An Overview of Evolutionary Algorithms: Practical Issues and Common Pitfalls

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

Abstract

An overview of evolutionary algorithms is presented covering genetic algorithms, evolution strategies, genetic programming and evolutionary programming. The *schema theorem* is reviewed and critiqued. Gray codes, bit representations and real-valued representations are discussed for parameter optimization problems. Parallel Island models are also reviewed, and the evaluation of evolutionary algorithms is discussed.

Keywords: genetic algorithms, evolution strategies, genetic programming, evolutionary programming, search, automated programming, parallel algorithms

1 Introduction

Evolutionary Algorithms have become popular tools for search, optimization, machine learning and for solving design problems. These algorithms use simulated evolution to search for solutions to complex problems. There are many different types of evolutionary algorithms. Historically, *genetic algorithms* and *evolution strategies* are two of the most basic forms of evolutionary algorithms. Genetic algorithms were developed in the United States under the leadership of John Holland and his students. This tradition puts a great deal of emphasis on selection, recombination and mutation acting on a genotype that is decoded and evaluated for fitness. Recombination is emphasized over mutation. Evolution strategies were developed in Germany under the leadership of Ingo Rechenberg and Hans-Paul Schwefel and their students. Evolution strategies tend to use more direct representations [3]. Mutation is emphasized over recombination. Both genetic algorithms and evolution strategies have been used for optimization. But genetic algorithms have long been viewed as multipurpose tools with applications in search, optimization, design and machine learning [24, 18], while most of the work in evolution strategies has focused on optimization [39, 40, 2]. In the last decade, these two fields have influenced each other and many new algorithms freely borrow ideas from both traditions.

In the last ten years, *genetic programming* has also become an important new subarea of evolutionary algorithms [30, 27]. Genetic programming has been explicitly developed as an evolutionary methodology for automatic programming and machine learning. Design applications have also proven to be important. Another subarea of evolutionary computing is *evolutionary programming*. Evolutionary programming has its roots in the 1960's [16] but was inactive for many years until being reborn in the 1990's [14] in a new form that is extremely similar to *evolution strategies*.

Each of these paradigms has its own strengths and weaknesses. One goal of this overview is to highlight each model so that users can better decide which methods are best suited for particular types of applications. There are also some general high level concepts that are basic to evolutionary algorithms that might be applied in conjunction with any of the various paradigms. The use of a parallel evolutionary algorithm can often boost performance. The *island model* in particular has low cost in terms of software development and can have a significant impact on performance. This overview also addresses the question as to when it is reasonable to use an evolutionary algorithm, and suggests other methods to utilize in order to evaluate the effectiveness of an evolutionary algorithm.

2 Genetic Algorithms

Genetic algorithms remain the most recognized form of evolutionary algorithms. John Holland and his students worked on the development of these algorithms in the 1960's, 70's and 80's. In the mid-1980's these algorithms started to reach other research communities–such as the machine learning and operations research communities. It is probably no coincidence that the explosion of research in genetic algorithms came soon after the explosion of research in articial neural networks. Both areas of research draw inspiration from biological systems as a computational and motivational model. In the current paper, a high level overview is given with the goal of providing some practical guidance to users as well an overview of more recent results. (For another tutorial on genetic algorithms see [51].)

Genetic algorithms emphasize the use of a “genotype” that is decoded and evaluated. These genotypes are often simple data structures. Often, the chromosomes are bit strings which can be recombined in a simplied form of “sexual reproduction” and can be mutated by simple bit flips. These algorithms can be described as function optimizers. This does not mean that they yield globally optimal solutions. Instead, Holland (in the introduction to the 1992 edition of his 1975 book [25]) and DeJong [9] have both emphasized that these algorithms find competitive solutions, but both also suggest that it is probably best to view genetic algorithms as a search process rather than strictly as an optimization process. As such, competition as implemented by “selection of the fittest” is a key aspect of *genetic search*.

An example application provides a useful vehicle for explaining certain aspects of these algorithms. Assume one wishes to optimize some process, such as paper production with the goal of maximizing quality. Assume we have 3 parameters we can control in the production process, such as temperature, pressure and some mixture parameter that controls the use of recycled paper versus pulp. (The goal is not to make these parameters overly realistic, but rather to illustrate a generic parameter optimization problem.) This can be viewed as 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.

One could represent the 3 parameters using three real valued parameters, such as

$$< 32.56, 18.21, 9.83 >$$

or the 3 parameters could be represented as bit strings, such as

$$< 000111010100, 110100101101, 001001101011 > .$$

Of course, this 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 potentially be important. Historically, genetic algorithms have typically been implemented using low precision, such as 10 bits per parameter.

Recombination is central to genetic algorithms. 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
```

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

