalent in the two parent structures. Booker (1987) refers to strings such as `----11---1-----1` and `----00---0-----0` as the "reduced surrogates" of the original parent chromosomes.

When viewed in this way, it is clear that recombination of these particular strings occurs in a 4-dimensional subcube, more or less identical to the one examined in the previous example. Uniform crossover is unbiased with respect to this subcube in the sense that uniform crossover will still sample in an unbiased, uniform fashion from all of the pairs of points that lie along complementary minimal paths in the subcube defined between the two original parent strings. On the other hand, simple 1-point or 2-point crossover will not. To help illustrate this idea, we recombine the *original* strings, but examine the offspring in their "reduced" forms. For example, simple 1-point crossover will generate offspring `----11---1-----0` and `----00---0-----1` with a probability of 6/15 since there are 6 crossover points in the original parent strings between the third and fourth bits in the reduced subcube and L-1 = 15. On the other hand, `----10---0-----0` and `----01---1-----1` are sampled with a probability of only 1/15 since there is only a single crossover point in the original parent structures that falls between the first and second bits that define the subcube.

One can remove this particular bias, however. We apply crossover on the reduced surrogates. Crossover can now exploit the fact that there is really only 1 crossover point between any significant bits that appear in the reduced surrogate forms. There is also another benefit. If at least 1 crossover point falls between the first and last significant bits in the reduced surrogates, the offspring are guaranteed not to be duplicates of the parents. (This assumes the parents differ by at least two bits). Thus, new sample points in hyperspace are generated.

The debate on the merits of uniform crossover and operators such as 2-point reduced surrogate crossover is not a closed issue. To fully understand the interaction between hyperplane sampling, population size, premature convergence, crossover operators, genetic diversity and the role of hill-climbing by mutation requires better analytical methods.

5 The Case for Binary Alphabets

The motivation behind the use of a minimal binary alphabet is based on relatively simple counting arguments. A minimal alphabet maximizes the number of hyperplane partitions directly available in the encoding for schema processing. These low order hyperplane partitions are also sampled at a higher rate than would occur with an alphabet of higher cardinality.

Any set of order-1 schemata such as 1*** and 0*** cuts the search space in half. Clearly, there are L pairs of order-1 schemata. For order-2 schemata, there are $\binom{L}{2}$ ways to pick locations in which to place the 2 critical bits positions, and there are 2^2 possible ways to assign values to those bits. In general, if we wish to count how many schemata representing hyperplanes exist at some order i, this value is given by $2^i\binom{L}{i}$ where $\binom{L}{i}$ counts the number of ways to pick i positions that will have significant bit values in a string of length L and 2^i is the number of ways to assign values to those positions. This ideal can be illustrated for order-1 and order-2 schemata as follows:

```
   Order 1 Schemata                          Order 2 Schemata
0***  *0**  **0*  ***0         00**  0*0*  0**0  *00*  *0*0  **00
1***  *1**  **1*  ***1         01**  0*1*  0**1  *01*  *0*1  **01
                               10**  1*0*  1**0  *10*  *1*0  **10
                               11**  1*1*  1**1  *11*  *1*1  **11
```

These counting arguments naturally lead to questions about the relationship between population size and the number of hyperplanes that are sampled by a genetic algorithm. One can take a very simple view of this question and ask how many schemata of order-1 are sampled and how well are they represented in a population of size N. These numbers are based on the assumption that we are interested in hyperplane representations associated with the initial random population, since selection changes the distributions over time. In a population of size N there should be N/2 samples of each of the 2L order-1 hyperplane partitions. Therefore 50% of the population falls in any particular order-1 partition. Each order-2 partition is sampled by 25% of the population. In general then, each hyperplane of order i is sampled by $(1/2)^i$ of the population.

5.1 The N^3 Argument

These counting arguments set the stage for the claim that a genetic algorithm processes on the order of N^3 hyperplanes when the population size is N. The derivation used here is based on work found in the appendix of Fitzpatrick and Grefenstette (1988).

Let θ be the highest order of hyperplane which is represented in a population of size N by at least ϕ copies; θ is given by $log(N/\phi)$. We wish to have at least ϕ samples of a hyperplane before claiming that we are statistically sampling that hyperplane.

Recall that the number of different hyperplane partitions of order-θ is given by $2^\theta \binom{L}{\theta}$ which is just the number of different ways to pick θ different positions and to assign all possible binary values to each subset of the θ positions. Thus, we now need to show

$$2^\theta \left(\begin{array}{c} L \\ \theta \end{array} \right) \geq N^3 \qquad \text{which implies} \qquad 2^\theta \left(\begin{array}{c} L \\ \theta \end{array} \right) \geq (2^\theta \phi)^3$$

since $\theta = log(N/\phi)$ and $N = 2^{\theta}\phi$. Fitzpatrick and Grefenstette now make the following arguments. Assume $L \geq 64$ and $2^6 \leq N \leq 2^{20}$. Pick $\phi = 8$, which implies $3 \leq \theta \leq 17$. By inspection the number of schemata processed is greater than N^3.

This argument does not hold in general for any population of size N. Given a string of length L, the number of hyperplanes in the space is finite. However, the population size can be chosen arbitrarily. The total number of schemata associated with a string of length L is 3^L. Thus if we pick a population size where $N = 3^L$ then at most N hyperplanes can be processed (Michael Vose, personal communication). Therefore, N must be chosen with respect to L to make the N^3 argument reasonable. At the same time, the range of values $2^6 \leq N \leq 2^{20}$ does represent a wide range of *practical* population sizes.

Still, the argument that N^3 hyperplanes are *usefully* processed assumes that all of these hyperplanes are processed with some degree of independence. Notice that the current derivation counts only those schemata that are exactly of order-θ. The sum of all schemata from order-1 to order-θ that should be well represented in a random initial population is given by: $\sum_{x=1}^{\theta} 2^x \binom{L}{x}$. By only counting schemata that are exactly of order-θ we might hope to avoid arguments about interactions with lower order schemata. However, all the N^3 argument really shows is that there may be as many as N^3 hyperplanes that are well represented given an appropriate population size. But a simple static count of the number of schemata available for processing fails to consider the dynamic behavior of the genetic algorithm.

As discussed later in this tutorial, dynamic models of the genetic algorithm now exist (Vose and Liepins, 1991; Whitley, Das and Crabb 1992). There has not yet, however, been any real attempt to use these models to look at complex interactions between large numbers of hyperplane competitions. It is obvious in some vacuous sense that knowing the distribution of the initial population as well as the fitnesses of these strings (and the strings that are subsequently generated by the genetic algorithm) is sufficient information for modeling the dynamic behavior of the genetic algorithm (Vose 1993). This suggests that we only need information about those strings sampled by the genetic algorithm. However, this micro-level view of the genetic algorithm does not seems to explain its macro-level processing power.

