21 Genetic Algorithms — A Survey of Models and Methods

Darrell Whitley[1] · *Andrew M. Sutton*[2]
[1]Department of Computer Science, Colorado State University, Fort Collins, CO, USA
whitley@cs.colostate.edu
[2]Department of Computer Science, Colorado State University, Fort Collins, CO, USA
sutton@cs.colostate.edu

1 The Basics of Genetic Algorithms *638*

2 Interpretations and Criticisms of the Schema Theorem *653*

3 Infinite Population Models of Simple Genetic Algorithms *655*

4 The Markov Model for Finite Populations *658*

5 Theory Versus Practice *659*

6 Where Genetic Algorithms Fail *662*

7 An Example of Genetic Algorithms for Resource Scheduling *664*

8 Conclusions *669*

G. Rozenberg et al. (eds.), *Handbook of Natural Computing*, DOI 10.1007/978-3-540-92910-9_21,
© Springer-Verlag Berlin Heidelberg 2012

Abstract

This chapter first reviews the simple genetic algorithm. Mathematical models of the genetic algorithm are also reviewed, including the schema theorem, exact infinite population models, and exact Markov models for finite populations. The use of bit representations, including Gray encodings and binary encodings, is discussed. Selection, including roulette wheel selection, rank-based selection, and tournament selection, is also described. This chapter then reviews other forms of genetic algorithms, including the steady-state Genitor algorithm and the CHC (cross-generational elitist selection, heterogenous recombination, and cataclysmic mutation) algorithm. Finally, landscape structures that can cause genetic algorithms to fail are looked at, and an application of genetic algorithms in the domain of resource scheduling, where genetic algorithms have been highly successful, is also presented.

1 The Basics of Genetic Algorithms

Genetic algorithms were the first form of evolutionary algorithms to be widely accepted across a diverse set of disciplines ranging from operations research to artificial intelligence. Today, genetic algorithms and other evolutionary algorithms are routinely used as search and optimization tools for engineering and scientific applications.

Genetic algorithms were largely developed by John Holland and his students in the 1960s, 1970s, and 1980s. The term "genetic algorithms" came into common usage with the publication of Ken De Jong's 1975 PhD dissertation. Holland's classic 1975 book, *Adaptation in Natural and Artificial Systems* (Holland 1975), used the term "genetic plans" rather than "genetic algorithms." In the mid-1980s genetic algorithms started to reach other research communities. An explosion of research in genetic algorithms came soon after a similar explosion of research in artificial neural networks. Both areas of research draw inspiration from biological systems as a computational model.

There are several forms of evolutionary algorithms that use simulated evolution as a mechanism to solve problems where the problems can be expressed as a search or optimization problem. Other areas of evolutionary computation, such as evolutionary programming, evolution strategies, and genetic programming, are discussed in other chapters. Compared to other evolutionary algorithms, genetic algorithms put a great deal of emphasis on the combined interactions of selection, recombination, and mutation acting on a genotype. In most early forms of genetic algorithms, recombination was emphasized over mutation. This emphasis still endures in some forms of the genetic algorithm. However, hybridization of genetic algorithms with local search is also very common.

Genetic algorithms emphasize the use of a "genotype" that is decoded and evaluated. These genotypes are often simple data structures. In most early applications, the genotypes (which are sometimes thought of as artificial chromosomes) are bit strings which can be recombined in a simplified form of "sexual reproduction" and can be mutated by bit flips. Because of the bit encoding, it is sometimes common to think of genetic algorithms as function optimizers. However, this does not mean that they yield globally optimal solutions. Instead, Holland (in the introduction to the 1992 edition of his book *Adaptation in Natural and Artificial Systems* Holland (1992)) and De Jong (1993) have both emphasized that these algorithms find competitive solutions rather than optimal solutions, but both also suggest that it is probably best to view genetic algorithms not as an optimization process, but rather as adaptive systems.

At the risk of overemphasizing optimization, an example application from function optimization provides a useful vehicle for explaining certain aspects of these algorithms. For example, consider a control or production process which must be optimized with respect to some evaluation criterion. The input domain to the problem, $\mathcal{X}$, is the set of all possible configurations to the system. Given an evaluation function f, one seeks to maximize (or minimize) $f(x),\ x \in \mathcal{X}$. The input domain can be characterized in many ways. For instance, if $\mathcal{X} = \{0, 1, \ldots, 2^L - 1\}$ is the search space then our input domain might be the set of binary strings of length L. If we elect to use a "real-valued" encoding, a floating point representation might be used, but in any discrete computer implementation, the input domain is still a finite set.

In the evolutionary algorithm community, it has become common to refer to the evaluation function as a *fitness* function. Technically, one might argue that the objective function $f(x)$ is the evaluation function, and that the fitness function is a second function more closely linked with selection. For example, we might assign fitness to an artificial chromosome based on its rank in the population, assigning a "fitness" of 2.0 to the best member of the population and a fitness of 1.0 to the median member of the population. Thus, the fitness function and evaluation function are not always the same. Nevertheless in common usage, the term fitness function has become a surrogate name for the evaluation function.

Returning to our simple example of an optimization problem, assume that there is a system with three parameters we can control: temperature, pressure, and duration. These are in effect three inputs to a black box optimization problem where inputs from the domain of the function are fed into the black box and a value from the co-domain of the function is produced as an output. The system's response could be a *quality* measure dependent on the temperature, pressure, and duration parameters.

One could represent the three parameters using a vector of three real-valued parameters, such as

$$\langle 32.56, 18.21, 9.83 \rangle$$

or the three parameters could be represented as bit strings, such as

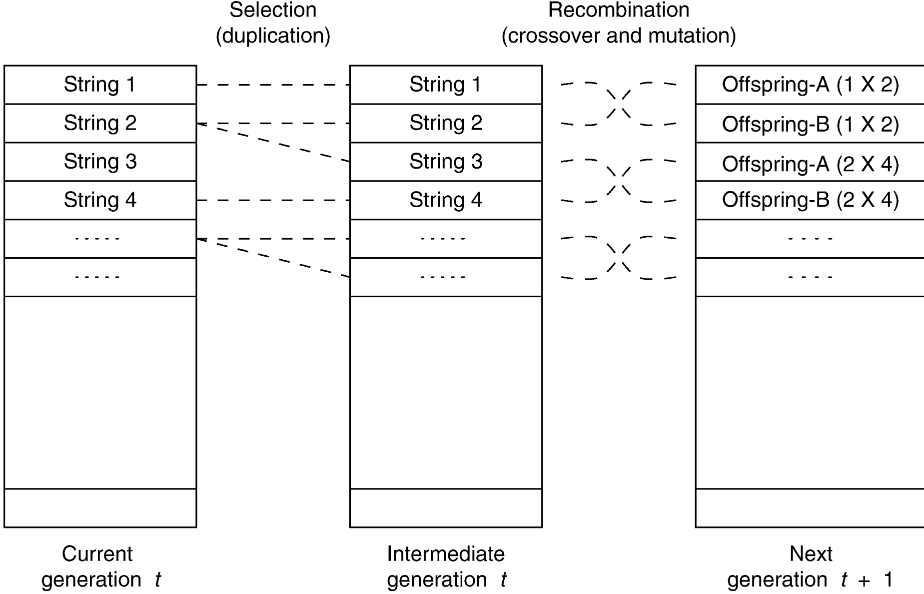
$$\langle 000111010100, 110100101101, 001001101011 \rangle.$$

Using an explicit bit encoding automatically raises the question as to what precision should be used, and what should be the mapping between bit strings and real values. Picking the right precision can have a large impact on performance. Historically, genetic algorithms have typically been implemented using low precision; using 10 bits per parameter is common in many test functions. Using high precision (e.g., 32 bits per parameters) generally results in poor performance, and real-valued representations should be considered if high precision is required.

Recombination is central to genetic algorithms. For now, assume that the artificial chromosomes are bit strings and that 1-point crossover will be used. 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 crossover point, a 1-point recombination might occur as follows.

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

Fig. 1
One generation is broken down into a selection phase and a recombination phase. This figure shows strings being assigned into adjacent slots during selection. It is assumed that selection randomizes the location of strings in the intermediate population. Mutation can be applied before or after crossover.



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

```
11010yxxyyyxyxxy        and        yxyyx01100101101
```

Parameter boundaries are ignored. We can also apply a mutation operator. For each bit in the population, mutate with some low probability p_m. For bit strings of length L a mutation rate of $1/L$ is often suggested; for separable problems, this rate of mutation can be shown to be particularly effective.

In addition to mutation and recombination operators, the other key component to a genetic algorithm (or any other evolutionary algorithm) is the selection mechanism. For a genetic algorithm, it is instructive to view the mechanism by which a standard genetic algorithm moves from one generation to the next as a two-stage process. Selection is applied to the current population to create an *intermediate generation*, as shown in Fig. 1. Then, recombination and mutation are applied to the intermediate population to create the *next generation*. The process of going from the current population to the next population constitutes one generation in the execution of a genetic algorithm.

We will assume that our selection mechanism is *tournament selection*. This is not the mechanism used in early genetic algorithms, but it is commonly used today. A simple version of tournament selection randomly samples two strings out of the initial population, then "selects" the best of the two strings to insert into the intermediate generation. If the population size is N, then this must be done N times to construct the intermediate population. We can then apply mutation and crossover to the intermediate population.

1.1 The Canonical Holland-Style Genetic Algorithm

In a Holland-style genetic algorithm, tournament selection is not used; instead *fitness proportional selection* is used. Proportional 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. Under fitness proportional selection, fitness is usually being maximized. All early forms of genetic algorithms used fitness proportional selection. Theoretical models also often assume fitness proportional selection because it is easy to model mathematically.

The fitness value f_i may be the direct output of an evaluation function, or it may be scaled in some way. 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 that is proportional to their fitness.

For a maximization problem, if $f_i/\bar{f}$ is used as a measure of fitness for string i, then strings where $f_i/\bar{f}$ is greater than 1.0 have above average fitness and strings where $f_i/\bar{f}$ is less than 1.0 have below average fitness. We would like to allocate more chances to reproduce to those strings that are above average. One way to do this is to directly duplicate those strings that are above average. To do so, f_i is broken into an integer part, x_i, and a remainder, r_i. We subsequently place x_i duplicates of string i directly into the intermediate population and place 1 additional copy with probability r_i.

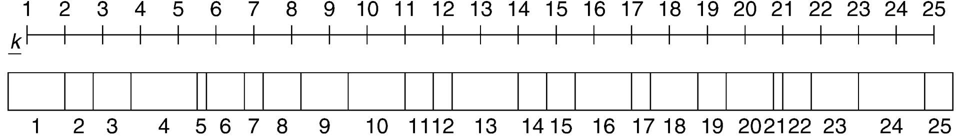
This is efficiently implemented using *stochastic universal sampling*. Assume that the population is laid out in random order as a number line where each individual is assigned space on the number line in proportion to fitness. Now generate a random number between 0 and 1 denoted by k. Next, consider the position of the number $i+k$ for all integers i from 1 to N where N is the population size. Each number $i+k$ will fall on the number line in some space corresponding to a member of the population. The position of the N numbers $i+k$ for $i = 1$ to N in effect selects the members of the intermediate population. This is illustrated in ➋ *Fig. 2*.

This method is also known as roulette wheel selection: the space assigned to the 25 members of the population in ➋ *Fig. 2* could be laid out along the circumference of a circle representing the roulette wheel. The choice of k in effect "spins" the roulette wheel (since only the fractional part of the spins effects the outcome) and determines the position of the evenly spaced pointers, thereby simultaneously picking all N members of the intermediate population. The resulting selection is unbiased (Baker 1987).

After selection has been executed, the construction of the intermediate population is complete. The *next generation* of the population is created from the intermediate population.

Fig. 2

Stochastic Universal Sampling. The fitnesses of the population can be seen as being laid out on a number line in random order as shown at the bottom of the figure. A single random value, $0 \leq k \leq 1$, shifts the uniformly spaced "pointers" which now select the member of the next intermediate population.



Crossover is applied to randomly paired strings with a probability denoted p_c. The offsprings created by "recombination" go into the next generation (thus replacing the parents). If no recombination occurs, the parents can pass directly into the next generation. However, parents that do not undergo recombination may still be changed due to the application of a mutation operator.