```
11010yxxyyyxyxxy          and          yxyyx01100101101
```

Note that parameter boundaries are ignored.

After recombination, we can apply a mutation operator. For each bit in the population, mutate with some low probability p_m. Typically the mutation rate is applied with less than 1% probability.

In 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 population*, as shown in Figure 1. Then recombination and mutation are applied to the intermediate population to create the *next population.* The process of going from the current population to the next population constitutes one generation in the execution of a genetic algorithm.

We will first consider the construction of the intermediate population from the current population. In the first generation the current population is also the initial population. In the canonical genetic algorithm, fitness is defined by $f_i/\bar{f}$, where f_i is the evaluation associated with string i and $\bar{f}$ is the average evaluation of all the strings in the population. This is known as *fitness proportional reproduction.*

The 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

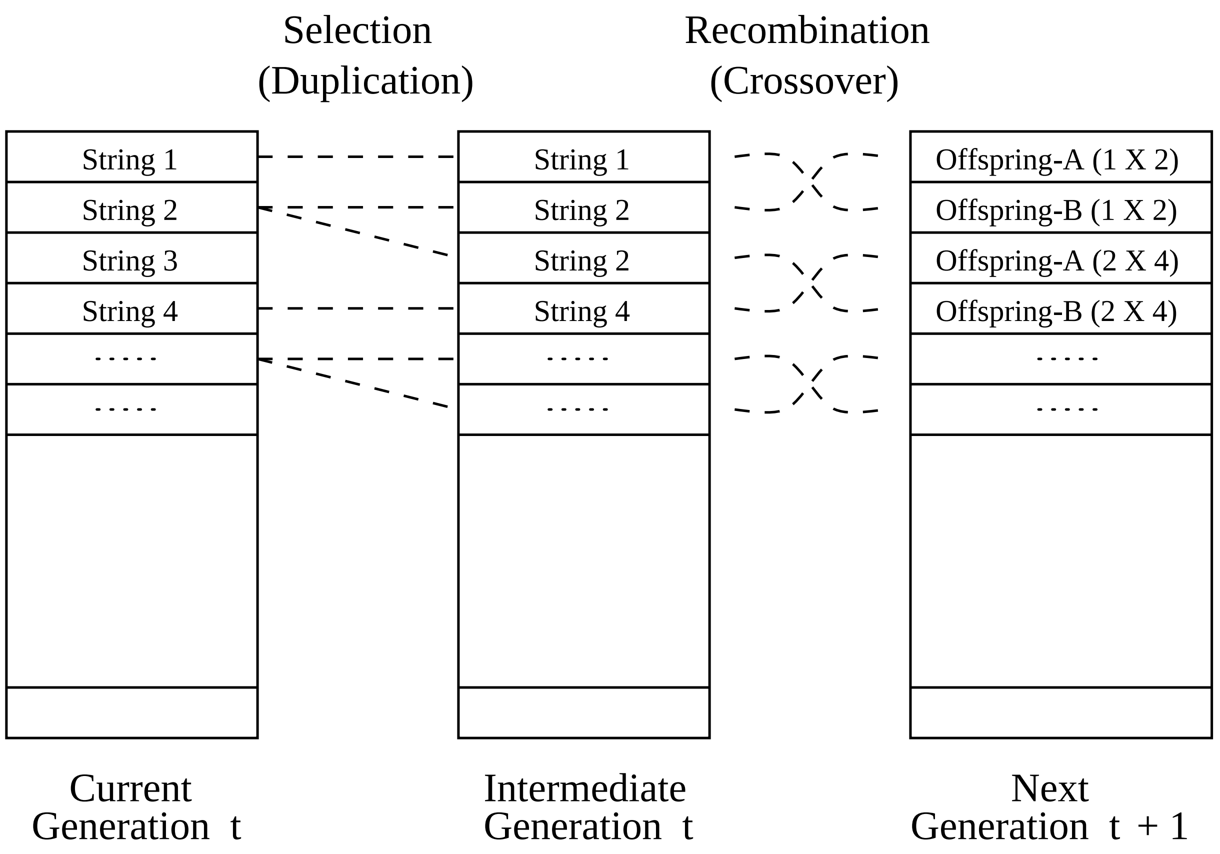


Figure 1: *One generation is broken down into a selection phase and recombination phase. This figure shows strings being assigned into adjacent slots during selection. In fact, they can be assigned slots randomly in order to shuffle the intermediate population. Mutation (not shown) can be applied after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

is carried out. In the canonical genetic algorithm the probability that strings in the current population are copied (i.e., duplicated) and placed in the intermediate generation is 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; break f_i into an integer part, x_i, and a remainder, r_i. 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 zero 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 Figure 2.

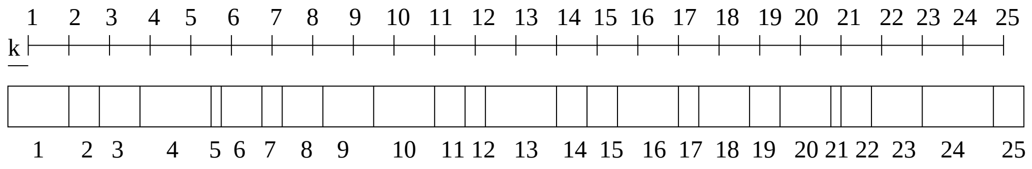


Figure 2: *Stochashtic Universal Sampling. The fitnesses of the population can be seen as being laid out on a number line in random order a shown at the bottom of the figure. A single random value, $0 \leq k \leq 1$, shifts the uniformly spaced "pointers" which now selects the member of the next intermediate population.*

This same mechanism can viewed as a roulette wheel. The roulette wheel has N equally spaced pointers. The choice of k in effect spins the roulette wheel and the position of the evenly space pointers, thereby simultaneously picking all N members of the intermediate population. The resulting selection is unbiased [4].

After selection has been executed, the construction of the intermediate population is complete. The *next generation* of the population is created from from the 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 (in a sense replacing the parents). If no recombination occurs, the parents can pass directly into the next generation. However, as a last step mutation is applied.

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.

There can be a couple of problems with *fitness proportional reproduction.* First selection can be too strong in the first few generations: too many duplicates are sometimes allocated to very good individuals found early in the search. Second, as individuals in the population improve over time, there tends to be less variation in fitness, with more individuals 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. In this case, the search begins to stagnate.

The selection mechanism can also be based on a rank-based mechanism. Assume the population is sorted by fitness. A linear ranking mechanism with bias 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 selection bias between the best and worst member of the population is constant, 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 [50].

Another fast but noisy way to implement ranking is *Tournament Selection* [19, 17] To construct the intermediate population, select two strings at random and place the best in the intermediate population. In expectation, every string is sampled twice. The best string wins both tournaments and gets 2 copies in the intermediate population. The median string wins one and loses one and gets 1 copy in the intermediate population. The worse strings loses both tournaments and does not reproduce. In expectation, this produces a linear ranking with a bias 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 bias is less than 2.0. If a tournament size larger than 2 is used and the winner is choosen deterministic, then the bias is greater than 2.0.

2.1 Schemata and Hyperplanes.

In his 1975 book, *Adaptation in Natural and Artificial Systems* [24], Holland develops the concepts of schemata and hyperplane sampling to explain how a genetic algorithm can yield a robust search by implicitly sampling hyperplane partitions of a search space. Since 1975, the concepts of schemata and hyperplane sampling have become the central concepts in what at times seems like a religious war. The idea that genetic algorithms search by hyperplane sample is now controversial, but the debate over this issue continues to generate more heat (and smoke) than light.

A bit string matches a particular schemata if that bit string can be constructed from the schemata by replacing the "*" symbol with the appropriate bit value. 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. In general, all bit strings that match a particular schemata are contained in the hyperplane partition represented by that particular schemata. The string of all * symbols corresponds to the space itself and is not counted as a partition of the space. There are 3^L possible schemata since there are L positions in the bit string and each position can be a 0,1, or * symbol.