5.2 The Case for Nonbinary Alphabets

There are two basic arguments against using higher cardinality alphabets. First, there will be fewer explicit hyperplane partitions. Second, the alphabetic characters (and the corresponding hyperplane partitions) associated with a higher cardinality alphabet will not be as well represented in a finite population. This either forces the use of larger population sizes or the effectiveness of statistical sampling is diminished.

The arguments for using binary alphabets assume that the schemata representing hyperplanes must be explicitly and directly manipulated by recombination. Antonisse (1989) has argued that this need not be the case and that higher order alphabets offer as much richness in terms of hyperplane samples as lower order alphabets. For example, using an alphabet of the four characters A, B, C, D one can define all the same hyperplane partitions in a binary alphabet by defining partitions such as (A and B), (C and D), etc. In general, Antonisse argues that one can look at the all subsets of the power set of schemata as also defining hyperplanes. Viewed in this way, higher cardinality alphabets yield *more* hyperplane partitions than binary alphabets. Antonisse's arguments fail to show however, that the hyperplanes that corresponds to the subsets defined in this scheme actually provide new *independent* sources of information which can be processed in a meaningful way by a genetic algorithm. This does not disprove Antonisse's claims, but does suggest that there are unresolved issues associated with this hypothesis.

There are other arguments for nonbinary encodings. Davis (1991) argues that the disadvantages of nonbinary encodings can be offset by the larger range of operators that can be applied to problems, and that more problem dependent aspects of the coding can be exploited. Schaffer and Eshelman (1992) as well as Wright (1991) present interesting arguments for real-valued encodings. Goldberg (1991) suggests that virtual minimal alphabets that facilitate hyperplane sampling can emerge from higher cardinality alphabets.

6 Criticisms of the Schema Theorem

There are some obvious limitations of the schema theorem which restrict its usefulness. First, it is an inequality. By ignoring string gains and undercounting string losses, a great deal of information is lost. The inexactness of the inequality is such that if one were to try to use the schema theorem to predict the representation of a particular hyperplane over multiple generations, the resulting predictions would in many cases be useless or misleading (e.g. Grefenstette 1993; Vose, personal communication, 1993). Second, the *observed* fitness of a hyperplane H at time t can change dramatically as the population concentrates its new samples in more specialized subpartitions of hyperspace. Thus, looking at the average fitness of all the strings in a particular hyperplane (or using a random sample to estimate this fitness) is only relevant to the first generation or two (Grefenstette and Baker, 1989). After this, the sampling of strings is biased and the inexactness of the schema theorem makes it impossible to predict computational behavior.

In general, the schema theorem provides a lower bound that holds for only one generation into the future. Therefore, one cannot predict the representation of a hyperplane H over multiple generations without considering what is simultaneous happening to the other hyperplanes being processed by the genetic algorithm.

These criticisms imply that the views of hyperplane sampling presented in section 3 of this tutorial may be good rhetorical tools for explaining hyperplane sampling, but they fail to capture the full complexity of the genetic algorithm. This is partly because the discussion in section 3 focuses on the impact of selection without considering the disruptive *and generative* effects of crossover. The schema theorem does not provide an exact picture of the genetic algorithms behavior and cannot predict how a specific hyperplane is processed over time. In the next section, a introduction is to an exact version of the schema theorem.

7 An Executable Model of the Genetic Algorithm

Consider the complete version of the schema theorem before dropping the gains term and simplifying the losses calculation.

$$P(Z,t+1) = P(Z,t)\frac{f(Z,t)}{\bar{f}}(1 - \{p_c \text{ losses}\}) + \{p_c \text{ gains.}\}$$

In the current formulation, Z will refer to a string. Assume we apply this equation to each string in the search space. The result is an exact model of the computational behavior of a genetic algorithm. Since modeling strings models the highest order schemata, the model implicitly includes all lower order schemata. Also, the fitnesses of strings are constants in the canonical genetic algorithm using fitness proportional reproduction and one need not worry about changes in the *observed* fitness of a hyperplane as represented by the current population. Given a specification of Z, one can exactly calculate losses and gains. Losses occur when a string crosses with another string and the resulting offspring fails to preserve the original string. Gains occur when two different strings cross and independently create a new copy of some string. For example, if Z = 000 then recombining 100 and 001 will always produce a new copy of 000. Assuming 1-point crossover is used as an operator, the probability of "losses" and "gains" for the string Z = 000 are calculated as follows:

$$\text{losses} = P_{I0}\frac{f(111)}{\bar{f}}P(111,t) + P_{I0}\frac{f(101)}{\bar{f}}P(101,t)$$

$$+P_{I1}\frac{f(110)}{\bar{f}}P(110,t) + P_{I2}\frac{f(011)}{\bar{f}}P(011,t).$$

$$\text{gains} = P_{I0}\frac{f(001)}{\bar{f}}P(001,t)\frac{f(100)}{\bar{f}}P(100,t) + P_{I1}\frac{f(010)}{\bar{f}}P(010,t)\frac{f(100)}{\bar{f}}P(100,t)$$

$$+P_{I1}\frac{f(011)}{\bar{f}}P(011,t)\frac{f(100)}{\bar{f}}P(100,t) + P_{I2}\frac{f(001)}{\bar{f}}P(001,t)\frac{f(110)}{\bar{f}}P(110,t)$$

$$+P_{I2}\frac{f(001)}{\bar{f}}P(001,t)\frac{f(010)}{\bar{f}}P(010,t).$$

The use of P_{I0} in the preceding equations represents the probability of crossover in any position on the corresponding string or string pair. Since Z is a string, it follows that $P_{I0} =$ 1.0 and crossover in the relevant cases will always produce either a loss or a gain (depending on the expression in which the term appears). The probability that one-point crossover will fall between the first and second bit will be denoted by P_{I1}. In this case, crossover must fall in exactly this position with respect to the corresponding strings to result in a loss or a gain. Likewise, P_{I2} will denote the probability that one-point crossover will fall between the second and third bit and the use of P_{I2} in the computation implies that crossover must fall in this position for a particular string or string pair to effect the calculation of losses or gains. In the above illustration, $P_{I1} = P_{I2} = 0.5$.

The equations can be generalized to cover the remaining 7 strings in the space. This translation is accomplished using bitwise addition modulo 2 (i.e., a bitwise exclusive-or denoted by $\oplus$. See Figure 4 and section 6.4). The function $(S_i \oplus Z)$ is applied to each bit string, S_i, contained in the equation presented in this section to produce the appropriate corresponding strings for generating an expression for computing all terms of the form P(Z,t+1).

7.1 A Generalized Form Based on Equation Generators

The 3 bit equations are similar to the 2 bit equations developed by Goldberg (1987). The development of a general form for these equations is illustrated by generating the loss and gain terms in a systematic fashion (Whitley, Das and Crabb, 1992). Because the number of terms in the equations is greater than the number of strings in the search space, it is only practical to develop equations for encodings of approximately 15 bits. The equations need only be defined once for one string in the space; the *standard form* of the equation is always defined for the string composed of all zero bits. Let S represent the set of binary strings of length L, indexed by i. In general, the string composed of all zero bits is denoted S_0.