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

One way to characterize *selective pressure* is to calculate the selection bias toward the best individual in the population compared to the average fitness or the median fitness of the population. This is more useful for "rank-based" than fitness proportional, but it still serves to highlight the fact that fitness proportional selection can result in a number of problems. First, selective pressure can be too strong: too many duplicates are sometimes allocated to the best individual(s) in the population in the early generations of the search. Second, in the later stages of search, as the individuals in the population improve over time, there tends to be less variation in fitness, with more individuals (including the best individual(s)) being close to the population average. As the population average fitness increases, the fitness variance decreases and the corresponding uniformity in fitness values causes selective pressure to go down. When this happens, the search stagnates.

Because of these problems, few implementations of genetic algorithms use simple fitness proportionate reproduction. At the very least, the fitness may be rescaled before fitness proportionate reproduction. Fitness scaling mechanisms are discussed in detail in Goldberg's textbook *Genetic Algorithm in Search, Optimization and Machine Learning* (Goldberg 1989b). In recent years, the use of rank-based selection and in particular tournament selection has become common.

1.2 Rank-Assigned Tournament Selection

Selection can be based on fitness assigned according to rank, or relative fitness. This can be done explicitly. Assume the population is sorted by the evaluation function. A linear ranking mechanism with selective pressure Z (where $1 < Z \leq 2$) allocates a fitness bias of Z to the top-ranked individual, $2-Z$ to the bottom-ranked individual, and a fitness bias of 1.0 to the median individual. Note that the difference in selective pressure between the best and the worst member of the population is constant and independent of how many generations have passed. This has the effect of making selective pressure more constant and controlled. Code for linear ranking is given by Whitley (1989).

The use of rank-based selection also transforms the search into what is known as an *ordinal optimization method*. The exact evaluation of sample points from the search space is no longer important. All that is important is the relative value (or relative fitness) of the strings representing sample points from the search space. Note that the fitness function can be an ordinal measure, but the evaluation function itself is not ordinal.

Ordinal optimization may also sometimes relax computation requirements (Ho et al. 1992; Ho 1994). It may be easier to determine $f(x_1) < f(x_2)$ than to exactly compute $f(x_1)$ and $f(x_2)$. An approximate evaluation or even noisy evaluation may still provide enough information to determine the relative ranking, or to determine the relative ranking with high probability.

This can be particularly important when the evaluation is a simulation rather than an exact mathematical objective function.

A fast and easy way to implement ranking is *tournament selection* (Goldberg 1990; Goldberg and Deb 1991). We have already seen a simple form of tournament selection which selects two strings at random and places the best in the intermediate population. In expectation, every string is sampled twice. The best string wins both tournaments and gets two copies in the intermediate population. The median string wins one and loses one and gets one copy in the intermediate population. The worst string loses both tournaments and does not reproduce. In expectation, this produces a linear ranking with a selective pressure of 2.0 toward the best individual. If the winner of the tournament is placed in the intermediate population with probability $0.5 < p < 1.0$, then the selective pressure is less than 2.0. If a tournament size larger than 2 is used and the winner is chosen deterministically, then the selective pressure is greater than 2.0.

We can first generalize tournament selection to generate a linear selection bias less than 2, or a nonlinear bias greater than 2. For stronger selection, pick $t>2$ individuals uniformly at random and select the best one of those individuals for reproduction. The process is repeated the number of times necessary to achieve the desired size of the intermediate population. The *tournament size*, t, has a direct correspondence with selective pressure because the expected number of times we sample the fittest individual in the intermediate population is t.

The original motivation behind the modern versions of tournament selection is that the algorithm is embarrassingly parallel because each tournament is independent (Suh and Gucht 1987). If one has the same number of processors as the population size, all N tournaments can be run simultaneously in parallel and constructing an intermediate population would require the same amount of time as executing a single tournament. Each processor samples the population t times and selects the best of those individuals.

Poli (2005) has noted that there are two factors that lead to the loss of diversity in regular tournament selection. Due to the randomness of tournaments, some individuals might not get *sampled* to participate in a tournament at all. Other individuals might be sampled but not be *selected* for the intermediate population because the individual loses the tournament.

One way to think of tournament selection (where $t = 2$) is a comparison of strings based on two vectors of random numbers.

```
vector A:  5 4 2 6 2 2 8 3
vector B:  5 3 6 5 4 3 4 6
```

The integer stored at $A(i)$ identifies a member of the population. We will assume that the lower integers correspond to better evaluations. During the ith tournament, $A(i)$ is compared to $B(i)$ to determine the winner of the tournament. Note that in this example, if the integers 1–8 represent the population members, then individuals 1 and 7 are not *sampled*. Poli points out that the number of individuals neglected in the first decision if two offsprings are produced by recombination is $P(1 - 1/P)^{TP}$ where P is the population size and T is the tournament size. Expressed another way, what Poli calculates is the expected number of population members that fail to appear in vector A or B if these were used to keep track of tournament selection during one generation (or the amount of time needed to sample P new points in the search space). It is also possible to under-sample. If the tournament size is 2, in expectation, all members of the population would be sampled twice if tournament selection were used in combination with a generational genetic algorithm. However, some members of the population might be sampled only once.

Other forms of selection do not have this problem. Universal stochastic sampling guarantees that *all* individuals with above-average fitness get to reproduce, and all individuals below average get a chance to reproduce with a probability proportional to their fitness. Note that fitness in this case could be based on the evaluation function or it could still be a rank-based assignment of space on the roulette wheel in *Fig. 2*.

Unbiased tournament selection (Sokolov and Whitley 2005) eliminates loss of diversity related to the failure to sample. Unbiased tournament selection also reduces the variance associated with how often an individual is sampled in one generation (variance reduction tournament selection might have been a more accurate name, but is somewhat more cumbersome). Unbiased tournament selection operates much like regular tournament selection except that permutations are used in place of random sampling during tournament construction. Rather than randomly sampling the population (or using random vectors of numbers to sample the population), t random permutations are generated. The ith element in each permutation indexes to a population member: the t population members pointed to by the ith element of each permutation form a tournament. An effective improvement is to use only $t-1$ permutations; the indices from 1 to P in sorted order can serve as one permutation (e.g., as generated in a for-loop). Other permutations can be constructed by sequentially sampling the population without replacement.

An example of unbiased tournament selection for tournament size $t = 3$ can be seen in *Fig. 3*. Assume that $f(x) = x$ and that lower numbers correspond to better individuals; then the last row, labeled "Winners," presents the resulting intermediate population as chosen by unbiased tournament selection. Note that permutation 1 is in sorted order. Recall that the selective pressure is partially controlled through the number of permutations.

Unbiased stochastic tournament selection can also be employed for selective pressure $S < 2$. We align two permutations, perform a pairwise comparison of elements, but select the best with $0.5 < p_s < 1.0$. For selective pressure less than or equal to 2 we also enforce the constraint that the ith element of the second permutation is not equal to i. This provides a guarantee that every individual will participate in a tournament twice. This is easily enforced: when generating the ith element of a permutation, we withhold the ith individual from consideration. There are not enough degrees of freedom to guarantee that this constraint holds for the last element of the permutation, but a violation can be fixed by swapping the last element with its immediate predecessor. The constraint is not enforced for $t>2$ as too much overhead is involved.

To illustrate the congruence between "formula-based" ranking and tournament selection, a simulation was performed to compare three methods of linear ranking: (1) by random sampling using an explicit mathematical formula, (2) using regular tournament selection, and (3) using unbiased tournament selection. The number of times an individual with each rank was selected was measured as the search progressed. *Figure 4* presents these measurements

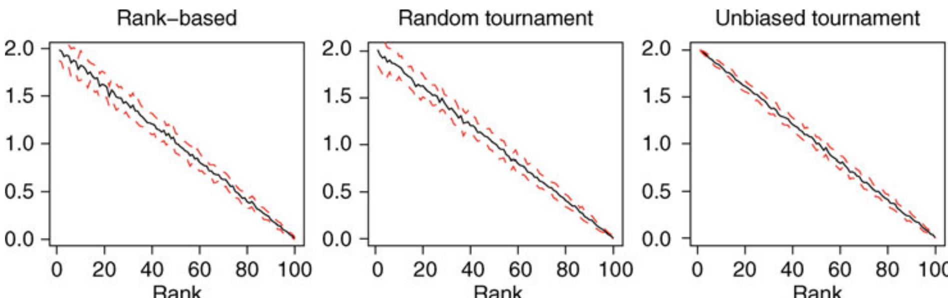
Fig. 3

Unbiased tournament selection with tournament size t = 3. In this case, each column represents a slot in the population; the best individual in each column wins the tournament for that slot.

```
Permutation 1: 1  2  3  4  5  6  7  8
Permutation 2: 6  2  1  5  8  4  3  7
Permutation 3: 2  8  1  4  3  6  7  5
      Winners: 1  2  1  4  3  4  3  5
```

Fig. 4

The number of times an individual with a particular rank was selected for recombination. The results are presented for rank-based, regular, and unbiased tournament selection schemes. *Solid line* is the mean computed across 50 runs with 100 generations/run. *Dashed lines* lie one standard deviation away from the mean.



for a population of size 100 averaged over 50 runs with 100 generations each. A selective pressure of 2.0 was used. The solid line represented the mean and the two dashed lines were one standard deviation away from it. Notice that linear ranking behavior is problem independent because the only measure used in the calculations is the relative goodness of solutions.

As can be seen, formula-based ranking and regular tournament selection are essentially the same. On the other hand, unbiased tournament selection controls for variance by insuring the actual sample in each case is within one integer value of its expected value.

1.3 Different Forms of Representation, Crossover, and Mutation

One of the long-standing debates in the field of evolutionary algorithms involves the use of binary versus real-valued encodings for parameter optimization problems. The genetic algorithms community has largely emphasized bit representations. The main argument for bit encodings is that this representation decomposes the problem into the largest number of smallest possible building blocks and that a genetic algorithm works by processing these building blocks. This viewpoint, which was widely accepted 10 years ago, is now considered to be controversial. On the other hand, the evolution strategies community (Schwefel 1981, 1995; Bäck 1996) has emphasized the use of real-valued encodings for parameter optimization problems. Application-oriented researchers were also among the first in the genetic algorithms community to experiment with real-valued encodings (Davis 1991; Schaffer and Eshelman 1993).

Real-valued encodings can be recombined using simple crossover. The assignment of parameter values can be directly inherited from one parent or the other. But other forms of recombination are also possible, such as blending crossover that averages the real-valued parameter values. This idea might be generalized by considering the two parents as points p_1 and p_2 in a multidimensional space. These points might be combined as a weighted average (using some bias $0 \leq \alpha \leq 1$) such that an offspring might be produced by computing a new point p_0 in the following way:

$$p_0 = \alpha p_1 + (1 - \alpha) p_2$$

One can also compute a centroid associated with multiple parents.

Because real-valued encodings are commonly used in the evolution strategy community, the reader is encouraged to consult the chapter on evolution strategies. This chapter will focus on binary versus Gray coded bit representations.

1.3.1 Binary Versus Gray Representations

A related issue that has long been debated in the evolutionary algorithms community is the relative merit of Gray codes versus standard binary representations for parameter optimization problems. Generally, "Gray code" refers to the *standard binary reflected Gray code* (Bitner et al. 1976); but there are exponentially many possible Gray codes for a particular Hamming space. A Gray code is a bit encoding where adjacent integers are also Hamming distance 1 neighbors in Hamming space.

There are at least two basic ways to compute a Gray code. In general, a Gray matrix can be used that acts as a transform of a standard binary string. The standard binary reflected Gray code encoding matrix has 1s along the diagonal and 1s along the upper minor diagonal and 0s everywhere else. The matrix for decoding the standard binary reflected Gray code has 1s along the diagonal and all of the upper triangle of the matrix is composed of 1 bits. The lower triangle of the matrix is composed of 0 bits. The following is an example of the Graying matrix G (on the left) and the deGraying matrix D (on the right).

$$G = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad D = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Given a binary string (vector) b, a Graying matrix G, and a deGraying matrix D, bG produces a binary string (vector), which is the binary reflected Gray code of string b. Using the deGraying matrix, $(bG)D = b$.

A faster way to produce the standard binary reflected Gray code is to shift vector b by 1 bit, which will be denoted by the function $s(b)$. Using exclusive-or (denoted by $\oplus$) over the bits in b and $s(b)$,

$$b \oplus s(b) = bG$$

This produces a string with $L+1$ bits; the last bit is deleted. In implementation, no shift is necessary and one can have exclusive-or pairs of adjacent bits.