The notion of a population based search is critical to genetic algorithms. A population of sample points provides information about numerous hyperplanes; furthermore, low order hyperplanes should be sampled by numerous points in the population. 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.

2.2 An Illustration of Hyperplane Sampling

Holland [24] suggested the following view of hyperplane sampling. In Figure 3 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 Figure 3 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 points of Figure 3 is that the sampling of hyperplane partitions is not really affected by local minima. At the same time, increasing the sampling rate of partitions that are above average compared to other competing partitions does not guarantee convergence to a global optimum. The global optimum could be a relatively isolated peak, for example. Nevertheless, good solutions that are globally competitive are often found. The notion that hyperplane sampling is a useful way to guide search should be viewed as a 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 more detail in Section 3.

2.3 The Schema Theorem

Holland [24] 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. This weakness is just one of many reasons that the concept of "hyperplane sampling" is controversial.

Let $P(H,t)$ be the proportion of the population that samples hyperplane H at time t. Let $P(H, t + \mathit{intermediate})$ 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 important because the average fitness of the population is *not* constant.)

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

Thus, ignoring crossover and mutation, 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. 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 this effect is not limited to the 3^L hyperplanes that Holland considered to be relevant. Selection acts exactly the same way on any aribtrarily choosen 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.

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 $t + 1$. We first just consider selection and crossover.

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

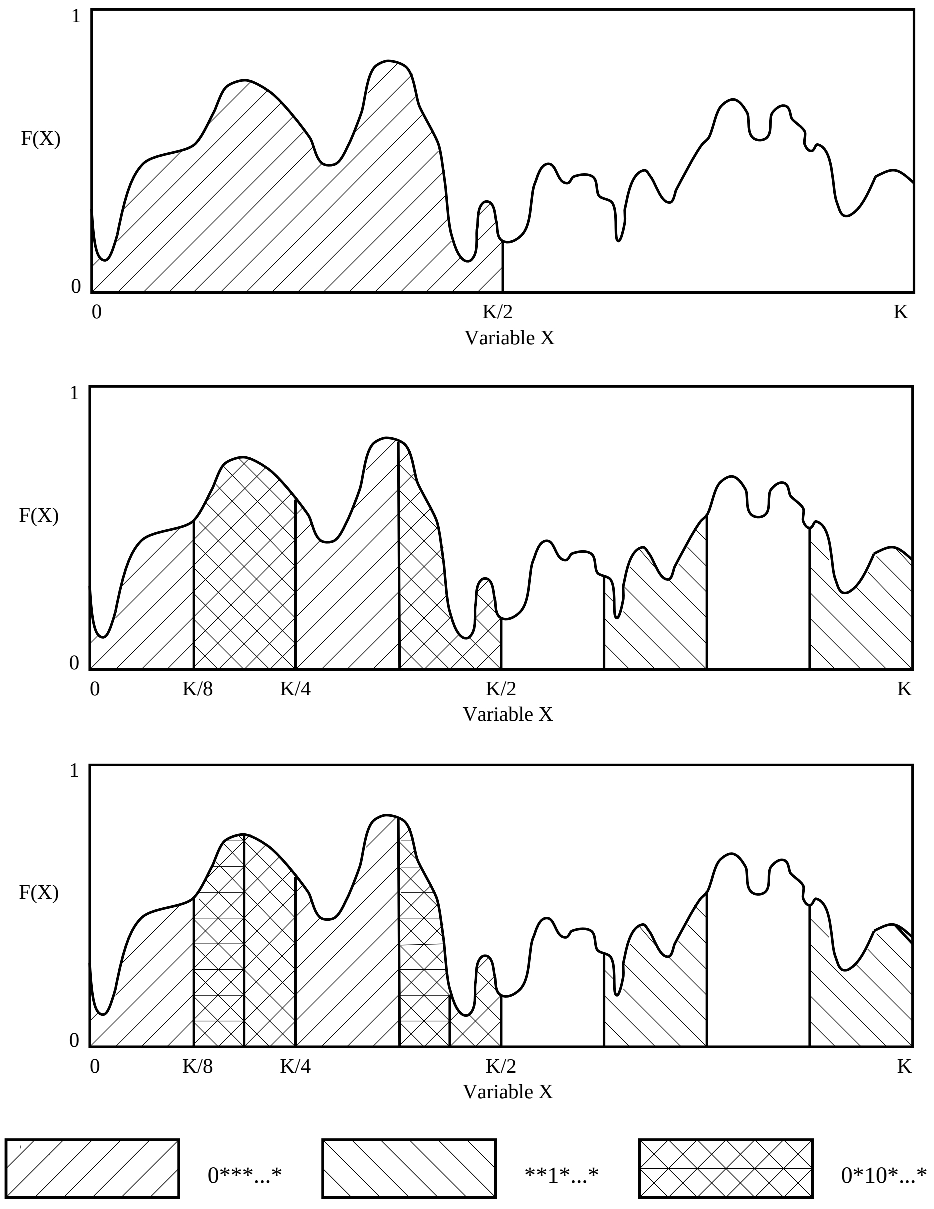


Figure 3: *A function and various partitions of hyperspace. Fitness is scaled to a 0 to 1 range in this diagram.*

where p_c is the probability of doing crossover. When crossover does not occur (which happens with probability $(1 - p_c)$, then only selection changes the sampling rate. However, when crossover does occur (with probability p_c) then we have to consider how crossover 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 sources of *gains* that is referred to in the above calculation. To simplify things, *gains* are ignored and the conservative assumption is made that crossover falling in the significant portion of a schema always leads to disruption. Thus,

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

The *defining length* of a schemata is based on the distance between the first and last bits in the schema with value either 0 or 1 (i.e., not a * symbol). Given that each position in a schema can be 0, 1 or *, then scanning left to right, if I_x is the index of the position of the rightmost occurrence of either a 0 or 1 and I_y is the index of the leftmost occurrence of either a 0 or 1, then the defining length is merely $I_x - I_y$. The defining length of a schema representing a hyperplane H is denoted here by $\Delta(H)$. If 1-point crossover is used, then disruption is bounded by:

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

and including this terms yields:

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

We now have a useful version of the schema theorem (although it does not yet consider mutation); but it is not the only version in the literature. For example, this version assumes that selection for the first parent string is fitness based and the second parent is chosen randomly. But we have also examined a form of the simple genetic algorithm where both parents are chosen based on fitness. This can be added to the schema theorem by merely indicating the alternative parent is chosen from the intermediate population after selection [38].

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

Finally, mutation is included. Let $o(H)$ be a function that returns the order of the hyperplane H. The order of H exactly corresponds to a count of the number of bits in the schema representing H that have value 0 or 1. Let the mutation probability be p_m where mutation always flips the bit. Thus the probability that mutation does not affect the schema representing H is $(1-p_m)^{o(H)}$. This leads to the following expression of the schema theorem.

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

3 Some Criticisms of the Schema Theorem

There are many different criticisms of the schema theorem. First of all, it is an inequality, and it only applies for one generation into the future. So while the bound provided by the schema theorem absolutely holds for one generation into the future, it says nothing about how trials will be allocated in future generations.