7.2 Generating String Losses for 1-point crossover

Consider two strings 00000000000 and 00010000100. Using 1-point crossover, if the crossover occurs before the first "1" bit or after the last "1" bit, no disruption will occur. Any crossover between the 1 bits, however, will produce disruption: neither parent will survive crossover. Also note that recombining 00000000000 with any string of the form 0001#####100 will produce the same pattern of disruption. We will refer to this string as a generator: it is like a schema, but # is used instead of * to better distinguish between a generator and the corresponding hyperplane. Bridges and Goldberg (1987) formalize the notion of a generator as follows. Consider strings B and B′ where the first x bits are equal, the middle $(\delta + 1)$ bits have the pattern $b\#\#...\#b$ for B and $\bar{b}\#\#...\#\bar{b}$ for B′. Given that the strings are of length L, the last $(L - \delta - x - 1)$ bits are equivalent. The $\bar{b}$ bits are referred to as "sentry bits" and they are used to define the probability of disruption. In standard form, $B = S_0$ and the sentry bits must be 1. The following directed acyclic graph illustrates all generators for "string losses" for the standard form of a 5 bit equation for S_0.

```
                    1###1
                   /     \
                  /       \
            01##1           1##10
           /     \         /     \
          /       \       /       \
      001#1         01#10           1#100
     /     \       /     \         /     \
    /       \     /       \       /       \
00011         00110         01100           11000
```

The graph structure allows one to visualize the set of all generators for string losses. In general, the root of this graph is defined by a string with a sentry bit in the first and last bit positions, and the generator token "#" in all other intermediate positions. A move down and to the left in the graph causes the leftmost sentry bit to be shifted right; a move down and to the right causes the rightmost sentry bit to be shifted left. All bits outside the sentry positions are "0" bits. Summing over the graph, one can see that there are $\sum_{j=1}^{L-1} \; j \cdot 2^{L-j-1}$ or $(2^L - L - 1)$ strings generated as potential sources of string losses.

For each string S_i produced by one of the "middle" generators in the above graph structure, a term of the following form is added to the *losses* equations:

$$\frac{\delta(S_i)}{L-1}\frac{f(S_i)}{\bar{f}}\ P(S_i,t)$$

where $\delta(S_i)$ is a function that counts the number of crossover points between sentry bits in string S_i.

7.3 Generating String Gains for 1-point crossover

Bridges and Goldberg (1987) note that string gains for a string B are produced from two strings Q and R which have the following relationship to B.

Region ->	beginning	middle	end
Length ->	a	r	w
Q Characteristics	$\#\#...\#\bar{b}$	=	=
R Characteristics	=	=	$\bar{b}\#...\#$

The "=" symbol denotes regions where the bits in Q and R match those in B; again B = S_0 for the *standard form* of the equations. Sentry bits are located such that 1-point crossover between sentry bits produces a new copy of B, while crossover of Q and R outside the sentry bits will not produce a new copy of B.

Bridges and Goldberg define a beginning function A[B,α] and ending function $\Omega\,[B,\omega]$, assuming $L-\omega > \alpha - 1$, where for the standard form of the equations:

$$A\,[S_0,\alpha] = \#\#...\#\#1_{\alpha-1}0_{\alpha}...0_{L-1} \qquad \text{and} \qquad \Omega\,[S_0,\omega] = 0_0...0_{L-\omega-1}1_{L-\omega}\#\#...\#\#.$$

These generators can again be presented as a directed acyclic graph structure composed of paired templates which will be referred to as the upper A-generator and lower Ω-generator. The following are the generators in a 5 bit problem.

```
                    10000
                    00001
                   /     \
                  /       \
              #1000         10000
              00001         0001#
             /     \       /     \
            /       \     /       \
        ##100         #1000         10000
        00001         0001#         001##
       /     \       /     \       /     \
      /       \     /       \     /       \
  ###10         ##100         #1000         10000
  00001         0001#         001##         01###
```

In this case, the root of the directed acyclic graph is defined by starting with the most specific generator pair. The A-generator of the root has a "1" bit as the sentry bit in the first position, and all other bits are "0." The Ω-generator of the root has a "1" bit as the sentry bit in the last position, and all other bits are "0." A move down and left in the graph is produced by shifting the left sentry bit of the current upper A-generator to the right. A move down and right is produced by shifting the right sentry bit of the current lower Ω-generator to the left. Each vacant bit position outside of the sentry bits which results from a shift operation is filled using the # symbol.

For any level k of the directed graph there are k generators and the number of string pairs generated at that level is 2^{k-1} for each pair of generators (the root is level 1). Therefore, the total number of string pairs that must be included in the equations to calculate string gains for S_0 of length L is $\sum_{k=1}^{L-1} k \cdot 2^{k-1}$.

Let $S_{\alpha+x}$ and $S_{\omega+y}$ be two strings produced by a generator pair, such that $S_{\alpha+x}$ was produced by the A-generator and has a sentry bit at location $\alpha-1$ and $S_{\omega+y}$ was produced by the Ω-generator with a sentry bit at $L-\omega$. (The x and y terms are correction factors added to α and ω in order to uniquely index a string in S.) Let the critical crossover region associated with $S_{\alpha+x}$ and $S_{\omega+y}$ be computed by the function $\rho(S_{\alpha+x}, S_{\omega+y}) = L - \omega - (\alpha - 1)$. For each string pair $S_{\alpha+x}$ and $S_{\omega+y}$ a term of the following form is added to the *gains* equations:

$$\frac{\rho(S_{\alpha+x}, S_{\omega+y})}{L-1} \; \frac{f(S_{\alpha+x})}{\bar{f}} \; P(S_{\alpha+x}, t) \; \frac{f(S_{\omega+y})}{\bar{f}} \; P(S_{\omega+y}, t)$$

where $\rho(S_{\alpha+x}, S_{\omega+y})$ counts the number of crossover points that fall in the critical region defined by the sentry bits located at $\alpha - 1$ and $L - \omega$.

The generators are used as part of a two stage computation where the generators are first used to create an exact equation in standard form. A simple transformation function maps the equations to all other strings in the space.

7.4 The Vose and Liepins Models

The executable equations developed by Whitley (1993a) represent a special case of the model of a simple genetic algorithm introduced by Vose and Liepins (1991). In the Vose and Liepins model, the vector $s^t \in \Re$ represents the t th generation of the genetic algorithm and the i th component of s^t is the probability that the string i is selected for the gene pool. Using i to refer to a string in s can sometimes be confusing. The symbol S has already been used to denote the set of binary strings, also indexed by i. This notation will be used where appropriate to avoid confusion. Note that s^t corresponds to the expected distribution of strings in the *intermediate population* in the generational reproduction process (after selection has occurred, but before recombination).