Assume that bit strings are assigned to the corner of a hypercube so that the strings assigned to the adjacent points in the hypercube differ by one bit. In general, a Gray code is a permutation of the corners of the hypercube, such that the resulting ordering forms a path that visits all of the points in the space and every point along the path differs from the point before and after it by a single bit flip.

There is no known closed form for computing the number of Gray codes for strings of length L. Given any Gray code (such as the standard binary reflected Gray code), one can (1) permute the order of the bits themselves to create $L!$ different Gray codes, and for each of these Gray codes one can (2) shift the integer assignment to the strings by using modular addition of a constant and still maintain a Gray code. This can be done in 2^L different ways. If all the Gray codes produced by these permutation and shift operations were unique, there would be at least $2^L(L!)$ Gray codes. Different Gray codes have different neighborhood

structures. In practice, it is easy to use the trick of shifting the Gray code by a constant to explore different Gray codes with different neighborhoods. But one can also prove that the neighborhood structure repeats after a shift of $2^L/4$.

The "reflected" part of the standard binary reflected Gray code derives from the following observation. Assume that we have a sequence of 2^L strings and we wish to extend that sequence to 2^{L+1} strings. Let the sequence of 2^L strings be denoted by

$$a\ b\ c\ \ldots\ x\ y\ z$$

Without loss of generality, assume the strings can be decoded to correspond to integers such that $a = 0, b = 1, c = 2, \ldots, y = (2^L - 2), z = (2^L - 1)$. In standard binary representations, we extend the string by duplicating the sequence and appending a 0 to each string in the first half of the sequence and a 1 to each string in the second half of the sequence. Thus, if we start with a standard binary sequence of strings, then the following is also a standard binary encoding:

$$0a\ 0b\ 0c\ \ldots 0x\ 0y\ 0z\ 1a\ 1b\ 1c\ \ldots 1x\ 1y\ 1z$$

However, in a reflected Gray code, we extend the string by duplicating and reversing the sequence; then a 0 is appended to all the strings in the original sequence and a 1 appended to the string in the reflected or reversed sequence.

$$0a\ 0b\ 0c\ \ldots 0x\ 0y\ 0z\ 1z\ 1y\ 1x\ \ldots 1c\ 1b\ 1a$$

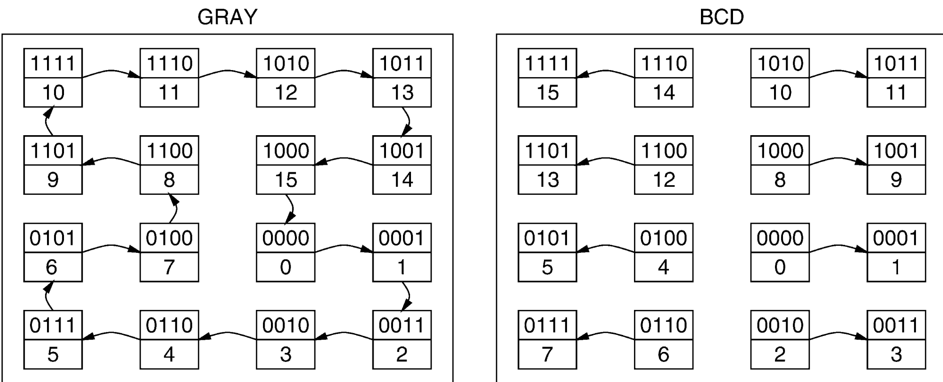
It is easy to verify that if the original sequence is a Gray code, then the reflected expansion is also a Gray code.

Over all possible discrete functions that can be mapped onto bit strings, the space of any and all Gray code representations and the space of binary representations must be identical. This is another example of what has come to be known as a "no-free-lunch" result (Wolpert and Macready 1995; Radcliffe and Surry 1995; Whitley and Rowe 2008). The empirical evidence suggests, however, that Gray codes are often superior to binary encodings. It has long been known that Gray codes remove Hamming cliffs, where adjacent integers are represented by complementary bit strings: for example, 7 and 8 encoded as 0111 and 1000. Whitley et al. (1996) first made the rather simple observation that every Gray code must preserve the connectivity of the original real-valued functions and that this impacts the number of local optima that exists under Gray and binary representation. This is illustrated in *Fig. 5*.

A consequence of the connectivity of the Gray code representation is that for every 1-dimensional parameter optimization problem, the number of optima in the Gray coded space must be less than or equal to the number of optima in the original real-valued function. Binary encodings offer no such guarantee. Binary encodings destroy half of the connectivity of the original real-valued function; thus, given a large basin of attraction with a globally competitive local optimum, many of the points near the optimum that are not optima become new local optima under a binary representation. The theoretical and empirical evidence suggests (Whitley 1999) that for parameter optimization problems with a bounded number of optima, Gray codes are better than binary in the sense that Gray codes induce fewer optima.

As for the debate over whether Gray bit encodings are better or worse than real-coded representations, the evidence is very unclear. If high precision is required, real-valued encodings are usually better. In other cases, a lower precision Gray code outperforms a real-valued encoding. This also means that an accurate comparison is difficult. The same parameter optimization problem encoded with high precision real-valued representation versus a low

Fig. 5
Adjacency in 4-bit Hamming space for Gray and standard binary encodings. The binary representation destroys half of the connectivity of the original function.



precision bit encoding results in two different search spaces. There are no clear theoretical or empirical answers to this question.

1.3.2 *N*-Point and Uniform Recombination

Early mathematical models of genetic algorithms often employed 1-point crossover in conjunction with binary strings. This bias again seems to be connected to the fact that 1-point crossover is easy to model mathematically. It is easy to see that 2-point crossover has advantages over 1-point crossover. In 1-point crossover, bits at opposite ends of the encoding are virtually always separated by recombination. Thus, there is a bias against inheriting bits from the same parents if they are at opposite ends of the encoding. However, with 2-point crossover, there is no longer any bias with regard to the end of the encodings. In 1-point crossover, the "end" of the encoding is a kind of explicit crossover point. 2-Point crossover treats the encoding as if it were circular so that the choice of both of the two crossover points are randomized.

Spears and De Jong (1991) argue for a variable *N*-point recombination. By controlling the number of recombination points, one can also control the likelihood of the two bits (or real-values) which are close together on one parent string and which are likely to be inherited together. With a small number of crossover points, recombination is less disruptive in terms of assorting which bits are inherited from which parents. With a large number of crossover points, recombination is more disruptive.

Is disruptive crossover good or bad? The early literature on genetic algorithms generally argued disruption was bad. On the other hand, disruption is also a source of new exploration. The most disruptive form of crossover is *uniform crossover*. Syswerda (1989) was one of the first application-oriented researchers to champion the use of uniform crossover. When uniform crossover is applied to a binary encoding, every bit is inherited randomly from one of the two parents. This means that uniform crossover on bit strings can actually be viewed as a blend of crossover and mutation. Any bits that are shared by the two parents must be passed along to

offspring, because no matter which parent the bit comes from, the bit is the same. But if the two parents differ in a particular bit position, then the bit that is inherited is randomly chosen. Another way of looking at uniform crossover is that all bit assignments that are shared by the two parents are automatically inherited. All bits that do not share the same value are set to a random value, because the inheritance is random. In some sense, the bits that are not set to the same value in the parents are determined by mutation: each of their values has an equal probability of being either one or zero.

Does it really matter if bits from a single parent that are close together on the encoding are inherited together or not? The early dogma of genetic algorithms held that inheritance of bits from a single parent that are close to each other on the chromosome was very important; very disruptive recombination, and uniform crossover in particular, should be bad. This in some sense was the central dogma at the time. This bias had a biological source.

This idea was borrowed in part from the biological concept of *coadapted alleles*. A widely held tenant of biological evolution is that distinct fragments of genetic information, or alleles, that work well *together* to enhance survivability of an organism should be inherited together. This view had a very strong influence on Holland's theory regarding the computational power of genetic algorithms: a theory based on the concept of hyperplane sampling.

1.4 Schemata and Hyperplanes

In his 1975 book, *Adaptation in Natural and Artificial Systems* (Holland 1975), Holland develops the concepts of schemata and hyperplane sampling to explain how a genetic algorithm can yield a robust search by implicitly sampling subpartitions of a search space. The idea that genetic algorithms derive their primary search power by hyperplane sampling is now controversial.

The set of strings of length L over a binary alphabet correspond to the vertices of an L-dimensional hypercube. A *hyperplane* is simply a subset of such vertices. These are defined in terms of the bit values they share in common. The concept of schemata is used to describe hyperplanes containing strings that contain particular shared bit patterns. Bits that are shared are represented in the schema; bits that are not necessarily shared are replaced by the * symbol. We will say that a bit string *matches* a particular schema if that bit string can be constructed from the schema by replacing a * symbol with the appropriate bit value. Thus, a 10-bit schema such as 1********* defines a subset that contains half the points in the search space, namely, all the strings that begin with a 1 bit in the search space. All bit strings that match a particular schema are contained in the hyperplane partition represented by that particular schema. The string of all * symbols corresponds to the space itself and is not counted as a partition of the space. There are $3^L - 1$ possible schemata since there are L positions in the bit string and each position can be a 0,1, or * symbol and we do not count the string with no zeros or ones. A *low order* hyperplane is represented by a schema that has few bits, but many * symbols. The number of points contained in a hyperplane is 2^k where k is the number of * symbols in the corresponding schema.

The notion of a population-based search is critical to the schema-based theory of the search power of 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. Holland introduced the concept of *intrinsic* or *implicit parallelism* to describe a situation where many hyperplanes are sampled when a population of strings is evaluated; it

has been argued that far more hyperplanes are sampled than the number of strings contained in the population.

Holland's theory suggests that schemata representing competing hyperplanes increase or decrease their representation in the population according to the relative fitness of the strings that lie in those hyperplane partitions. By doing this, more trials are allocated to regions of the search space that have been shown to contain above average solutions.

1.5 An Illustration of Hyperplane Sampling

Holland (1975) suggested the following view of hyperplane sampling. In *Fig. 6*, a function over a single variable is plotted as a one-dimensional space. The function is to be maximized. Assume the encoding uses 8 bits. 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 middle graph of *Fig. 6*, 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 bottom graph, $0^{*}10^{*****}$ is highlighted.

One of the assumptions behind the illustration in *Fig. 6* is that the sampling of hyperplane partitions is not affected to a significant degree by local minima. At the same time, increasing the sampling rate of regions of the search space (as represented by hyperplanes) that are above average compared to other competing regions does not guarantee convergence to a global optimum. The global optimum could be a relatively isolated peak that might never be sampled, for example. A small randomly placed peak that is not sampled is basically invisible; all search algorithms are equally blind to such peaks if there is no information to guide search.

Nevertheless, good solutions that are globally competitive might be found by such a strategy. This is particularly true if the search space is structured in such a way that pieces of good solutions can be recombined to find better solutions. The notion that hyperplane sampling is a useful way to guide search should be viewed as heuristic. In general, even having perfect knowledge of schema averages up to some fixed order provides little guarantee as to the quality of the resulting search. This is discussed in detail in Sect. 2.