It is also true that the schema theorem does hold true independently for all possible hyperplanes for 1 generation. However, over over multiple generations these dependencies 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 and the schema theorem indicates that both have increasing representation. But trials allocated to schemata 11****** and *00***** are in conflict (because they disagree about the value of the second bit). Over time, both regions cannot receive increasing trials. These schema are *inconsistent* about what bit value is preferred in the second position.

Whitley et al. [48, 23] have shown that problems can have varying degrees of consistency. For problems that display higher *consistency*, the "most fit" schemata tend to agree about what the values of particular bits should be. And problems can be highly inconsistent, so that the most fit individuals 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.

One criticism of pragmatic significance is that users of the standard or canonical genetic algorithm often use small populations. The number of bits that value 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 (for example, 100) is inadequate unless we only care about relatively low order schemata. So even if hyperplane sampling is a robust form of heuristic search, the user destroys this potential by using small population sizes.

What if we had perfect schema information? What if we could compute schema information exactly in polynomial time? Rana et al. [37] have shown that schema information up to any fixed order can be computed in polynomial time for some NP-Complete problems. This includes MAXSAT problems and NK-Landscapes. This is very surprising. One theoretical consequence of this is the following:

If $P \neq NP$ then, in the general case, exactly knowing the static schema fitness averages up to some fixed order cannot provide information that can be used to infer the location of a global optimum, or even an above average solution, in polynomial time. (For proofs see: [22, 21]).

This seems like a very negative result. But it is dangerous to over-interpret either positive or negative results. In practice, random MAXSAT problems are characterized by highly inconsistent schema information–so that there is really little or no information that can be exploited to guide the search [22]. And in practice, genetic algorithms perform very poorly on MAXSAT problems [37]. 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 at best a heuristic.

There are many other criticisms of the schema theorem. Historically, too much has been claimed about schema and hyperplane processing that is not backed up by solid proofs. A kind of folklore grew up around the schema theorem in the 1970's and 1980's. There is no longer any evidence to support the claim 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 counter examples 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 true; 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 they are relatively large regions. There is still a great deal of work to be done to understand the role that hyperplane sampling plays in genetic search.

4 Evolution Strategies

About the same time Holland and his students were developing "genetic algorithms" in the late 1960's and early 1970's in the United States, Ingo Rechenberg and Hans-Paul Schwefel and others were working in Germany developing *evolution strategies*. Historically, these algorithms developed more or less independently and in very different directions. Evolution strategies are generally applied to real-valued representations of optimization problems, and tend to emphasize mutation over crossover. The algorithms are also often used with much smaller population sizes (e.g. 1 to 20 members of the population) than genetic algorithms.

The theory behind evolution strategies also developed in different and independent directions. There is no notion of schema processing associated with evolution strategies. The evolution strategies community was also more aggressive in exploring variations on the basic evolutionary algorithm and developed a notation to describe various population sizes and different ways of manipulating parents and offspring.

The two basic types of evolution strategies are known as the (μ, λ)-ES and the $(\mu + \lambda)$-ES. The symbol μ refers to the size of the parent population. The symbol λ refers to the number of offspring that are produced in a single generation before selection is applied. In a (μ, λ)-ES the offspring replace the parents. In a $(\mu + \lambda)$-ES selection picks from both the offspring and the parents to create the next generation. These variations on selection were explored in depth much earlier by the evolution strategies community than by the genetic algorithms community.

In practice, early evolution strategies were often simple. This is due in part because they executed on early simple computers–or were implemented on paper without the use of computers. Thus a (1+1)-ES has a single parent structure. The parent is modified to produce an offspring. And since this is a "+" strategy, selection picks the best of the parent and offspring to become the new parent. Clearly, this algorithm can be viewed as a hill-climber making some kind of random change and only accepting improving moves. Rechenberg also introduced the idea of using a $(\mu + 1)$-ES where a population of parents generates a single offspring; this might involve some kind of recombination. Since this is a "+" strategy, the worst member of the combined parent and offspring population is deleted. (Thus, the offspring survives only if it is better than one of the parents.)