In the Vose and Liepins formulation,

$$s_i^t \sim P(S_i, t) f(S_i)$$

where $\sim$ is the equivalence relation such that $x \sim y$ if and only if $\exists \gamma > 0 \mid x = \gamma y$. In this formulation, the term $1/\bar{f}$, which would represent the average population fitness normally associated with fitness proportional reproduction, can be absorbed by the γ term.

Let $V = 2^L$, the number of strings in the search space. The vector $p^t \in \Re^V$ is defined such that the k th component of the vector is equal to the proportional representation of string k at generation t **before** selection occurs. The k component of p^t would be the same as $P(S_k, t)$ in the notation more commonly associated with the schema theorem. Finally let $r_{i,j}(k)$ be the probability that string k results from the recombination of strings i and j. Now, using $\mathcal{E}$ to denote expectation,

$$\mathcal{E}\ p_k^{t+1} = \sum_{i,j}\ s_i^t\ s_j^t\ r_{i,j}(k).$$

To further generalize this model, the function $r_{i,j}(k)$ is used to construct a mixing matrix M where the i, jth entry $m_{i,j} = r_{i,j}(0)$. Note that this matrix gives the probabilities that crossing strings i and j will produce the string S_0. Technically, the definition of $r_{i,j}(k)$ assumes that exactly one offspring is produced. But note that M has two entries for each string pair i, j where $i \neq j$, which is equivalent to producing two offspring. For current purposes, assume *no mutation* is used and 1-point crossover is used as the recombination operator. The matrix M is symmetric and is zero everywhere on the diagonal except for entry $m_{0,0}$ which is 1.0. Note that M is expressed entirely in terms of string gain information. Therefore, the first row and column of the matrix has entries inversely related to the string *losses* probabilities, each entry is given by $1 - (0.5\ \delta(S_i)/L - 1)$, where each string in the set S is crossed with S_0. For completeness, $\delta(S_i)$ for strings not produced by the string loss generators is 0 and, thus, the probability of obtaining S_0 during reproduction is 1.0. The remainder of the matrix entries are given by $0.5\ \frac{\rho(S_{\alpha+x}, S_{\omega+y})}{L-1}$. For each pair of strings produced by the *string gains generators* determine their index and enter the value returned by the function into the corresponding location in M. For completeness, $\rho(S_j, S_k) = 0$ for all pairs of strings not generated by the string gains generators (i.e., $m_{j,k} = 0$).

Once defined M does not change since it is not affected by variations in fitness or proportional representation in the population. Thus, given the assumption of no mutations, that s is updated each generation to correct for changes in the population average, and that 1-point crossover is used, then the standard form of the executable equations corresponds to the following portion of the Liepins and Vose model (T denotes transpose):

$$s^T M s.$$

An alternative form of M denoted M' can be defined by having only a single entry for each string pair i, j where $i \neq j$. This is done by doubling the value of the enties in the lower triangle and setting the entries in the upper triangle of the matrix to 0.0. Assuming each component of s is given by $s_i = P(S_i, t)(f(S_i)/\bar{f}))$, this has the rhetorical advantage that

$$s^T M'(:, 1) s_0 = P(S_0, t)(f(S_0)/\bar{f})(1 - losses).$$

where $M'(:, 1)$ is the first column of M' and s_0 is the first component of s. Not including the above subcomputation, the remainder of the computation of $s^T M' s$ calculates string gains.

Vose and Liepins formalize the notion that bitwise exclusive-or can be used to remap all the strings in the search space, in this case represented by the vector s. They show that if

A Transform Function to Redefine Equations	
$000 \oplus 010 \Rightarrow 010$	$100 \oplus 010 \Rightarrow 110$
$001 \oplus 010 \Rightarrow 011$	$101 \oplus 010 \Rightarrow 111$
$010 \oplus 010 \Rightarrow 000$	$110 \oplus 010 \Rightarrow 100$
$011 \oplus 010 \Rightarrow 001$	$111 \oplus 010 \Rightarrow 101$

Figure 5: *The operator $\oplus$ is bit-wise exclusive-or. Let $r_{i,j}(k)$ be the probability that k results from the recombination of strings i and j. If recombination is a combination of crossover and mutation then $r_{i,j}(k \oplus 0) = r_{i \oplus k, j \oplus k}(0)$. The strings are reordered with respect to 010.*

recombination is a combination of crossover and mutation then

$$r_{i,j}(k \oplus q) = r_{i \oplus k, j \oplus k}(q) \quad \text{and specifically} \quad r_{i,j}(k) = r_{i,j}(k \oplus 0) = r_{i \oplus k, j \oplus k}(0).$$

This allows one to reorder the elements in s with respect to any particular point in the space. This reordering is equivalent to remapping the variables in the executable equations (See Figure 4). A permutation function, σ, is defined as follows:

$$\sigma_j < s_0, ..., s_{V-1} >^T = < s_{j \oplus 0}, ..., s_{j \oplus (V-1)} >^T$$

where the vectors are treated as columns and $V = 2^L$, the size of the search space. A general operator $\mathcal{M}$ can be defined over s which remaps $s^T M s$ to cover all strings in the space.

$$\mathcal{M}(s) = < (\sigma_0\ s)^T M \sigma_0\ s, ..., (\sigma_{V-1}\ s)^T M \sigma_{V-1}\ s >^T$$

Recall that s denoted the representation of strings in the population during the intermediate phase as the genetic algorithm goes from generation t to $t+1$ (after selection, but before recombination). To complete the cycle and reach a point at which the Vose and Liepins models can be executed in an iterative fashion, fitness information is now explicitly introduced to transform the population at the beginning of iteration $t+1$ to the next intermediate population. A fitness matrix F is defined such that fitness information is stored along the diagonal; the i, i th element is given by $f(i)$ where f is the evaluation function.

The transformation from the vector p^{t+1} to the next intermediate population represented by s^{t+1} is given as follows:

$$s^{t+1} \sim F\mathcal{M}(s^t).$$

Vose and Liepins give equations for calculating the mixing matrix M which not only includes probabilities for 1-point crossover, but also mutation. More complex extension of the Vose and Liepins model include finite population models using Markov chains (Nix and Vose, 1992). Vose (1993) surveys the current state of this research.

8 Other Models of Evolutionary Computation

There are several population based algorithms that are either spin-offs of Holland's genetic algorithm, or which were developed independently. *Evolution Strategies* and *Evolutionary Programming* refer to two computational paradigms that use a population based search.

Evolutionary Programming is based on the early book by L. Fogel, Owens and Walsh (1966) entitled, *Artificial Intelligence Through Simulated Evolution.* The individuals, or "organisms," in this study were finite state machines. Organisms that best solved some target function obtained the opportunity to reproduce. Parents were mutated to create offspring. There has been renewed interest in Evolution Programming as reflected by the 1992 *First Annual Conference on Evolutionary Programming* (Fogel and Atmar 1992).