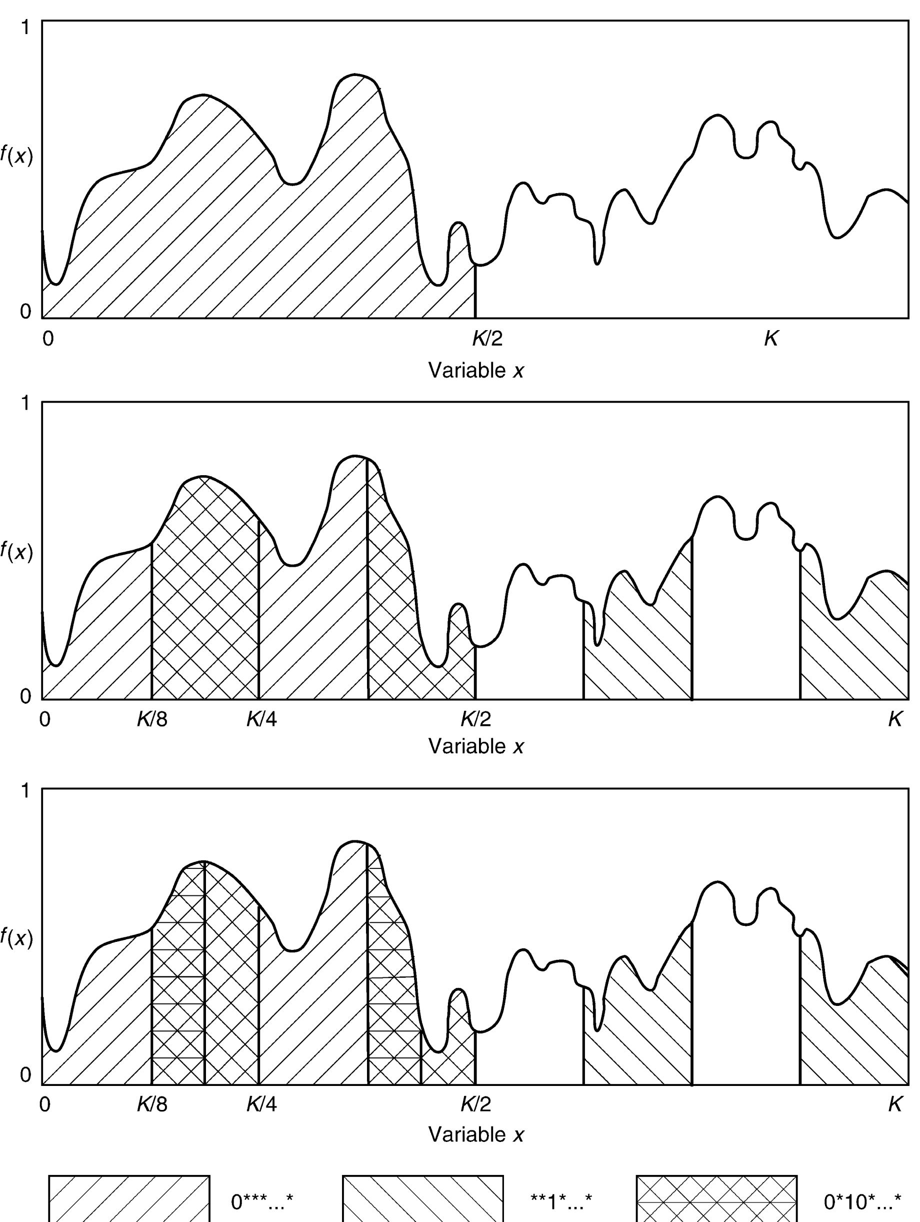
1.6 The Schema Theorem

Holland (1975) developed the *schema theorem* to provide a lower bound on the change in the sampling rate for a single hyperplane from generation t to generation $t+1$. By developing the theorem as a lower bound, Holland was able to make the schema theorem hold independently for every schema/hyperplane. At the same time, as a lower bound, the schema theorem is inexact, and the bounds hold for only one generation into the future. After one generation, the bounds are no longer guaranteed to hold. This weakness is just one of the many reasons that the concept of "hyperplane sampling" is controversial.

Let $P(H,t)$ be the proportion of the population that samples a hyperplane H at time t. Let $P(H,t + \text{intermediate})$ be the proportion of the population that samples hyperplane H after fitness proportionate selection but before crossover or mutation. Let $f(H,t)$ be the average fitness of the strings sampling hyperplane H at time t and denote the population average by $\bar{f}$. Note that $\bar{f}$ should also have a time index, but this is often not denoted explicitly. This is

Fig. 6

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



important because the average fitness of the population is *not* constant. Assuming that selection is carried out using fitness proportional selection:

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

Thus, ignoring crossover and mutation, under just selection, the sampling rate of hyperplanes changes according to their average fitness. Put another way, selection "focuses" the search in what appears to be promising regions where the strings sampled so far have above-average fitness compared to the remainder of the search space. Some of the controversy related to "hyperplane sampling" begins immediately with this characterization of selection. The equation accurately describes the focusing effects of selection; the concern, however, is that the focusing effect of selection is not limited to the $3^L - 1$ hyperplanes that Holland considered to be relevant. Selection acts exactly the same way on any arbitrarily chosen subset of the search space. Thus, it acts in exactly the same way on the $2^{(2^L)}$ members of the power set over the set of all strings. While there appears to be nothing special about the sampling rate of hyperplanes under selection, all subsets of strings are not acted on in the same way by crossover and mutation. Some subsets of bit patterns corresponding to schemata are more likely to survive and be inherited in the population under crossover and mutation. For example, the sampling rate of order 1 hyperplanes is not disrupted by crossover and, in general, lower order hyperplanes are less affected by crossover than higher order hyperplanes.

Laying this issue aside for a moment, it is possible to write an exact version of the schema theorem that considers selection, crossover, and mutation. What we want to compute is $P(H, t+1)$, the proportion of the population that samples hyperplane H at the next generation as indexed by $t+1$. First just consider selection and crossover.

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

where p_c is the probability of performing a crossover operation. When crossover does not occur (which happens with probability $(1-p_c)$), only selection changes the sampling rate. However, when crossover does occur (with probability p_c) then we have to consider how crossover and mutation can destroy hyperplane samples (denoted by losses) and how crossover can create new samples of hyperplanes (denoted by gains).

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 source of *gains* that is referred to in the above calculation.

We will return to an exact computation, but, for now, instead of computing *losses* and *gains*, what if we compute a bound on them instead? 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, we now have a bound on the sampling rate of schemata rather than an exact characterization:

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

The *defining length* of a schema 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 *, scanning left to right, if I_x is the index of the position of the rightmost occurrence of either a 0 or a 1 and I_y is the index of the leftmost occurrence of either a 0 or a 1, then the defining length is merely $I_x - I_y$. The defining length of a schema representing a hyperplane H

is denoted here by $\Delta(H)$. If 1-point is used, then the defining length can be used to also calculate a bound on disruption:

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

and including this term (and applying simple algebra) yields:

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

We now have a useful version of the schema theorem (although it does not yet consider mutation). This version assumes that selection for the first parent string is fitness-based and the second parent is chosen randomly. But typically both parents are chosen based on fitness. This can be added to the schema theorem by merely indicating the alternative parent chosen from the intermediate population after selection (Schaffer 1987).

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

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 not affect the schema representing H is $(1-p_{\mathrm{m}})^{o(H)}$. This leads to the following expression of the schema theorem.

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

2 Interpretations and Criticisms of the Schema Theorem

For many years, the schema theorem was central to the theory of how genetic algorithms are able to effectively find good solutions in complex search spaces. Groups of bits that are close together on the encoding are less likely to be disrupted by crossover. Therefore, if groups of bits that are close together define a (hyperplane) subregion of the subspace that contains good solutions, selection should increase the representation of these bits in the population, and crossover and mutation should not "interfere" with selection since the probability of disruption should be low. In effect, such groups of bits act as coadapted sets of alleles; these are so important that they have been termed the "building blocks" of genetic search. As different complexes of coadaptive alleles (or bits) emerge in a population, these building blocks are put together by recombination to create even better individuals. The most aggressive interpretation of the schema theorem is that a genetic algorithm would allocate nearly optimal trials to sample different partitions of the search space in order to achieve a near-optimal global search strategy.

There are many different criticisms of the schema theorem. The schema theorem is not incorrect, but arguments have been made that go beyond what is actually proven by the schema theorem. First of all, the schema theorem is an inequality, and it only applies to one generation into the future. So while the bound provided by the schema theorem absolutely holds for one

generation into the future, it provides no guarantees about how strings or hyperplanes will be sampled in future generations.

It is true that the schema theorem does hold true independently for all possible hyperplanes for one generation. However, over multiple generations, the interactions between different subpartitions of the search space (as represented by hyperplanes and schemata) are extremely important. For example, in some search space of size 2^8 suppose that the schemata 11****** and *00***** are both "above average" in the current generation of a population in some run of a genetic algorithm. Assume the schema theorem indicates that both will have increasing representation in the next generation. But trials allocated to schemata 11****** and *00***** are in conflict because they disagree about the value of the second bit. Over multiple generations, both regions cannot receive increasing trials. These schema are *inconsistent* about what bit value is to be preferred in the second position. The schema theorem does not predict how such inconsistencies will be sorted out.

Whitley et al. (1995b) and Heckendorn et al. (1996) have shown that problems can have varying degrees of consistency in terms of which hyperplanes appear to be promising. For problems that display higher *consistency*, the "most fit" schemata tend to agree about what the values of particular bits should be. In a problem where there is a great deal of consistency, genetic search is usually effective. But other problems can be highly inconsistent, so that the most fit individuals (and sets of individuals as represented by schemata) display a large degree of conflict in terms of what bit values are preferred in different positions. It seems reasonable to assume that a genetic algorithm should do better on problems that display greater consistency, since inconsistency means that the search is being guided by conflicting information (this is very much related to the notion of "deception" but the concept of deception is controversial and much misunderstood).

One criticism of pragmatic significance is that users of the standard or canonical genetic algorithm often use very small populations. The number of positions containing 0 or 1 is referred to as the order of a schema. Thus, **1***** is an order 1 schema, ***0***1 is an order 2 schema, and *1**0*1* is an order 3 schema. Many users employ a population size of 100 or smaller. In a population of size 100, we would expect 50 samples of any order 1 schema, 25 samples of any order 2 schema, 12.5 samples of any order 3 schema, and exponentially decaying numbers of samples to higher order schema. Thus, if we suppose that the genetic algorithm is implicitly attempting to allocate trials to different regions of the search space based on schema averages, a small population (e.g., 100) is inadequate unless we only care about very low order schemata. Therefore, even if hyperplane sampling is a robust form of heuristic search, the user destroys this potential by using small population sizes. Small populations require that the search rely more on hill-climbing. But in some cases, hill-climbing is very productive.

What if we had perfect schema information? What if we could compute schema information exactly in polynomial time? Rana et al. (1998) have shown that schema information up to any fixed order can be computed in polynomial time for some nondeterministic polynomial time (NP)-Complete problems. This includes Maximum Satisfiability (MAXSAT) problems and NK-Landscapes. This is very surprising. One theoretical consequence of this is captured by the following theorem:

- If $P \neq NP$ then, in the general case, exact knowledge of the static schema fitness averages up to some fixed order cannot provide information that can be used to guarantee finding a global optimum, or even an above average solution, in polynomial time. (For proofs, see Heckendorn et al. 1999a,b).

This seems like a very negative result. But it is dangerous to overinterpret either positive or negative results. In practice, random MAXSAT problems are characterized by highly inconsistent schema information, so there is really little or no information that can be exploited to guide the search (Heckendorn et al. 1999b). And in practice, genetic algorithms perform very poorly on MAXSAT problems (Rana et al. 1998) unless they are aided by additional hill-climbing techniques. On the other hand, genetic algorithms are known to work well in many other domains. Again, the notion of using schema information to guide search is heuristic.

There are many other criticisms of the schema theorem. In the early literature, too much was claimed about schema and hyperplane processing that was not backed up by solid proofs. It is no longer accepted that genetic algorithms allocate trials in an "optimal way" and it is certainly not the case that the genetic algorithm is guaranteed to yield optimal or even near-optimal solutions. In fact, there are good counterexamples to these claims. On the other hand, some researchers have attacked the entire notion of schema processing as invalid or false. Yet, the schema theorem itself is clearly a valid bound; and, experimentally, in problems where there are clearly defined regions that are above average, the genetic algorithm does quickly allocate more trials to such regions as long as these regions are relatively large.

There is still a great deal of work to be done to understand the role that hyperplane sampling plays in genetic search. Historically, the role of hyperplane sampling has been exaggerated, and the role played by hill-climbing has been underestimated. At the same time, many of the empirical results that call into question how genetic algorithms really work have been carried out on a highly biased sample of test problems. These test problems tend to be *separable*. A separable optimization problem is one in which the parameters are independent in terms of interaction. These problems are inherently easy to solve. For example, using a fixed precision (e.g., 32 bits per parameter), a separable problem can be solved exactly in a time that is a polynomial in the number of parameters by searching each parameter separately. On such problems, hill-climbing is a highly effective search strategy. Perhaps because of these simple test problems, the effectiveness (and speed) of simple hill-climbing has been overestimated.

In the next section, exact models of Holland's simple genetic algorithm will be introduced.