In the $(\mu + \lambda)$-ES it is common that $\lambda > \mu$, and in fact, the number of offspring (λ) can be much larger than the number of parents (μ). In this case, some form of selection is used to prune back the population to only μ parents. This is reminiscent of biological species where many offspring are produced, but few survive to reproduce.

Another way in which evolution strategies differ from most genetic algorithms is that evolution strategies have long exploited *self-adaptive* mechanisms. The algorithms often include strategy parameters that are used to adapt the search to exploit the local topology of the search space.

Evolution strategies typically use a real-valued representation. Recombination is sometimes used, but mutation is generally the more emphasized operator. Because the representation is real-valued, what form should the mutation take? It is typically implemented as some distribution around the individual being mutated. A Gaussian distribution can be used with zero mean; the standard deviation must be specified. Given N parameters, the same standard deviation could be used for each Gaussian mutation operation, or a different standard deviation could be used by each of the N parameters. In practice, a log-normal distribution is now more commonly used in place of a Gaussian distribution.

One simple way in which the evolution strategy can be self-adaptive is to encode "strategy parameters" directly onto the "chromosome".

For example, a 3-parameter problem might have the encoding

$$< x_1, x_2, x_3, \sigma_1, \sigma_2, \sigma_3 >$$

where x_i is a parameter value and σ_i is the standard deviation used to mutate that particular parameter. We will refer to x_i as an "object parameter." It is also possible to include a "rotation parameter" for each pair of parameters. Including a vector/matrix of the rotation parameters could expand the encoding as follows.

$$< x_1, x_2, x_3, \sigma_1, \sigma_2, \sigma_3, \alpha_{1,2}, \alpha_{1,3}, \alpha_{2,3} >$$

where $\alpha_{i,j}$ is a rotation angle used to change the orientation of the mutation.

Note that there is one strategy parameter σ_i for every object parameter x_i. But there

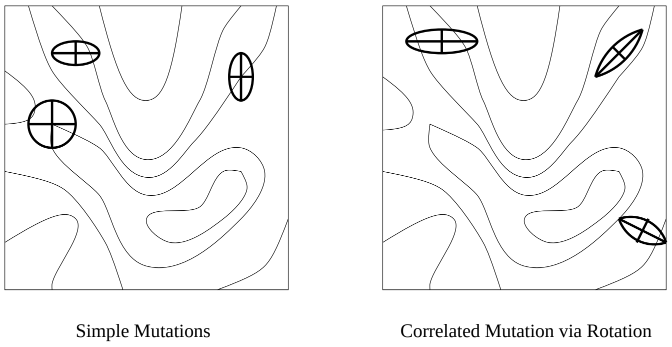


Figure 4: *Adaptive forms of mutation used by an Evolution Strategy.*

is a rotation parameter $\alpha_{i,j}$ for all possible pairs of object parameters. If there are n object parameters, then there are n σ strategy parameters and $n(n-1)/2$ α strategy parameters.

Figure 4 illustrates two cases of mutation. In each case, there are three members of the population, represented as hyperellipsoids. There are 2 parameters associated with each individual/chromosome. The fitness function is represented by contour lines representing equal evaluation. The leftmost graph shows individuals where there is a standard deviation associated with each parameter. In the rightmost graph, there is also a rotation associated with each individual; in this case, since there is only one pair of parameters, there is only one rotation to consider.

In vector notation, then, a chromosome can be denoted by

$$< \vec{x}, \vec{\sigma}, \vec{\alpha} > .$$

The question to be addressed is how mutation should be used to update the chromosome. The following description is based on Thomas Bäck's book *Evolutionary Algorithms in Theory and Practice* [2] and readers should refer to this work for more details.

Let $N(0,1)$ be a function returning a normally distributed one-dimensional random variable with zero mean and standard deviation one. Let $N_i(0,1)$ denote the same function, but with a new sample being drawn for each i. The symbols τ, τ' and β represent constants that control step sizes. Mutation then acts on a chromosome

$$< \vec{x}, \vec{\sigma}, \vec{\alpha} >$$

to yield a new chromosome

$$< \vec{x}', \vec{\sigma}', \vec{\alpha}' >$$

where $\forall i \in 1, ..., n, \forall j \in 1, ..., n(n-1)/2 :$

$$\sigma_i' = \sigma_i \exp(\tau' N(0,1) + \tau N_i(0,1))$$

$$\alpha_j' = \alpha_j + \beta N_j(0,1)$$

$$\vec{x}' = \vec{x} + \vec{N}(\vec{0}, \mathbf{C}(\vec{\sigma}', \vec{\alpha}'))$$

where $\vec{N}(\vec{0}, \mathbf{C})$ denotes a function that returns a random vector distribution that is normally distributed with zero mean and covariance matrix $\mathbf{C^{-1}}$. The "rotations" along with the variances are used to implement the covariance matrix. The variances form the diagonal of the covariance matrix ($c_{ii} = \sigma_i^2$).

The rotation angles are limited to the range $[\pi, -\pi]$. Rotations are implemented using sine and cosine functions. If mutation moves rotation outside of this range, it is circularly remapped back onto the the range $[\pi, -\pi]$.

Bäck [2] and Schwefel [39] suggest the following values for the constants.

$$\tau \propto \left(\sqrt{2\sqrt{n}}\right)^{-1}$$

$$\tau' \propto \left(\sqrt{2n}\right)^{-1}$$

$$\beta \approx 0.0873$$

Note that the strategy variable σ serves to determine the step size of the mutation that acts on the object parameters. The step size, σ, can also become very small. For this reason, a threshold is used so that σ is not allow to be smaller than ϵ_σ.

In practice, it is possible for σ to be driven down to the threshold ϵ_σ and to stay there. Given a relatively smooth evaluation function, shorter jumps are perhaps more likely to yield values similar to the fitness associated with the current set of object values. This is especially true as search continues and it becomes harder to find improving moves. Assuming the evolution strategy is a $(\mu + \lambda)$-ES, saving the improved solution also means saving the strategies value that produced that improved move. If there is a bias, such that shorter hops a more likely to yield an improvement that a longer hop, then σ will be driven toward the minimum possible value. Mathias et al. [31] suggest using some form of “restart” mechanism to open the search up again.