Evolution Strategies are based on the work of Rechenberg (1973) and Schwefel (1975; 1981) and are discussed in a survey by Bäck, Hoffmeister and Schwefel (1991). Two examples of Evolution Strategies (ES) are the $(\mu+\lambda)$-ES and (μ, λ)-ES. In $(\mu+\lambda)$-ES μ parents produce λ offspring; the population is then reduced again to μ parents by selecting the best solutions from among both the parents and offspring. Thus, parents survive until they are replaced by better solutions. The (μ, λ)-ES is closer to the generational model used in canonical genetic algorithms; offspring replace parents and then undergo selection. Recombination operators for evolutionary strategies also tend to differ from Holland-style crossover, allowing operations such as averaging parameters, for example, to create an offspring.

8.1 Genitor

Genitor (Whitley 1988; 1989) was the first of what Syswerda (1989) has termed "steady state" genetic algorithms. The name "steady state" is somewhat misleading, since these algorithms show more variance than canonical genetic algorithms in the terms of hyperplane sampling behavior (Syswerda, 1991) and are therefore more susceptible to sample error and genetic drift. The advantage is that the best points found in the search are maintained in the population. This results in a more aggressive search that in practice is often quite effective.

There are three differences between Genitor-style algorithms and canonical genetic algorithms. First, reproduction produces one offspring at a time. Two parents are selected for reproduction and produce an offspring that is immediately placed back into the population. The second major difference is in how that offspring is placed back in the population. Offspring do not replace parents, but rather the least fit (or some relatively less fit) member of the population. In Genitor, the worst individual in the population is replaced. The third difference between Genitor and most other forms of genetic algorithms is that fitness is assigned according to rank rather than by fitness proportionate reproduction. Ranking helps to maintain a more constant selective pressure over the course of search.

Goldberg and Deb (1991) have shown replacing the worst member of the population generates much higher selective pressure than random replacement. But higher selective pressure is not the only difference between Genitor and the canonical genetic algorithm. To borrow terminology used by the Evolution Strategy community (as suggested by Larry Eshelman), Genitor is a $(\mu + \lambda)$ strategy while the canonical genetic algorithm is a (μ, λ) strategy. Thus, the accumulation of improved strings in the population is monotonic.

8.2 CHC

Another genetic algorithm that monotonically collects the best strings found so far is the CHC algorithm developed by Larry Eshelman (1991). CHC stands for *Cross generational elitist selection*, *Heterogeneous recombination* (by incest prevention) and *Cataclysmic mutation*, which is used to restart the search when the population starts to converge.

CHC explicitly borrows from the $(\mu + \lambda)$ strategy of Evolution Strategies. After recombination, the N best unique individuals are drawn from the parent population and offspring population to create the next generation. Duplicates are removed from the population. As Goldberg has shown with respect to Genitor, this kind of "survival of the fittest" replacement method already imposes considerable selective pressure, so that there is no real need to use any other selection mechanisms. Thus CHC uses random selection, except restrictions are imposed on which strings are allowed to mate. Strings with binary encodings must be a certain Hamming distance away from one another before they are allowed to reproduce. This form of "incest prevention" is designed to promote diversity. Eshelman also uses a form of uniform crossover called HUX where exactly half of the differing bits are swapped during crossover. CHC is typically run using small population sizes (e.g. 50); thus using uniform crossover in this context is consistent with DeJong and Spears (1991) conjecture that uniform crossover can provide better sampling coverage in the context of small populations.

The rationale behind CHC is to have a very aggressive search (by using monotonic selection through survival of the best strings) and to offset the aggressiveness of the search by using highly disruptive operators such as uniform crossover. With such small population sizes, however, the population converges to the point that it begins to more or less reproduce many of the same strings. At this point the CHC algorithm uses cataclysmic mutation. All strings undergo heavy mutation, except that the best string is preserved intact. After mutation, genetic search is restarted using only crossover.

8.3 Hybrid Algorithms

L. "Dave" Davis states in the *Handbook of Genetic Algorithms,* "Traditional genetic algorithms, although robust, are generally not the most successful optimization algorithm on any particular domain" (1991:59). Davis argues that hybridizing genetic algorithms with the most successful optimization methods for particular problems gives one the best of both worlds: correctly implemented, these algorithms should do no worst than the (usually more traditional) method with which the hybridizing is done. Of course, it also introduces the additional computational overhead of a population based search.

Davis often uses real valued encodings instead of binary encodings, and employs "recombination operators" that may be domain specific. Other researchers, such as Michalewicz (1992) also use nonbinary encodings and specialized operations in combination with a genetic based model of search. Mühlenbein takes a similar opportunistic view of hybridization. In a description of a parallel genetic algorithm Mühlenbein (1991:320) states, after the initial population is created, "Each individual does local hill-climbing." Furthermore, after each offspring is created, "The offspring does local hill-climbing."

Experimental researchers and theoreticians are particularly divided on the issue of hybridization. By adding hill-climbing or hybridizing with some other optimization methods, learning is being added to the evolution process. Coding the learned information back onto the chromosome means that the search utilizes a form of Lamarckian evolution. The chromosomes improved by local hill-climbing or other methods are placed in the genetic population and allowed to compete for reproductive opportunities.

The main criticism is that *if* we wish to preserve the schema processing capabilities of the genetic algorithm, then Lamarckian learning should not be used. Changing information in the offspring inherited from the parents results in a loss of inherited schemata. This alters the statistical information about hyperplane partitions that is implicitly contained in the population. Therefore using local optimization to improve each offspring undermines the genetic algorithm's ability to search via hyperplane sampling.

Despite the theoretical objections, hybrid genetic algorithms typically do well at optimization tasks. There may be several reasons for this. First, the hybrid genetic algorithm is hill-climbing from multiple points in the search space. Unless the objective function is severely multimodal it may be likely that some strings (offspring) will be in the basin of attraction of the global solution, in which case hill-climbing is a fast and effective form of search. Second, a hybrid strategy impairs hyperplane sampling, but does not disrupt it entirely. For example, using local optimization to improve the initial population of strings only biases the initial hyperplane samples, but does not interfere with subsequent hyperplane sampling. Third, in general hill-climbing may find a small number of significant improvements, but may not dramatically change the offspring. In this case, the effects on schemata and hyperplane sampling may be minimal.

9 Hill-climbers or Hyperplane Samplers?

In a recent paper entitled, "How genetic algorithms really work: I. Mutation and Hill-climbing," Mühlenbein shows that an Evolution Strategy algorithm using only mutation works quite well on a relatively simple test suite. Mühlenbein states that for many problems "many *nonstandard* genetic algorithms work well and the *standard* genetic algorithm performs poorly." (1992:24).