3 Infinite Population Models of Simple Genetic Algorithms

Goldberg (1987, 1989b) and Bridges and Goldberg (1987) were the first to model critical details of how the genetic algorithm processes infinitely large populations under recombination. These models were independently derived in 1990 by Vose and Whitley in a more precise and generalized form. Vose and Liepins (1991) and Vose (1993) extended and generalized this model while Whitley et al. (1992) and Whitley (1993) introduced another version of the infinite population model that connects the work of Goldberg and Vose. In this section, the Vose model is reviewed and it is shown how the effects of various operators fit into this model.

The models presented here all use fitness proportionate reproduction because it is simpler to model mathematically. Vose (1999) also presents models for rank-based selection.

The vector $p^t \in \mathbb{R}$ is such that the kth component of the vector is equal to the proportional representation of string k at generation t. It is assumed that $n = 2^L$ is the number of points

in the search space defined over strings of length L and that the vector p is indexed 0 to $n-1$. The vector $s^t \in \mathbb{R}$ represents the tth generation of the genetic algorithm after selection and the ith component of s^t is the proportional representation of string i in the population after selection, but before any operators (e.g., recombination, mutation, and local search) are applied. Likewise, p_i^t represents the proportional representation of string i at generation t *before* selection occurs.

The function $r_{i,j}(k)$ yields the probability that string k results from the recombination of strings i and j. (For now, assume that r only yields the results for recombination; the effect of other operators could also be included in r.) Now, using $\mathcal{E}$ to denote expectation,

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

To begin the construction of a general model, we first consider how to calculate the proportional representation of string 0 (i.e., the string composed of all zeros) at generation $t+1$; in other words, we compute p_0^{t+1}. A mixing matrix M is constructed where the (i,j)th entry $m_{i,j} = r_{i,j}(0)$. Here M is built by assuming that each recombination generates a single offspring. The calculation of the change in representation for string $k = 0$ is now given by

$$\mathcal{E}\{p_0^{t+1}\} = \sum_{i,j} s_i^t s_j^t r_{i,j}(0) = s^T M s \tag{2}$$

where T denotes transpose. Note that this computation gives the expected representation of a single string, 0, in the next genetic population.

It is simple to see that the exact model for the infinite population genetic algorithm has the same structure as the model on which the schema theorem is based. For example, if we accept that the string of all zeros (denoted here simply by the integer 0) is a special case of a hyperplane, then when $H = 0$, the following equivalence holds:

$$\begin{aligned} s^T M s = P(H, t+1) &= (1 - p_c) P(H, t) \frac{f(H, t)}{\bar{f}} \\ &\quad + p_c \left[P(H, t) \frac{f(H, t)}{\bar{f}} (1 - \text{losses}) + \text{gains} \right] \end{aligned} \tag{3}$$

The information about losses and gains is contained in the matrix M. To keep matters simple, assume the probability of crossover is $p_c = 1$. Then, in the first row and column of matrix M, the calculation is really the probability of *retaining* a copy of the string, which is $(1 - \text{losses})$ (except at the intersection of the first row and column); the probabilities elsewhere in M are the gains in the above equation. To include the probability of crossover p_c in the model one must include the crossover probability in the construction of the M matrix.

The point of ➲ Eq. 3 is that the exact Vose/Liepins model and the model on which the schema theorem is based is really the same. Whitley (1993) presents the calculations of losses and gains so as to make the congruent aspects of the infinite population model and the schema theorem more obvious.

The equations that have been looked at so far tell us how to compute the expected representation of one point in the search space one generation into the future. The key is to generalize this calculation to all points in the search space. Vose and Liepins formalized the

notion that bitwise exclusive-or can be used to access various probabilities from the recombination function r. Specifically,

$$r_{i,j}(k) = r_{i,j}(k \oplus 0) = r_{i\oplus k, j\oplus k}(0). \tag{4}$$

This implies that the mixing matrix M, which was defined such that entry $m_{i,j} = r_{i,j}(0)$, can provide mixing information for any string k just by changing how M is accessed. By reorganizing the components of the vector, s, the mixing matrix M can yield information about the probability $r_{i,j}(k)$. A permutation function, σ, is defined as follows:

$$\sigma_j \langle s_0, \ldots, s_{n-1} \rangle^T = \langle s_{j\oplus 0}, \ldots, s_{j\oplus(n-1)} \rangle^T \tag{5}$$

where the vectors are treated as columns and n is the size of the search space. The computation

$$(\sigma_q \; s^t)^T M (\sigma_q \; s^t) = p_q^{t+1} \tag{6}$$

thus reorganizes s with respect to string q and produces the expected representation of string q at generation $t+1$. A general operator $\mathcal{M}$ can now be defined over s, which remaps $s^T M s$ to cover all strings in the search space.

$$\mathcal{M}(s) = \langle (\sigma_0 \; s)^T M (\sigma_0 \; s), \ldots, (\sigma_{n-1} \; s)^T M (\sigma_{n-1} \; s) \rangle^T \tag{7}$$

This model has not yet addressed how to generate the vector s^t given p^t. 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 fitness function. Following Vose and Wright (1997),

$$s^t = Fp^t / 1^T Fp^t \tag{8}$$

since $Fp^t = \langle f_0 p_0^t, f_1 p_1^t, \ldots, f_{n-1}, p_{n-1}^t \rangle$ and the population average is given by $1^T Fp^t$.

Vose (1999) refers to this complete model as the $\mathcal{G}$ function. Given any population distribution p, $\mathcal{G}(p)$ can be interpreted in two ways. On one hand, if the population is infinitely large, then $\mathcal{G}(p)$ is the exact distribution of the next population. On the other hand, given any *finite* population p, if strings in the next population are chosen one at a time, then $\mathcal{G}(p)$ also defines a vector such that element i is chosen for the next generation with probability $\mathcal{G}(p)_i$. This is because $\mathcal{G}(p)$ defines the exact sampling distribution for the next generation. This is very useful when constructing finite Markov models of the simple genetic algorithm.

Next, we look at how mutation can be added to this model in a modular fashion. This could be built directly into the construction of the M matrix. But looking at mutation as an additional process is instructive.

Recall that M is the mixing matrix, which we initially defined to cover only crossover. Define $\mathcal{Q}$ as the mutation matrix. Assuming mutation is independently applied to each bit with the same probability, $\mathcal{Q}$ can be constructed by defining a mutation vector Γ such that component Γ_i is the probability of mutating string i and producing string 0. The vector Γ is the first column of the mutation matrix and in general Γ can be reordered to yield column j of the matrix by reordering such that element $q_{i,j} = \Gamma_{i\oplus j}$.

Having defined a mutation matrix, mutation can now be applied after recombination in the following fashion:

$$p^{t+1,m} = (p^{t+1})^T \mathcal{Q},$$

where $p^{t+1,m}$ is just the p vector at time $t+1$ after mutation has occurred. Mutation also can be done before crossover; the effect of mutation on the vector s immediately after selection produces the following change: $s^T \mathcal{Q}$, or equivalently, $\mathcal{Q}^T s$.

Now we drop the $p^{t+1,m}$ notation and assume that the original p^{t+1} vector is defined to include mutation, such that

$$
\begin{aligned}
{p_0}^{t+1} &= (\mathcal{Q}^T s)^T M (\mathcal{Q}^T s) \\
{p_0}^{t+1} &= s^T (\mathcal{Q}\mathcal{M}\mathcal{Q}^T) s
\end{aligned}
$$

and we can therefore define a new matrix M_2 such that

$$ {p_0}^{t+1} = s^T M_2 s \quad \text{where} \quad M_2 = (\mathcal{Q}\mathcal{M}\mathcal{Q}^T) $$

As long as the mutation rate is independently applied to each bit in the string, it makes no difference whether mutation is applied before or after recombination. Also, this view of mutation makes it clear how the general mixing matrix can be built by combining matrices for mutation and crossover.

4 The Markov Model for Finite Populations

Nix and Vose (1992) show how to structure the finite population for a simple genetic algorithm. Briefly, the Markov model is an $N \times N$ transition matrix Q, where N is the number of finite populations of K strings and $Q_{i,j}$ is the probability that the kth generation will be population $\mathcal{P}_j$, given that the $(k-1)$th population is $\mathcal{P}_i$.

Let

$$ \langle Z_{0,j}, Z_{1,j}, Z_{2,j}, \ldots, Z_{r-1,j} \rangle $$

represent a population, where $Z_{x,j}$ represents the number of copies of string x in population j, and $r=2^L$. The population is built incrementally. The number of ways to place the $Z_{0,j}$ copies of string 0 in the population is:

$$ \binom{K}{Z_{0,j}} $$

The number of ways $Z_{1,j}$ strings can be placed in the population is:

$$ \binom{K - Z_{0,j}}{Z_{1,j}} $$

Continuing for all strings,

$$ \binom{K}{Z_{0,j}} \binom{K - Z_{0,j}}{Z_{1,j}} \binom{K - Z_{0,j} - Z_{1,j}}{Z_{2,j}} \cdots \binom{K - Z_{0,j} - Z_{1,j} - \cdots - Z_{r-2,j}}{Z_{r-1,j}} $$

which yields

$$ \frac{K!}{(K - Z_{0,j})! Z_{0,j}!} \frac{(K - Z_{0,j})!}{(K - Z_{0,j} - Z_{1,j})! Z_{1,j}!} \frac{(K - Z_{0,j} - Z_{1,j})!}{(K - Z_{0,j} - Z_{1,j} - Z_{2,j})! Z_{2,j}!} \cdots \frac{(K - Z_{0,j} - Z_{1,j} - \cdots - Z_{r-2,j})!}{Z_{r-1,j}!} $$

which in turn reduces to

$$ \frac{K!}{Z_{0,j}! Z_{1,j}! Z_{2,j}! \ldots Z_{r-1,j}!} $$

Let $C_i(y)$ be the probability of generating string y from the finite population P_i. Then

$$Q_{i,j} = \frac{K!}{Z_{0,j}!Z_{1,j}!Z_{2,j}! \ldots Z_{r-1,j}!} \prod_{y=0}^{r-1} C_i(y)^{Z_{y,j}}$$

and

$$Q_{i,j} = K! \prod_{y=0}^{r-1} \frac{C_i(y)^{Z_{y,j}}}{Z_{y,j}!}$$

So, how do we compute $C_i(y)$? Note that the finite population P_i can be described by a vector p. Also note that the sampling distribution from which P_j is constructed is given by the infinite population model $\mathcal{G}(p)$ (Vose 1999). Thus, replacing $C_i(y)$ by $\mathcal{G}(p)_y$ yields

$$Q_{i,j} = K! \prod_{y=0}^{r-1} \frac{(\mathcal{G}(p)_y)^{Z_{y,j}}}{Z_{y,j}!}$$

5 Theory Versus Practice

As the use of evolutionary algorithms became more widespread, a number of alternative genetic algorithm implementations have also come into common use. Some of the evolutionary algorithms in common use are based on evolution strategies. Select algorithms closest to traditional genetic algorithms will be reviewed.

The widespread use of alternative forms of genetic algorithms also means there is a fundamental tension between (at least some part of) the theory community and the application community. There are beautiful mathematical models and some quite interesting results for Holland's original genetic algorithm. But there are few results for the alternative forms of genetic algorithms that are used by many practitioners.

5.1 Steady-State and Island Model Genetic Algorithms

Genitor (Whitley and Kauth 1988; Whitley 1989) was the first of what was later termed "steady-state" genetic algorithms by Syswerda (1989). The name "steady-state" is somewhat unfortunate and the term "monotonic" genetic algorithm has been suggested by Alden Wright: these algorithms keep the best solutions found so far, and thus the population average monotonically improves over time. The distinction between steady-state genetic algorithms and regular generational genetic algorithms was also foreshadowed by the evolution strategy community. The Genitor algorithm, for example, can also be seen as a variant of a $(\mu+1)$-Evolution strategy in terms of its selection mechanism. In contrast, Holland's generational genetic algorithm is an example of a (μ,λ)-Evolution Strategy where $\mu = \lambda$. The genetic algorithms community also used "monotonic" selection for classifier systems in the early 1980s.

Reproduction occurs one individual at a time in the Genitor algorithm. Two parents are selected for reproduction and produce an offspring that is immediately placed back into the population. The worst individual in the population is deleted. Ignoring the worst member of the population, the remainder of the population monotonically improves.