Also, just because a particular strategy variable results in a very good move, it does not automatically imply that the strategy variable is a good one. But with a $(\mu+\lambda)$-ES selection acts on the object variables and not on the strategy variables. Thus, a good individual with poor strategy variables can stay in the popoulation. So a (μ, λ)-ES selection mechanism where children replace parents can sometimes be more effective than a $(\mu + \lambda)$-ES selection mechanism.

One of the well-known theoretical results of evolution strategies is the *1/5 success rule*: on average, one out of five mutations should yield an improvement in order to achieve the best convergence rate. There are, of course, very special conditions under which such a result holds. First, the algorithm for which these theoretical results have been developed is a simple (1+1)-ES. Second, the results hold for two relatively simple functions, a function with a simple linear form and a function with a simple quadratic form. Again, in practice, the 1/5 success rule may imply shorter and shorter hops as one moves infinitely closer to a (potentially nonglobal) optimum.

Recombination is sometimes used in evolution strategies, but there has been less empirical and theoretical work looking at the use of recombination. Since real-valued representations are used, how should recombination be done? Averaging of two or more parents is one strategy. Eiben and Bäck [1] present an empirical study of the use of multiparent recombination operators in evolution strategies.

For many benchmark parameter optimization test problems a $(\mu + \lambda)$-evolution strategy yields better results than a canonical genetic algorithm. This is particularly true if the objective function is relatively smooth. On the other hand, the canonical genetic algorithm is a (μ, λ) evolutionary algorithm, with offspring replacing parents, so perhaps such a comparison is unfair. There are $(\mu + \lambda)$ evolutionary algorithms such as Genitor and CHC (described in Section 5) that are much more competitive with $(\mu + \lambda)$ evolution strategies. Evolution strategies are often used with population sizes as small as 5 to 20 individuals. This is also very different from canonical genetic algorithms.

4.1 Evolutionary Programming

The term *evolutionary programming* dates back to early work in the 1960's by L. Fogel [16]. In this work, evolutionary methods were applied to the evolution of finite state machines. Mutation operators were used to alternate finite state machines that were being evolved for specific tasks.

Evolutionary programming was dormant for many years, but the term was resurrected in the early 1990s. The new evolutionary programming, as reintroduced by D. Fogel, [15, 14, 13], is for all practical purposes, nearly identical to an evolution strategy. Mutation is done in a fashion that is more or less identical to that used in evolution strategies. A slightly different selection process (a form of Tournament Selection) is used than that normally used with evolution strategies, but this difference is not critical. Given that evolution strategies go back to the 1970's and predate the modern evolutionary programming methods by approximately 20 years, there appears to be no reason to see evolutionary programming as anything other than a minor variation on the well-established evolution strategy paradigm. Historically, however, evolution strategies were not well known outside of Germany until the early 1990's and evolutionary programming has now been widely promoted as one branch of Evolutionary Computation.

There are a couple of conceptual ideas that are closely associated with evolutionary programming. First, evolutionary programming does not use recombination and there is a general philosophical stance that recombination is unnecessary in evolutionary programming–and in evolutionary computation in general! The second idea is related to the first. Evolutionary programming is viewed as working in the *phenotype* space whereas genetic algorithms are seen as working in the *genotype* space. A philosophical tenant of evolutionary programming is that operators should act as directly as possible in the phenotype space to change the behavior of a system. Genetic algorithms, on the other hand, make changes to some encoding of a problem must be decoded and operationalized in order for behaviors to be observed and evaluated. Sometimes this (partially philosophical) distinction is clear in practice and sometimes it is not.

5 Two Other Evolutionary Algorithms

5.1 Genitor

Genitor [47, 50] was the first of what has been termed "steady-state" genetic algorithms [43]. 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 an example of a $(\mu + 1)$-ES in terms of its selection mechanism. Reproduction occurs one individual at a time. Two parents are selected for reproduction and produce an offspring that is immediately placed back into the population. Otherwise, the algorithm retains the flavor of a genetic algorithm. The worst individual in the population is deleted.

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. The population is maintained in a sorted data structure. Fitness is pre-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 [17] have shown that by replacing the worst member of the population, Genitor can generate much higher selective pressure than the canonical genetic algorithm.