This raises a very interesting issue. When is a genetic algorithm a hyperplane sampler and when it is a hill-climber? This is a nontrivial question since it is the hyperplane sampling abilities of genetic algorithms that are usually touted as the source of global sampling. On the other hand, some researchers argue that crossover is unnecessary and that mutation is sufficient for robust and effective search. All the theory concerning hyperplane sampling has been developed with respect to the canonical genetic algorithm. Alternative forms of genetic algorithms often use mechanisms such as monotonic selection of the best strings which could easily lead to increased hill-climbing. Vose's work (personal communication, June 1993) with exact models of the canonical genetic algorithm indicates that even low levels of mutation can have a significant impact on convergence and change the number of fixed points in the space. (For the functions Vose has examined so far mutation always *reduces* the number of fixed points.)

In practice there may be clues as to when hill-climbing is a dominant factor in a search. Hyperplane sampling requires larger populations. Small populations are much more likely to rely on hill-climbing. A population of 20 individuals just doesn't provide very much information about hyperplane partitions, except perhaps very low order hyperplanes (there are only 5 samples of each order-2 hyperplane in a population of 20). Second, very high selective pressure suggests hill-climbing may dominate the search. If the 5 best individuals in a population of 100 strings reproduce 95% of the time, then the effective population size may not be large enough to support hyperplane sampling.

10 Parallel Genetic Algorithms

Part of the biological metaphor used to motivate genetic search is that it is inherently parallel. In natural populations, thousands or even millions of individuals exist in parallel. This suggests a degree of parallelism that is directly proportional to the population size used in genetic search. In this paper, three different ways of exploiting parallelism in genetic algorithms will be reviewed. First, a parallel genetic algorithm similar to the canonical genetic algorithm will be reviewed; next an *Island Model* using distinct subpopulations will be presented. Finally, a fine grain massively parallel implementation that assumes one individual resides at each processor will be explored. It can be shown that the fine grain models are a subclass of cellular automata (Whitley 1993b). Therefore, while these algorithms have been referred to by a number of somewhat awkward names (e.g., fine grain genetic algorithms, or massively parallel genetic algorithms) the name *cellular genetic algorithm* is used in this tutorial.

In each of the following models, strings are mapped to processors in a particular way. Usually this is done in a way that maximizes parallelism while avoiding unnecessary processor communication. However, any of these models could be implemented in massively parallel fashion. What tends to be different is the role of local versus global communication.

10.1 Global Populations with Parallelism

The most direct way to implement a parallel genetic algorithm is to implement something close to a canonical genetic algorithm. The only change that will be made is that selection will be done by *Tournament Selection* (Goldberg, 1990; Goldberg and Deb, 1991).

Tournament selection implements a noisy form of ranking. Recall that the implementation of one generation in a canonical genetic algorithm can be seen as a two step process. First, selection is used to create an intermediate population of duplicate strings selected according to fitness. Second, crossover and mutation are applied to produce the next generation. Instead of using fitness proportionate reproduction or directly using ranking, tournaments are held to fill the intermediate population. Assume two strings are selected out of the current population after evaluation. The best of the two strings is then placed in the intermediate population. This process of randomly selecting two strings from the current population and placing the best in the intermediate population is repeated until the intermediate population is full. Goldberg and Deb (1991) show analytically that this form of tournament selection is the same in expectation as ranking using a linear 2.0 bias. If a winner is chosen probabilistically from a tournament of 2, then the ranking is linear and the bias is proportional to the probability with which the best string is chosen.

With the addition of tournament selection, a parallel form of the canonical genetic algorithm can now be implemented in a fairly direct fashion. Assume the processors are numbered 1 to N/2 and the population size, N, is even; 2 strings reside at each processor. Each processor holds two independent tournaments by randomly sampling strings in the population and each processor then keeps the winners of the two tournaments. The new strings that now reside in the processors represent the intermediate generation. Crossover and evaluation can now occur in parallel.

10.2 Island Models

One motivation for using *Island Models* is to exploit a more coarse grain parallel model. Assume we wish to use 16 processors and have a population of 1,600 strings; or we might wish to use 64 processors and 6,400 strings. One way to do this is to break the total population down into subpopulations of 100 strings each. Each one of these subpopulations could then execute as a normal genetic algorithm. It could be a canonical genetic algorithm, or Genitor, or CHC. Occasionally, perhaps every five generations or so, the subpopulations would swap a few strings. This *migration* allows subpopulations to share genetic material (Whitley and Starkweather, 1990; Gorges-Schleuter, 1991; Tanese 1989; Starkweather et al., 1991.)

Assume for a moment that one executes 16 separate genetic algorithms, each using a population of 100 strings *without migration.* In this case, 16 independent searches occur. Each search will be somewhat different since the initial populations will impose a certain sampling bias; also, genetic drift will tend to drive these populations in different directions. Sampling error and genetic drift are particularly significant factors in small populations and, as previous noted, are even more pronounced in genetic algorithms such as Genitor and CHC when compared to the canonical genetic algorithm.

By introducing migration, the Island Model is able to exploit differences in the various subpopulations; this variation in fact represents a source of genetic diversity. Each subpopulation is an island, and there is some designated way in which genetic material is moved from one island to another. If a large number of strings migrate each generation, then global mixing occurs and local differences between islands will be driven out. If migration is too infrequent, it may not be enough to prevent each small subpopulation from prematurely converging.

10.3 Cellular Genetic Algorithms

Assume we have 2,500 simple processors laid out on a 50x50 2-dimensional grid. Processors communicate only with their immediate neighbors (e.g. north, south, east and west: NSEW). Processors on the edge of the grid wrap around to form a torus. How should one implement a genetic algorithm on such an architecture?