Another major difference between Genitor and other forms of genetic algorithms is that fitness is assigned according to rank rather than by fitness proportionate reproduction. In the original Genitor algorithm, the population is maintained in a sorted data structure. Fitness is assigned according to the position of the individual in the sorted population. This also allows one to prevent duplicates from being introduced into the population. This selection schema also means that the best $N-1$ solutions are always preserved in a population of size N. Goldberg and Deb (1991) have shown that by replacing the worst member of the population, Genitor generates much higher selective pressure than the canonical genetic algorithm.

Steady-state genetic algorithms retain the flavor of a traditional genetic algorithm. But the differences are significant. Besides the additional selective pressure, keeping the best strings seen so far means that the resulting search is more focused. This can be good or bad. The resulting search has a strong hill-climbing flavor, and the algorithm will continue to perturb the (best) strings in the population by crossover and mutation looking for an improved solution. If a population contains adequate building blocks to reach a solution in reasonable time, this focus can pay off. However, if the population does not contain adequate building blocks to generate further improvements, the search will be stuck. A traditional genetic algorithm or a (μ,λ)-evolution strategy allows a certain degree of drift and has the ability to continue moving in the space of possible populations. Thus, the greedy focus of a steady-state genetic algorithm sometimes pays off, and sometimes it does not.

To combat stagnation, it is sometimes necessary to use larger populations with steady-state genetic algorithms, or to use more aggressive mutation. Generally, larger populations result in slower progress, but improved solutions in the long run. Another way to combat stagnation is to use an *island model genetic algorithm*. Steady-state genetic algorithms seem to work better in conjunction with the island model paradigm than generational genetic algorithms. This may be due to the fact that steady-state genetic algorithms are more prone to stagnation.

Instead of using one large population, the population can be broken into several smaller populations. Thus, instead of running one population of size 1,000, one might run five populations of size 200. While the smaller population will tend to converge faster, diversity can be maintained by allowing migration between the subpopulations. For example, a small number of individuals (e.g., 1–5) might migrate from one subpopulation to another every 5–10 generations. It is important that migration be limited in size and frequency, otherwise the set of subpopulation becomes homogeneous too quickly (Starkweather et al. 1990).

In practice, steady-state genetic algorithms such as Genitor are often better optimizers than the canonical generational genetic algorithm. This has especially been true for scheduling problems such as the traveling salesman problem.

Current implementations of steady-state genetic algorithms are likely to use tournament selection instead of explicit rank-based selection. The use of tournament selection makes it unnecessary to keep the population in sorted order for the purposes of selection. But the population must still be sorted to keep track of the worst member of the population over time. Of course, once the population is sorted, inserting a new offspring takes $O(N)$ time, where N is the population size. In practice, poor offspring that are not inserted incur no insertion cost, and by starting from the bottom of the population, the insertion cost is proportional to the fitness of the offspring.

Some researchers have experimented with other ways of deciding which individuals should be replaced after new offspring are generated. Tournament selection can be used in reverse to decide if a new offspring should be allowed into the population and which member of the

current population should be replaced. When tournament selection is used in reverse, the tournament sizes are typically larger. One can also use an aspiration level, so that new offspring are not allowed to compete for entry into the population unless this aspiration level is met.

5.2 CHC

The CHC (Cross generational elitist selection, Heterogeneous recombination and Cataclysmic mutation) (Eshelman 1991; Eshelman and Schaffer 1991) algorithm was created by Larry Eshelman with the explicit idea of borrowing from both the genetic algorithm and the evolution strategy community. CHC explicitly borrows 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. This also implies that duplicates are removed from the population. This form of selection is referred to as *truncation selection*. From the genetic algorithm community, CHC builds on the idea that recombination should be the dominant search operator. A bit representation is typically used for parameter optimization problems. In fact, CHC goes so far as to use *only* recombination in the main search algorithm. However, it restarts the search where progress is no longer being made by employing what Eshelman refers to as *cataclysmic mutation*.

Since truncation selection is used, parents can be paired randomly for recombination. However, the CHC algorithm also employs a *heterogeneous recombination* restriction as a method of "incest prevention" (Eshelman 1991). This is accomplished by only mating those string pairs which differ from each other by some number of bits; Eshelman refers to this as a mating threshold. The initial mating threshold is set at $L/4$, where L is the length of the string. If a generation occurs, in which no offspring are inserted into the new population, then the threshold is reduced by 1.

The crossover operator in CHC performs uniform crossover; bits are randomly and independently exchanged, but exactly half of the bits that differ are swapped. This operator, called HUX (Half Uniform Crossover) ensures that offspring are equidistant between the two parents. This serves as a diversity preserving mechanism. An offspring that is *closer* to a parent in terms of Hamming distance will have a tendency to be more similar to that parent in terms of evaluation. Even if this tendency is weak, it can contribute to loss of diversity. If the offspring and parent that are closest in Hamming space tend to be selected together in the next generation, diversity is reduced. By requiring that the offspring be exactly halfway in between the two parents in terms of Hamming distance, the crossover operator attempts to slow down loss of genetic diversity. One could also argue that the HUX operator attempts to maximize the distribution of new samples in new regions of the search space by placing the offspring as far as possible from the two parents while still retaining the bits which the two parents share in common.

No mutation is applied during the regular search phase of the CHC algorithm. When no offspring can be inserted into the population of a succeeding generation and the mating threshold has reached a value of 0, CHC infuses new diversity into the population via a form of restart. Cataclysmic mutation uses the best individual in the population as a template to re-initialize the population. The new population includes one copy of the template string; the remainder of the population is generated by mutating some percentage of bits (e.g., 35%) in the template string.

Bringing this all together, 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.

The rationale behind CHC is to have a very aggressive search by using truncation selection which guarantees the survival of the best strings, but to offset the aggressiveness of the search by using highly disruptive uniform crossover. Because of these mechanisms, CHC is able to use a relatively small population size. It generally works well with a population size of 50. Eshelman and Schaffer have reported quite good results using CHC on a wide variety of test problems (Eshelman 1991; Eshelman and Schaffer 1991). Other experiments (c.f. Mathias and Whitley 1994; Whitley et al. 1995) have shown that it is one of the most effective evolutionary algorithms for parameter optimization on many common test problems. Given the small population size and the operators, it seems unreasonable to think of an algorithm such as CHC as a "hyperplane sampling" genetic algorithm. Rather, it can be viewed as an aggressive population-based hill-climber that also uses restarts to quickly explore new regions of the search space. This is also why the small population size is important to the performance of CHC. If the population size is increased, there is some additional potential for increased performance before a restart is triggered. But as with steady-state genetic algorithms, an increase in population size generally results in a small increase in performance after a significant increase in time to convergence/stagnation. The superior performance of CHC using a population of 50 suggests that more restarts have a better payoff than increasing the population size.

6 Where Genetic Algorithms Fail

Do genetic algorithms have a particular mode of failure? There are probably various modes of failure, some of which are common to many search algorithms. If functions are too random, too noisy, or fundamentally unstructured, then search is inherently difficult. "Deception" in the form of misleading hyperplane samples is also a mode of failure that has been studied (Whitley 1991; Goldberg 1989a; Grefenstette 1993).

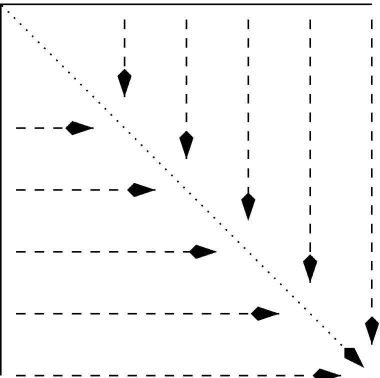
There is another fundamental mode of failure such that there exists a well-defined set of problems where the performance of genetic algorithms is likely to be poor. Experiments show that various genetic algorithms and local search methods are more or less blind to "ridges" in the search space of parameter optimization problems. In two dimensions, the *ridge problem* is essentially this: a method that searches parallel to the x and y axes cannot detect improving moves that are oriented at a 45° angle to these axes.

A simplified representation of a ridge problem appears in ➲ *Fig. 7*. Changing one variable at a time will move local search to the diagonal. However, looking in either the x-dimension or the y-dimension, every point along the diagonal appears to be a local optimum. There is actually gradient information if one looks *along* the diagonal; however, this requires either (1) changing both variables at once or (2) transforming the coordinate system of the search space so as to "expose" the gradient information.

The ridge problem is relatively well documented in the mathematical literature on derivative free minimization algorithms (Rosenbrock 1960; Brent 1973). However, until recently, there has been little discussion of this problem in the evolutionary algorithm literature.

Let $\Omega = \{0, 1, \ldots, 2^\ell - 1\}$ be the search space which can be mapped onto a hypercube. Elements $x, z \in \Omega$ are *neighbors* when (x, z) is an edge in the hypercube. Bit climbing search algorithms terminate at a *local optimum*, denoted by $x \in \Omega$, such that none of the points in the neighborhood $N(x)$ improve upon x when evaluated by some objective function. Of course, the neighborhood structure of a problem depends upon the coding scheme used. Gray codes are often used for bit representations because, by definition, adjacent integers are adjacent neighbors.

Fig. 7
Local search moves only in the horizontal and vertical directions. It therefore "finds" the diagonal, but gets stuck there. Every point on the diagonal is locally optimal. Local search is blind to the fact that there is gradient information moving along the diagonal.



Suppose the objective function is defined on the unit interval $0 \leq x < 1$. To optimize this function, the interval is discretized by selecting n points. The *natural encoding* is then a map from Ω to the graph that has edges between points x and $x+1$ for all $x = 0, 1, \ldots, n-2$.

Under a Gray encoding, adjacent integers have bit representations that are neighbors at Hamming distance 1 (e.g., 3 = 010, 4 = 110). Thus a Gray encoding has the following nice property:

- A function $f : \Omega \to \mathbb{R}$ cannot have more local optima under a Gray encoding than it does under the natural encoding.

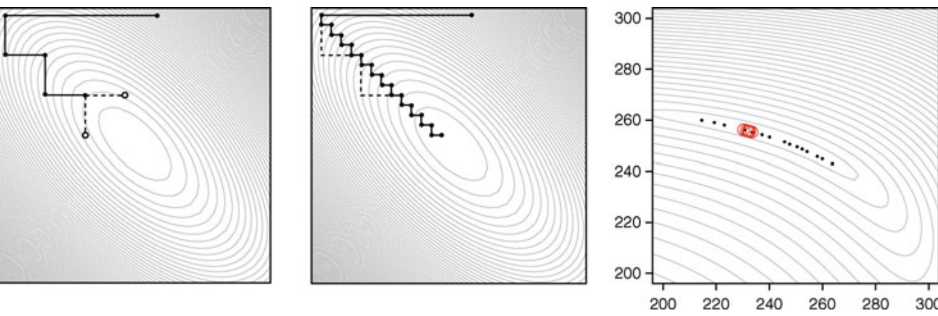
A proof first appears in Rana and Whitley (1997); this theorem states that when using a Gray code, local optima of the objective function considered as a function on the unit interval can be destroyed, but no new local optima can be created (*Fig. 8*).

But are there unimodal functions where the natural encoding is multimodal? If the function is one dimensional, the answer is no. However, if the function is not one dimensional, the answer is yes. False local optima are induced on ridges since there are points along the ridge where improving moves become invisible to search.

This limitation is not unique to local search, and it is not absolute for genetic algorithms. Early population sampling can potentially allow the search to avoid being trapped by "ridges." It is also well known that genetic algorithms quickly lose diversity and then the search must use mutation or otherwise random jumps to move along the ridge. Any method that tends to search one dimension at a time (or to find improvements by changing one dimension at a time via mutation) has the same limitation, including local search and simple "line search" methods.

Salomon (1960) showed that most benchmarks become much more difficult when the problems are *rotated*. Searching a simple two-dimensional elliptical bowl is optimally solved by one iteration of line search when the ellipse is oriented with the x and y axis. However, when the space is rotated 45°, the bowl becomes a ridge and the search problem is more difficult for many search algorithms.

Fig. 8
The two leftmost images show how local search moves on a 2D ridge using different step sizes. Smaller step sizes results in more progress on the ridge, but more steps to get there. The rightmost figure shows points at which local search becomes stuck on a real-world application.