In practice, steady-state genetic algorithms such as Genitor are often better optimizers than the canonical generational genetic algorithm. But this is somewhat of a comparison between apples and oranges, since the canonical generational genetic algorithm should be classified as a (μ, λ) evolutionary algorithm.

5.2 CHC

The CHC [12, 11] 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 also 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 uses restarts that employs 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" [12]. This is accomplished by only mating those string pairs which differ from each other by some number of bits (i.e., a mating threshold). The initial 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. If an offspring is closer to one parent or the other, it is more similar to that parent. If both the offspring and the similar parent make it into the next generation, this reduces diversity.

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 tuncation selection which guarantees the survival of the best strings) and to offset the aggressiveness of the search by using highly disruptive operators such as 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 [12, 11]. Other empirical experiments (c.f. [32, 46]) have shown that it is one of the most effective evolutionary algorithms for parameter optimization. Given the small population size, it seems unreasonable to think of an algorithm such as CHC as a "hyperplane sampling" genetic algorithm. It can be viewed as an agressive population based hill-climber.

6 Binary, Gray and Real-Coded Representations

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 ten years ago, is now considered to be controversial. On the other hand, the evolution strategies community [39, 40, 2] and more recently the evolutionary programming community [13] have emphasized the use of real-valued encodings. Application oriented researchers were also among the first in the genetic algorithms community to experiment with real-valued encodings [8, 26].

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 Standard Binary Reflected Gray

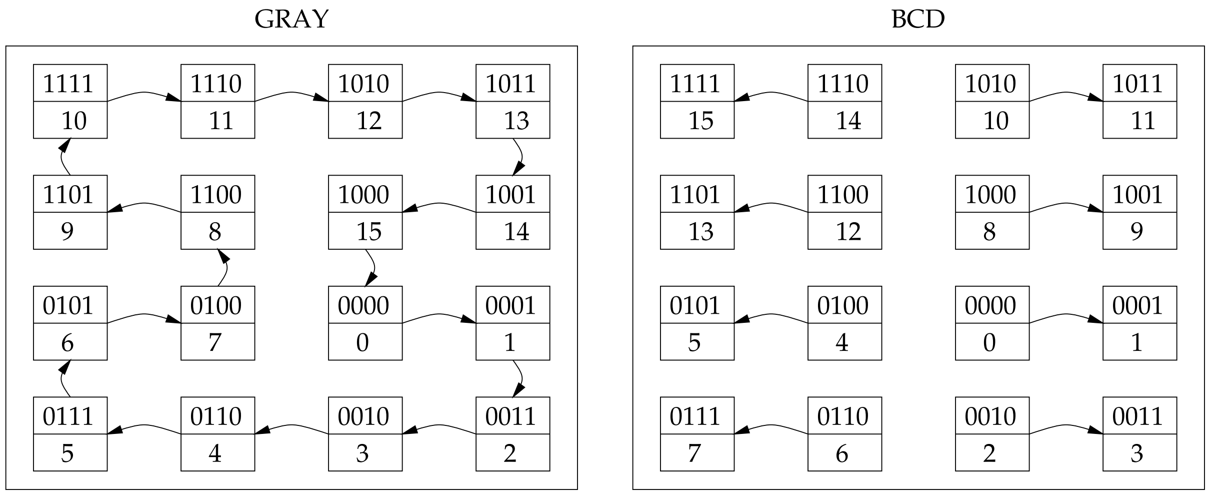


Figure 5: *Adjacency in 4-bit Hamming space for Gray and standard Binary encodings. The Binary representation destroys half of the connectivity of the original function.*

code [6]; but there are exponentially many possible Gray codes. A Gray code is a bit encoding where adjacent integers are also Hamming distance 1 neighbors in Hamming space.

Over all possible discrete functions that can be mapped onto bit strings, the space of all Gray codes and the space of all Binary representations are identical–this is another example of what has come to be known as a kind of "No Free Lunch" result [53, 36].

The empirical evidence suggests, however, that Gray codes are generally superior to Binary encodings. It has long been known that Gray codes remove Hamming Cliffs, where adjacent integers are represented by complementary bit strings: e.g., 7 and 8 encoded as 0111 and 1000. Whitley et al. [49] first made the rather simple observation that every Gray code must preserve the connectivity of the original real-valued functions. This is illustrated in Figure 5.

A consequence of the connectivity of the Gray code representation is that for *every* 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 guarantees. 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 (non-locally optimal) points near the optimum of that basin become new local optima under a Binary representation.

Whitley [45] has recently proven that Binary encodings work better than Gray codes on "worst case" problems; but this also means that Gray codes are better (on average) on all other problems. A "worst case" problem is a discrete function where half of all points in the search space are local optima. It is also simple to prove that for functions with a single optimum, Gray codes induce fewer optima than Binary codes. The theoretical and empirical evidence strongly indicates that for real-valued functions 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. In some cases real-valued encodings are better. Sometimes one also has to be careful to compare encodings using similar precision (e.g., 32 bits each). In other cases, a lower precision Gray code out-performs a real-valued encoding. There are no clear theoretical or empiricial answers to this question.

7 Genetic Programming

Genetic programming is very different from any of the algorithms reviewed so far. Genetic programming is not a parameter optimization method, but rather a form of automated programming. There have certainly been other applications of evolutionary algorithms that foreshadowed genetic programming. L. Fogel's early attempts to evolve finite state machines can be seen as a kind of programming (hence *evolutionary programming* has its roots in a form of programming). And in the 1980's, genetic algorithms were applied to evolving rule based systems such as classifier systems [18]. Steve Smith developed one of the first systems applying genetic search to variable length rule based systems in 1980 [41]. But genetic programming represents a major change in paradigm.

To start to understand genetic programming, it is perhaps best to look at a restricted example. Assume we are given the following function approximation task, where we wish to approximate a function of the form

$$F1 : x^3 - 2x^2 + 8.$$

Genetic programming is often implemented as a Lisp program. One Lisp program to implement function F1 is as follows.