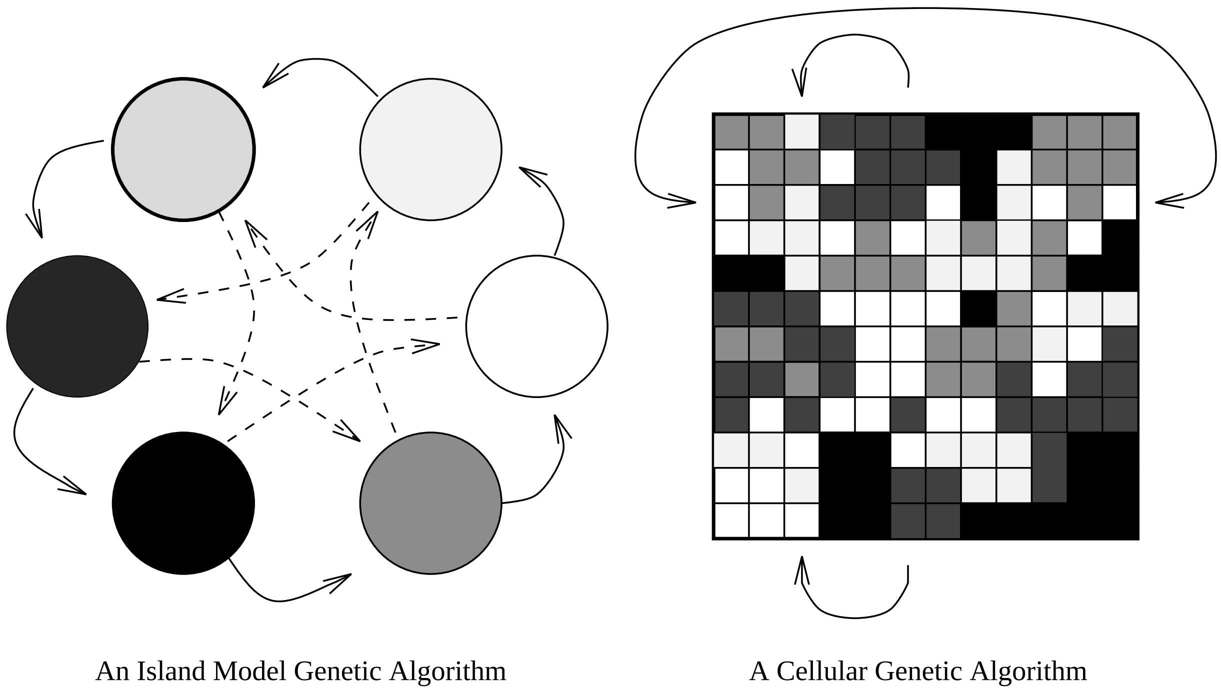

Figure 6: *An example of both an island model and a cellular genetic algorithm. The coloring of the cells in the cellular genetic algorithm represents genetically similar material that forms virtual islands isolated by distance. The arrows in the cellular model indicate that the grid wraps around to form a torus.*

One can obviously assign one string per processor or cell. But global random mating would now seem inappropriate given the communication restrictions. Instead, it is much more practical to have each string (i.e., processor) seek a mate close to home. Each processor can pick the best string in its local neighborhood to mate with, or alternatively, some form of local probabilistic selection could be used. In either case, only one offspring is produced. and becomes the new resident at that processor. Several people have proposed this type of computational model (Manderick and Spiessens, 1989; Collins and Jefferson, 1991; Hillis, 1990; Davidor, 1991). The common theme in cellular genetic algorithms is that selection and mating are typically restricted to a local neighborhood.

There are no explicit islands in the model, but there is the potential for similar effects. Assuming that mating is restricted to adjacent processors, if one neighborhood of strings is 20 or 25 moves away from another neighborhood of strings, these neighborhoods are just as isolated as two subpopulations on separate islands. This kind of separation is referred to as *isolation by distance* (Wright, 1932; Mühlenbein, 1991; Gorges-Schleuter, 1991). Of course, neighbors that are only 4 or 5 moves away have a greater potential for interaction.

After the first random population is evaluated, the pattern of strings over the set of processors should also be random. After a few generations, however, there emerge many small local pockets of similar strings with similar fitness values. Local mating and selection creates local evolutionary trends, again due to sampling effects in the initial population and genetic drift. After several generations, competition between local groups will result in fewer and larger neighborhoods.

11 Conclusions

One thing that is striking about genetic algorithms and the various parallel models is the richness of this form of computation. What may seem like simple changes in the algorithm often result in surprising kinds of emergent behavior. Recent theoretical advances have also improved our understanding of genetic algorithms and have opened to the door to using more advanced analytical methods.

Many other timely issues have not been covered in this tutorial. In particular, the issue of *deception* has not been discussed. The notion of deception, in simplistic terms, deals with conflicting hyperplane competitions that have the potential to either mislead the genetic algorithm, or to simply confound the search because the conflicting hyperplane competitions interfere with the search process. For an introduction to the notion of deception see Goldberg (1987) and Whitley (1991); for a criticism of the work on deception see Grefenstette (1993).

Acknowledgements: This tutorial not only represents information transmitted through scholarly works, but also through conference presentations, personal discussions, debates and even disagreements. My thanks to the people in the genetic algorithm community who have educated me over the years. Any errors or errant interpretations of other works are my own. Work presented in the tutorial was supported by NSF grant IRI-9010546 and the Colorado Advanced Software Institute.

References

Ackley, D. (1987) *A Connectionist Machine for Genetic Hillclimbing.* Kluwer Academic Publishers.

Antonisse, H.J. (1989) A New Interpretation of the Schema Notation that Overturns the Binary Encoding Constraint. *Proc 3rd International Conf on Genetic Algorithms*, Morgan-Kaufmann.

Bäck, T., Hoffmeister, F. and Schwefel, H.P. (1991) A Survey of Evolution Strategies. *Proc. 4th International Conf. on Genetic Algorithms*, Morgan-Kaufmann.

Baker, J. (1985) Adaptive selection methods for genetic algorithms. *Proc. International Conf. on Genetic Algorithms and Their Applications.* J. Grefenstette, ed. Lawrence Erlbaum.

Baker, J. (1987) Reducing Bias and Inefficiency in the Selection Algorithm. *Genetic Algorithms and Their Applications: Proc. Second International Conf.* J. Grefenstette, ed. Lawrence Erlbaum.

Booker, L. (1987) Improving Search in Genetic Algorithms. In, *Genetic Algorithms and Simulating Annealing,* L. Davis, ed. Morgan Kaufman, pp. 61-73.

Bridges, C. and Goldberg, D. (1987) An analysis of reproduction and crossover in a binary-coded genetic Algorithm. *Proc. 2nd International Conf. on Genetic Algorithms and Their Applications.* J. Grefenstette, ed. Lawrence Erlbaum.

Collins, R. and Jefferson, D. (1991) Selection in Massively Parallel Genetic Algorithms. *Proc. 4th International Conf. on Genetic Algorithms*, Morgan-Kaufmann, pp 249-256.

Davidor, Y. (1991) A Naturally Occurring Niche & Species Phenomenon: The Model and First Results. *Proc 4th International Conf on Genetic Algorithms*, Morgan-Kaufmann, pp 257-263.

Davis, L.D. (1991) *Handbook of Genetic Algorithms.* Van Nostrand Reinhold.

DeJong, K. (1975) An Analysis of the Behavior of a Class of Genetic Adaptive Systems. PhD Dissertation. Dept. of Computer and Communication Sciences, Univ. of Michigan, Ann Arbor.

Eshelman, L. (1991) The CHC Adaptive Search Algorithm. *Foundations of Genetic Algorithms*, G. Rawlins, ed. Morgan-Kaufmann. pp 256-283.

Fitzpatrick, J.M. and Grefenstette, J.J. (1988) Genetic Algorithms in Noisy Environments. *Machine Learning, 3(2/3):* 101-120.

Fogel, L.J., Owens, A.J., and Walsh, M.J. (1966) *Artificial Intelligence Through Simulated Evolution.* John Wiley.