Modern evolution strategies are invariant under rotation. In particular, the covariance matrix adaptation (CMA) evolution strategy uses a form of principal component analysis (PCA) to rotate the search space as additional information is obtained about the local landscape (Hansen 2006). Efforts are currently underway to generalize the concept of a rotationally invariant representation that could be used by different search algorithms (Hansen 2008). Until this work advances, current versions of genetic algorithms still have problems with some ridge structures. Crossover operators such as interval crossover (Schaffer and Eshelman 1993) attempt to deal with the ridge problem but are not as powerful as CMA.

7 An Example of Genetic Algorithms for Resource Scheduling

Resource scheduling problems involve the allocation of time and/or a resource to a finite set of requests. When demand for a resource becomes greater than the supply, conflicting requests require some form of arbitration and the scheduling problem can be posed as an optimization problem over a combinatorial domain. Thus, these problems are combinatorial in nature and are quite distinct from parameter optimization problems. Indeed, genetic algorithms have been very successful on scheduling applications of this nature.

A schedule may attempt to maximize the total number of requests that are filled, or the aggregate value (or priority) of requests filled, or to optimize some other metric of resource utilization. In some cases, a request needs to be assigned a time window on some appropriate resource with suitable capabilities and capacity. In other cases, the request does not correspond to a particular time window, but rather just a quantity of an oversubscribed resource.

One example of resource scheduling is the scheduling of flight simulators (Syswerda 1991). Assume a company has three flight simulators, and 100 people who want to use them. In this case, the simulators are a limited resource. A user might request to use a simulator from 9:00 a.m. to 10:00 a.m. on Tuesday. But if six users want a simulator at this time, then not all requests can be fulfilled. Some users may only be able to utilize the simulators at certain times or on certain days. How does one satisfy as many requests as possible? Or which requests are most important to satisfy? Does it make sense to schedule a user for less time than they requested?

One way to attempt to generate an approximate solution to this problem is to have users indicate a number of prioritized choices. If a user's first choice cannot be filled, then perhaps their second choice of times is available. A hypothetical evaluation function is to have a schedule evaluator that awards ten points for every user that gets their first choice, and five points for those that get their second choice and three points for those that get their third choice, one point for those scheduled (but not in their first, second, or third prioritized requests) and -10 points for user requests that cannot be scheduled.

Is this the best evaluation function for this problem? This raises an interesting issue. In real-world applications, sometimes the evaluation function is not strictly determined and developers must work with users to define a reasonable evaluation function. Sometimes what appears to be an "obvious" evaluation function turns out to be the wrong evaluation function.

Reasonably good solutions to a resource scheduling problem can usually be obtained by using a greedy scheduler which allocates to each request the best available resource at the best available time on a first-come, first-served basis.

The problem with a simple greedy strategy is that requests are not independent – when one request is assigned a slot on a resource, that resource is no longer available (or less available). Thus, placing a single request at its optimal position may preclude the optimal placement of multiple other requests. This is the standard problem with all greedy methods.

A significant improvement on a greedy first-come, first-served strategy is to explore the space of possible permutations of the requests where the permutation defines a priority ordering of the requests to be placed in the schedule. A genetic algorithm can then be applied to search the space of permutations. The fitness function is some measure of cost or quality of the resulting schedule.

The use of a permutation-based representation also requires the construction of a "schedule builder" which maps the permutation of requests to an actual schedule. In a sense, the schedule builder is also a greedy scheduling algorithm. Greedy scheduling on its own can be a relatively good strategy. However, by exploring the space of different permutations, one changes the order in which requests get access to resources; this can also be thought of as changing the order in which requests arrive. Thus, exploring the space of permutations provides the opportunity to improve on the simple greedy scheduler.

The separation of "permutation space" and "schedule space" allows for the use of two levels of optimization. At a lower level, a greedy scheduler converts each permutation into a schedule; at a higher level, a genetic algorithm is used to search the space of permutations. This approach thus creates a strong separation between the problem representation and the actual details of the particular scheduling application. This allows the use of relatively generic "genetic recombination" operators or other local search operators. A more direct representation of the scheduling problem would require search operators that are customized to the application. When using a permutation-based representation, this customization is hidden inside the schedule builder. Changes from one application to another only require that a new schedule builder be constructed for that particular application. This makes the use of permutation-based representations rather flexible.

Whitley et al. (1989) first used a strict permutation-based representation in conjunction with genetic algorithms for real-world applications. However, Davis (1985b) had previously used "an intermediary, encoded representation of schedules that is amenable to crossover operations, while employing a decoder that always yields legal solutions to the problem." This is also a strategy later adopted by Syswerda (1991) and Syswerda and Palmucci (1991), which he enhanced by refining the set of available recombination operators.

Typically, simple genetic algorithms encode solutions using bit-strings, which enable the use of “standard” crossover operators such as 1-point and 2-point (Goldberg 1989b). However, when solutions for scheduling problems are encoded as permutations, a special crossover operator is required to ensure that the recombination of two parent permutations results in a child that (1) inherits good characteristics of both parents and (2) is still a legal permutation. Numerous crossover operators have been proposed for permutations representing scheduling problems. For instance, Syswerda’s (1991) *order* and *position* crossover are methods for producing legal permutations that inherit various ordering or positional elements from parents.

Syswerda’s order crossover and position crossover differ from other permutation crossover operators such as Goldberg’s Partially Mapped Crossover (PMX) operator (Goldberg and Lingle 1985) or Davis’ order crossover (Davis 1985a) in that no contiguous block is directly passed to the offspring. Instead, several elements are randomly selected by absolute position. These operators are largely used for scheduling applications (e.g., Syswerda 1991; Watson et al. 1999; Syswerda and Palmucci 1991 for Syswerda’s operator) and are distinct from the permutation recombination operators that have been developed for the traveling salesman problem (Nagata and Kobayashi 1997; Whitley et al. 1989). Operators that work well for scheduling applications do not work well for the traveling salesman problem, and operators that work well for the traveling salesman problem do not work well for scheduling.

Syswerda’s order crossover operator can be seen as a generalization of Davis’ order crossover (Davis 1991) that also borrows from the concept of uniform crossover for bit strings. Syswerda’s order crossover operator starts by selecting K uniform-random positions in Parent 2. The corresponding elements from Parent 2 are then located in Parent 1 and reordered, so that they appear in the same relative order as they appear in Parent 2. Elements in Parent 1 that do not correspond to selected elements in Parent 2 are passed directly to the offspring.

```
           Parent 1 :  "A B C D E F G"
           Parent 2 :  "C F E B A D G"
Selected Elements :      *   * *
```

The selected elements in Parent 2 are F B and A in that order. A remapping operator reorders the relevant elements in Parent 1 in the same order found in Parent 2.

```
"A B _ _ _ F _"  remaps to  "FB _ _ _ A _"
```

The other elements in Parent 1 are untouched, thus yielding

```
"F B C D E A G"
```

Syswerda also defined a “position crossover.” Whitley and Yoo (1995) prove that Syswerda’s order crossover and position crossover are identical in expectation when order crossover selects K positions and position crossover selects $L-K$ positions over permutations of length L.

7.1 The Coors Warehouse Scheduling Problem

The Coors production facility (circa 1990) consists of 16 production lines, a number of loading docks, and a warehouse for product inventory. At the time this research was originally carried out (Starkweather et al. 1991), each production line could manufacture approximately 500 distinct products. Orders could be filled directly from the production lines or from inventory.

A solution is a priority ordering of customer orders. While the ultimate resource is the product that is being ordered, another limiting resource are the loading docks. Once a customer order is selected to be filled, it is assigned a loading dock and (generally) remains at the dock until it is completely filled, at which point the dock becomes empty and available for another order. All orders compete for product from either the production line or inventory. Note that scheduling also has secondary effects on how much product is loaded from the production line onto a truck (or train) and how much must temporarily be placed in inventory.

The problem representation used here is permutation of customer orders; the permutation queues up the customer orders which then wait for a vacant loading dock. When a dock becomes free, an order is removed from the queue and assigned to the dock.

A simulation is used to compute the evaluation function. Assume we wish to schedule one 24 h period. The simulation determines how long it takes to fill each order and how many orders can be filled in 24 h. The simulation must also track which product is drawn out of inventory, how much product is directly loaded off the production line, and how much product must first go into inventory until it is needed.

Figure 9 illustrates how a permutation is mapped to a schedule. Customer orders are assigned a dock based on the order in which they appear in the permutation; the permutation in effect acts as a customer priority queue. In the right-hand side of the illustration, note that initially customer orders **A–I** get first access to the docks (in a left to right order). **C** finishes first, and the next order, **J**, replaces **C** at the dock. Order **A** finishes next and is replaced by **K**. **G** finishes next and is replaced by **L**.

For the Coors warehouse scheduling problem, one is interested in producing schedules that simultaneously achieve two goals. One of these is to minimize the mean time that customer orders to remain at dock. Let N be the number of customer orders. Let M_i be the time that the truck or rail car holding customer order i spends at dock. Mean time at dock, $\mathcal{M}$, is then given by

$$\mathcal{M} = \frac{1}{N}\sum_{i=0}^{N} M_i.$$

The other goal is to minimize the running average inventory. Let F be the makespan of the schedule. Let $\mathcal{J}_t$ be inventory at time t. The running average inventory, I, is given by

$$I = \frac{1}{F}\sum_{t=0}^{F} \mathcal{J}_t.$$

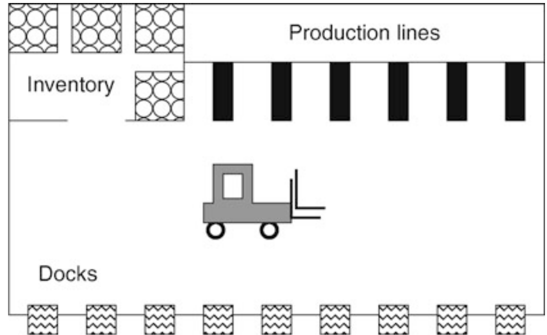
Technically this is a multi-objective problem. This problem was transformed into a single-objective problem using a linear combination of the individual objectives:

$$\text{obj} = \frac{(\mathcal{M} - \mu_{\mathcal{M}})}{\sigma_{\mathcal{M}}} + \frac{(I - \mu_I)}{\sigma_I} \quad (9)$$

where I represents running average inventory, $\mathcal{M}$ represents order mean time at dock, while μ and σ represent the respective means and standard deviations over a set of solutions.

In *Table 1*, results are given for the Genitor steady-state genetic algorithm compared to a stochastic hill-climber (Watson et al. 1999). Mean time at dock and average inventory are reported, also with performance and standard deviations over 30 runs. The move operator for the hill-climber was an "exchange operator." This operator selects two random customers and then swaps their position in the permutation. All of the algorithms reported here

Fig. 9
The warehouse model includes production lines, inventory, and docks. The columns in the schedule represent different docks. Customer orders are assigned to a dock left to right and product is drawn from inventory and the production lines.



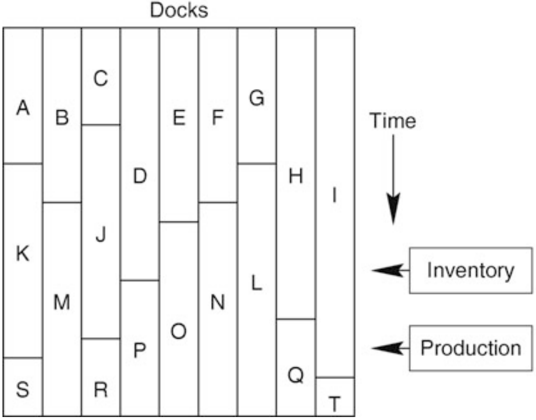


Table 1
Performance results. The final column indicates a human-generated solution used by Coors

	Genetic algorithm	Hill climber	Coors
Mean time-at-dock			
μ	392.49	400.14	437.55
σ	0.2746	4.7493	n.a.
Average inventory			
μ	364,080	370,458	549,817
σ	1,715	20,674	n.a.

used 100,000 function evaluations. The Genitor algorithm used a population size of 500 and a selective pressure of 1.1.

For our test data, we have an actual customer order sequence developed and used by Coors personnel to fill customer orders. This solution produced an average inventory of 549,817.25 product units and an order mean time of 437.55 min at dock.

Genitor was able to improve the mean time at dock by approximately 9%. The big change, however, is in average inventory. Both Genitor and the Hill Climber show a dramatic reduction in average inventory, meaning more product came out of inventory and that the schedule did a better job of directly loading product off the line. Watson et al. (1999) provide a more detailed discussion of the Coors warehouse scheduling application.

8 Conclusions

This paper has presented a broad survey of both theoretical and practical work related to genetic algorithms. For the reader who may be interested in using genetic algorithms to solve a particular problem, two comments might prove to be useful.

First, the details matter. A small change in representation or a slight modification in how an algorithm is implemented can change algorithm performance. The literature is full of papers that claim one algorithm is better than the other. What is often not obvious is how hard the researchers had to work to get good performance and how sensitive the results are to tuning the search algorithms? Are the benchmarks representative of real-world problems, and do the results generalize?

Second, genetic algorithms are a general purpose approach. An application-specific solution will almost always be better than a general purpose solution. Of course, in virtually every real-world application, application-specific knowledge gets integrated into the evaluation function, the operators, and the representation. Doing this well takes a good deal of experience and intuition about how to solve optimization and search problems which makes the practice of search algorithm design somewhat of a specialized art. This observation leads back to the first point: the details matter, and a successful implementation usually involves keen insight into the problem.

Acknowledgments

This research was partially supported by a grant from the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number FA9550-08-1-0422. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notation thereon. Funding was also provided by the Coors Brewing Company, Golden, Colorado.

References

Bäck T (1996) Evolutionary algorithms in theory and practice. Oxford University Press, Oxford

Baker J (1987) Reducing bias and inefficiency in the selection algorithm. In: Grefenstette J (ed) GAs and their applications: 2nd international conference, Erlbaum, Hilsdale, NJ, pp 14–21

Bitner JR, Ehrlich G, Reingold EM (1976) Efficient generation of the binary reflected gray code and its applications. Commun ACM 19 (9):517–521

Brent R (1973) Algorithms for minization with derivatives. Dover, Mineola, NY

Bridges C, Goldberg D (1987) An analysis of reproduction and crossover in a binary coded genetic algorithm. In: Grefenstette J (ed) GAs and their applications: 2nd international conference, Erlbaum, Cambridge, MA

Davis L (1985a) Applying adaptive algorithms to epistatic domains. In: Proceedings of the IJCAI-85, Los Angeles, CA

Davis L (1985b) Job shop scheduling with genetic algorithms. In: Grefenstette J (ed) International conference on GAs and their applications. Pittsburgh, PA, pp 136–140

Davis L (1991) Handbook of genetic algorithms. Van Nostrand Reinhold, New York

DeJong K (1993) Genetic algorithms are NOT function optimizers. In: Whitley LD (ed) FOGA – 2, Morgan Kaufmann, Los Altos, CA, pp 5–17

Eshelman L (1991) The CHC adaptive search algorithm: how to have safe search when engaging in nontraditional genetic recombination. In: Rawlins G (ed) FOGA – 1, Morgan Kaufmann, Los Altos, CA, pp 265–283

Eshelman L, Schaffer D (1991) Preventing premature convergence in genetic algorithms by preventing incest. In: Booker L, Belew R (eds) Proceedings of the 4th international conference on GAs. Morgan Kaufmann, San Diego, CA

Goldberg D (1987) Simple genetic algorithms and the minimal, deceptive problem. In: Davis L (ed) Genetic algorithms and simulated annealing. Pitman/ Morgan Kaufmann, London, UK, chap 6

Goldberg D (1989a) Genetic algorithms and Walsh functions: Part II, deception and its analysis. Complex Syst 3:153–171

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

Goldberg D (1990) A note on Boltzmann tournament selection for genetic algorithms and population-oriented simulated annealing. Tech. Rep. Nb. 90003. Department of Engineering Mechanics, University of Alabama, Tuscaloosa, AL

Goldberg D, Deb K (1991) A comparative analysis of selection schemes used in genetic algorithms. In: Rawlins G (ed) FOGA – 1, Morgan Kaufmann, San Mateo, CA, pp 69–93

Goldberg D, Lingle R (1985) Alleles, loci, and the traveling salesman problem. In: Grefenstette J (ed) International conference on GAs and their applications. London, UK, pp 154–159

Grefenstette J (1993) Deception considered harmful. In: Whitley LD (ed) FOGA – 2, Morgan Kaufmann, Vail, CO, pp 75–91

Hansen N (2006) The CMA evolution strategy: a comparing review. In: Toward a new evolutionary computation: advances on estimation of distribution algorithms. Springer, Heidelberg, Germany, pp 75–102

Hansen N (2008) Adaptive encoding: how to render search coordinate system invariant. In: Proceedings of 10th international conference on parallel problem solving from nature. Springer, Dortmund, Germany, pp 205–214

Heckendorn R, Rana S, Whitley D (1999a) Polynomial time summary statistics for a generalization of MAXSAT. In: GECCO-99, Morgan Kaufmann, San Francisco, CA, pp 281–288

Heckendorn R, Rana S, Whitley D (1999b) Test function generators as embedded landscapes. In: Foundations of genetic algorithms FOGA – 5, Morgan Kaufmann, Los Atlos, CA

Heckendorn RB, Whitley LD, Rana S (1996) Nonlinearity, Walsh coefficients, hyperplane ranking and the simple genetic algorithm. In: FOGA – 4, San Diego, CA

Ho Y (1994) Heuristics, rules of thumb, and the 80/20 proposition. IEEE Trans Automat Cont 39(5): 1025–1027

Ho Y, Sreenivas RS, Vakili P (1992) Ordinal optimization of discrete event dynamic systems. Discrete Event Dyn Syst 2(1):1573–7594

Holland J (1975) Adaptation in natural and artificial systems. University of Michigan Press, Ann Arbor, MI

Holland JH (1992) Adaptation in natural and artificial systems, 2nd edn. MIT Press, Cambridge, MA

Schaffer JD, Eshelman L (1993) Real-coded genetic algorithms and interval schemata. In: Whitley LD (ed) FOGA – 2, Morgan Kaufmann, Los Atlos, CA

Mathias KE, Whitley LD (1994) Changing representations during search: a comparative study of delta coding. J Evolut Comput 2(3):249–278

Nagata Y, Kobayashi S (1997) Edge assembly crossover: a high-power genetic algorithm for the traveling salesman problem. In: Bäck T (ed) Proceedings of the 7th international conference on GAs, Morgan Kaufmann, California, pp 450–457

Nix A, Vose M (1992) Modelling genetic algorithms with Markov chains. Ann Math Artif Intell 5:79–88

Poli R (2005) Tournament selection, iterated coupon-collection problem, and backward-chaining evolutionary algorithms. In: Foundations of genetic algorithms, Springer, Berlin, Germany, pp 132–155

Radcliffe N, Surry P (1995) Fundamental limitations on search algorithms: evolutionary computing in perspective. In: van Leeuwen J (ed) Lecture notes in computer science, vol 1000, Springer, Berlin, Germany

Rana S, Whitley D (1997) Representations, search and local optima. In: Proceedings of the 14th national conference on artificial intelligence AAAI-97. MIT Press, Cambridge, MA, pp 497–502

Rana S, Heckendorn R, Whitley D (1998) A tractable Walsh analysis of SAT and its implications for genetic algorithms. In: AAAI98, MIT Press, Cambridge, MA, pp 392–397

Rosenbrock H (1960) An automatic method for finding the greatest or least value of a function. Comput J 3:175–184

Salomon R (1960) Reevaluating genetic algorithm performance under coordinate rotation of benchmark functions. Biosystems 39(3):263–278

Schaffer JD (1987) Some effects of selection procedures on hyperplane sampling by genetic algorithms. In: Davis L (ed) Genetic algorithms and simulated annealing. Morgan Kaufmann, San Francisco, CA, pp 89–130

Schwefel HP (1981) Numerical optimization of computer models. Wiley, New York

Schwefel HP (1995) Evolution and optimum seeking. Wiley, New York

Sokolov A, Whitley D (2005) Unbiased tournament selection. In: Proceedings of the 7th genetic and evolutionary computation conference. The Netherlands, pp 1131–1138

Spears W, Jong KD (1991) An analysis of multi-point crossover. In: Rawlins G (ed) FOGA – 1, Morgan Kaufmann, Los Altos, CA, pp 301–315

Starkweather T, Whitley LD, Mathias KE (1990) Optimization using distributed genetic algorithms. In: Schwefel H, Männer R (eds) Parallel problem solving from nature. Springer, London, UK, pp 176–185

Starkweather T, McDaniel S, Mathias K, Whitley D, Whitley C (1991) A comparison of genetic sequencing operators. In: Booker L, Belew R (eds) Proceedings of the 4th international conference on GAs. Morgan Kaufmann, San Mateo, CA, pp 69–76

Suh J, Gucht DV (1987) Distributed genetic algorithms. Tech. rep., Indiana University, Bloomington, IN

Syswerda G (1989) Uniform crossover in genetic algorithms. In: Schaffer JD (ed) Proceedings of the 3rd international conference on GAs, Morgan Kaufmann, San Mateo, CA

Syswerda G (1991) Schedule optimization using genetic algorithms. In: Davis L (ed) Handbook of genetic algorithms, Van Nostrand Reinhold, New York, chap 21

Syswerda G, Palmucci J (1991) The application of genetic algorithms to resource scheduling. In: Booker L, Belew R (eds) Proceedings of the 4th international conference on GAs, Morgan Kaufmann, San Mateo, CA

Vose M (1993) Modeling simple genetic algorithms. In: Whitley LD (ed) FOGA – 2, Morgan Kaufmann, San Mateo, CA, pp 63–73

Vose M (1999) The simple genetic algorithm. MIT Press, Cambridge, MA

Vose M, Liepins G (1991) Punctuated equilibria in genetic search. Complex Syst 5:31–44

Vose M, Wright A (1997) Simple genetic algorithms with linear fitness. Evolut Comput 2(4):347–368

Watson JP, Rana S, Whitley D, Howe A (1999) The impact of approximate evaluation on the performance of search algorithms for warehouse scheduling. J Scheduling 2(2):79–98

Whitley D (1999) A free lunch proof for gray versus binary encodings. In: GECCO-99, Morgan Kaufmann, Orlando, FL, pp 726–733

Whitley D, Kauth J (1988) GENITOR: A different genetic algorithm. In: Proceedings of the 1988 Rocky Mountain conference on artificial intelligence, Denver, CO

Whitley D, Rowe J (2008) Focused no free lunch theorems. In: GECCO-08, ACM Press, New York

Whitley D, Yoo NW (1995) Modeling permutation encodings in simple genetic algorithm. In: Whitley D, Vose M (eds) FOGA – 3, Morgan Kaufmann, San Mateo, CA

Whitley D, Starkweather T, Fuquay D (1989) Scheduling problems and traveling salesmen: the genetic edge recombination operator. In: Schaffer JD (ed) Proceedings of the 3rd international conference on GAs. Morgan Kaufmann, San Francisco, CA

Whitley D, Das R, Crabb C (1992) Tracking primary hyperplane competitors during genetic search. Ann Math Artif Intell 6:367–388

Whitley D, Beveridge R, Mathias K, Graves C (1995a) Test driving three 1995 genetic algorithms. J Heuristics 1:77–104

Whitley D, Mathias K, Pyeatt L (1995b) Hyperplane ranking in simple genetic algorithms. In: Eshelman L (ed) Proceedings of the 6th international conference on GAs. Morgan Kaufmann, San Francisco, CA

Whitley D, Mathias K, Rana S, Dzubera J (1996) Evaluating evolutionary algorithms. Artif Intell J 85:1–32

Whitley LD (1989) The GENITOR algorithm and selective pressure: why rank based allocation of reproductive trials is best. In: Schaffer JD (ed) Proceedings of the 3rd international conference on GAs. Morgan Kaufmann, San Francisco, CA, pp 116–121

Whitley LD (1991) Fundamental principles of deception in genetic search. In: Rawlins G (ed) FOGA – 1, Morgan Kaufmann, San Francisco, CA, pp 221–241

Whitley LD (1993) An executable model of the simple genetic algorithm. In: Whitley LD (ed) FOGA – 2, Morgan Kaufmann, Vail, CO, pp 45–62

Wolpert DH, Macready WG (1995) No free lunch theorems for search. Tech. Rep. SFI-TR-95-02-010, Santa Fe Institute, Santa Fe, NM