Fogel, D., and Atmar, J.W., eds. (1992) *First Annual Conference on Evolutionary Programming.*

Goldberg, D. and Bridges, C. (1990) An Analysis of a Reordering Operator on a GA-Hard Problem. *Biological Cybernetics,* 62:397-405.

Goldberg, D. (1987) Simple Genetic Algorithms and the Minimal, Deceptive Problem. In, *Genetic Algorithms and Simulated Annealing,* L. Davis, ed., Pitman.

Goldberg, D. (1989) *Genetic Algorithms in Search, Optimization and Machine Learning.* Reading, MA: Addison-Wesley.

Goldberg, D. (1990) A Note on Boltzmann Tournament Selection for Genetic Algorithms and Population-oriented Simulated Annealing. TCGA 90003, Engineering Mechanics, Univ. Alabama.

Goldberg, D. (1991) The Theory of Virtual Alphabets. *Parallel Problem Solving from Nature*, Springer Verlag.

Goldberg, D., and Deb, K. (1991) A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. *Foundations of Genetic Algorithms*, G. Rawlins, ed. Morgan-Kaufmann. pp 69-93.

Gorges-Schleuter, M. (1991) Explicit Parallelism of Genetic Algorithms through Population Structures. *Parallel Problem Solving from Nature*, Springer Verlag, pp 150-159.

Grefenstette, J.J. (1986) Optimization of Control Parameters for Genetic Algorithms. *IEEE Trans. Systems, Man, and Cybernetics, 16(1):* 122-128.

Grefenstette, J.J. and Baker, J. (1989) How Genetic Algorithms Work: A Critical Look at Implicit Parallelism. *Proc 3rd International Conf on Genetic Algorithms*, Morgan-Kaufmann.

Grefenstette, J.J. (1993) Deception Considered Harmful. *Foundations of Genetic Algorithms -2-,* D. Whitley, ed., Morgan Kaufmann. pp: 75-91.

Hillis, D. (1990) Co-Evolving Parasites Improve Simulated Evolution as an Optimizing Procedure. *Physica D 42*, pp 228-234.

Holland, J. (1975) Adaptation In Natural and Artificial Systems. University of Michigan Press.

Liepins, G. and Vose, M. (1990) Representation Issues in Genetic Algorithms. Journal of Experimental and Theoretical Artificial Intelligence, 2:101-115.

Manderick, B. and Spiessens P. (1989) Fine Grained Parallel Genetic Algorithms. *Proc 3rd International Conf on Genetic Algorithms*, Morgan-Kaufmann, pp 428-433.

Michalewicz, Z. (1992) *Genetic Algorithms + Data Structures = Evolutionary Programs.* Springer-Verlag, AI Series, New York.

Mühlenbein, H. (1991) Evolution in Time and Space - The Parallel Genetic Algorithm. *Foundations of Genetic Algorithms*, G. Rawlins, ed. Morgan-Kaufmann. pp 316-337.

Mühlenbein, H. (1992) How genetic algorithms really work: I. Mutation and Hillclimbing, *Parallel Problem Solving from Nature -2-*, R. Männer and B. Manderick, eds. North Holland.

Nix, A. and Vose, M. (1992) Modeling Genetic Algorithms with Markov Chains. *Annals of Mathematics and Artificial Intelligence.* 5:79-88.

Rechenberg, I. (1973) *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Frommann-Holzboog Verland, Stuttgart.

Schaffer, J.D. (1987) Some Effects of Selection Procedures on Hyperplane Sampling by Genetic Algorithms. In, *Genetic Algorithms and Simulated Annealing,* L. Davis, ed. Pitman.

Schaffer, J.D., and Eshelman, L. (1993) Real-Coded Genetic Algorithms and Interval Schemata. *Foundations of Genetic Algorithms -2-*, D. Whitley, ed. Morgan-Kaufmann.

Schwefel, H.P. (1975) *Evolutionsstrategie und numerische Optimierung.* Dissertation, Technische Universität Berlin.

Schwefel, H.P. (1981) *Numerical Optimization of Computer Models.* Wiley.

Spears, W. and DeJong, K. (1991) An Analysis of Multi-Point Crossover. *Foundations of Genetic Algorithms*, G. Rawlins, ed. Morgan-Kaufmann.

Syswerda, G. (1989) Uniform Crossover in Genetic Algorithms. *Proc 3rd International Conf on Genetic Algorithms*, Morgan-Kaufmann, pp 2-9.

Syswerda, G. (1991) A Study of Reproduction in Generational and Steady-State Genetic Algorithms. *Foundations of Genetic Algorithms*, G. Rawlins, ed. Morgan-Kaufmann. pp 94-101.

Starkweather, T., Whitley, D., and Mathias, K. (1991) Optimization Using Distributed Genetic Algorithms. *Parallel Problem Solving from Nature*, Springer Verlag.

Tanese, R. (1989) Distributed Genetic Algorithms. *Proc 3rd International Conf on Genetic Algorithms*, Morgan-Kaufmann, pp 434-439.

Vose, M. (1993) Modeling Simple Genetic Algorithms. *Foundations of Genetic Algorithms -2-,* D. Whitley, ed., Morgan Kaufmann. pp: 63-73.

Vose, M. and Liepins, G. (1991) Punctuated Equilibria in Genetic Search. *Complex Systems* 5:31-44.

Whitley, D. (1989) The GENITOR Algorithm and Selective Pressure. *Proc 3rd International Conf on Genetic Algorithms*, Morgan-Kaufmann, pp 116-121.

Whitley, D. (1991) Fundamental Principles of Deception in Genetic Search. *Foundations of Genetic Algorithms.* G. Rawlins, ed. Morgan Kaufmann.

Whitley, D. (1993a) An Executable Model of a Simple Genetic Algorithm. *Foundations of Genetic Algorithms -2-.* D. Whitley, ed. Morgan Kaufmann.

Whitley, D. (1993b) Cellular Genetic Algorithms. *Proc. 5th International Conference on Genetic Algorithms.* Morgan Kaufmann.

Whitley, D., and Kauth, J. (1988) GENITOR: a Different Genetic Algorithm. *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, Denver, CO. pp 118-130.

Whitley, D. and Starkweather, T. (1990) Genitor II: a Distributed Genetic Algorithm. *Journal Expt. Theor. Artif. Intell.*, 2:189-214

Whitley, D., Das, R., and Crabb, C. (1992) Tracking Primary Hyperplane Competitors During Genetic Search. *Annals of Mathematics and Artificial Intelligence.* 6:367-388.

Winston, P. (1992) *Artificial Intelligence,* Third Edition. Addison-Wesley.

Wright, A. (1991) Genetic Algorithms for Real Parameter Optimization. *Foundations of Genetic Algorithms.* G. Rawlins, ed. Morgan Kaufmann.

Wright, S. (1932) The Roles of Mutation, Inbreeding, Crossbreeding, and Selection in Evolution. *Proc. 6th Int. Congr. on Genetics*, pp 356-366.