```
(+ (* x (* x x)) (+ (* -2 (* x x)) 8))
```

There are several important things to note about this expression that are also important to genetic programming. First, there is a tree structure that directly corresponds to this program. Second, the tree structures are composed of substructures that are also trees and that are also syntactically correct self-contained expressions. Third, the expression itself is made up of functions that appear as internal nodes and terminals that appear as leaf nodes.

In genetic programming a structure such as

```
(+ (* x (* x x)) (+ (* -2 (* x x)) 8))
```

can directly be used as an artificial chromosome. A natural question is, "How can one recombine such structures?" Recombination directly swaps subtrees from different expressions. Figure 6 shows how two subtrees can be recombined to produce a tree that exactly computes $x^3 - 2x^2 + 8$.

Mutation can use used to change leaf nodes and to change internal nodes. (The arity of the operator must be handled in some way–either by restricting recombination to subtrees of the same arity or by defining functions to be meaningful over different arities.)

The other critical question is what set of functions and terminals should be used. The set of functions and terminals must be defined when creating the initial population and also

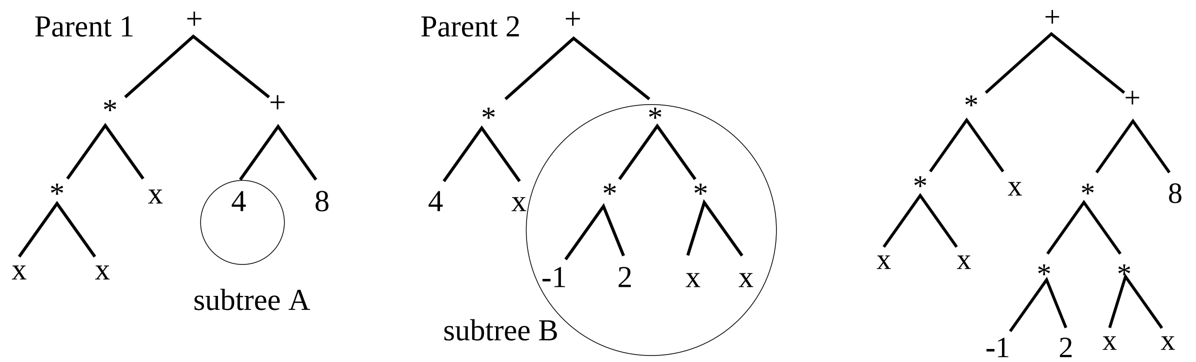


Figure 6: *Subtrees from Parent 1 and Parent 2 can be exchanged to produce a new tree. The rightmost tree is the offspring produced by taking the circled subtree of parent 2 and replacing the circled subtree in parent 1.*

when doing mutation. This is a somewhat critical question. Is it obvious what the set of terminals/functions should be? Does using a different set of terminals/functions change how difficult or easy the problem is to solve?

Another issue that arises when creating the initial population and also when doing recombination is the size of the trees that are generated. The larger the trees are allowed to be, the larger the search space becomes. Allowing trees to become too large can reduce the effectiveness of search, while making trees too small can limit the ability of genetic programming to find a solution. The depth of the trees in the initial population must obviously be limited to some maximum depth, and some similar limitation can be imposed during recombination.

Note again that we are attempting to find or approximate the following function:

$$F1 : x^3 - 2x^2 + 8.$$

Figure 7 shows two rough approximations to F1 given by

$$F2 : 30x^2 + 5000$$

$$F3 : 2000x - 9992.$$

The point of Figure 8 is that trees with forms similar to the target functions can also give rough approximations. It is also the case that the partial subtrees for these approximate solutions can be reconfigured by using recombination and mutation. Thus, it is possible to find other partial solutions that yield good approximations and eventually reconfigured to yield the desired results. As a result, the “fitness landscape” has some degree of smoothness. If one cannot find trees similar to the target that also yield approximate solutions, then it may be difficult to search the resulting program space.

There has been a very limited amount of theory developed to explain genetic programming. The theory that does exist has tried to explain genetic programming in terms of schema processing [35, 34, 33]. But while the field is perhaps short on having a strong theory, there have been some startling empirical successes. For example, Koza et al. [28] have used genetic programming to evolve circuit description programs. Genetic programming has

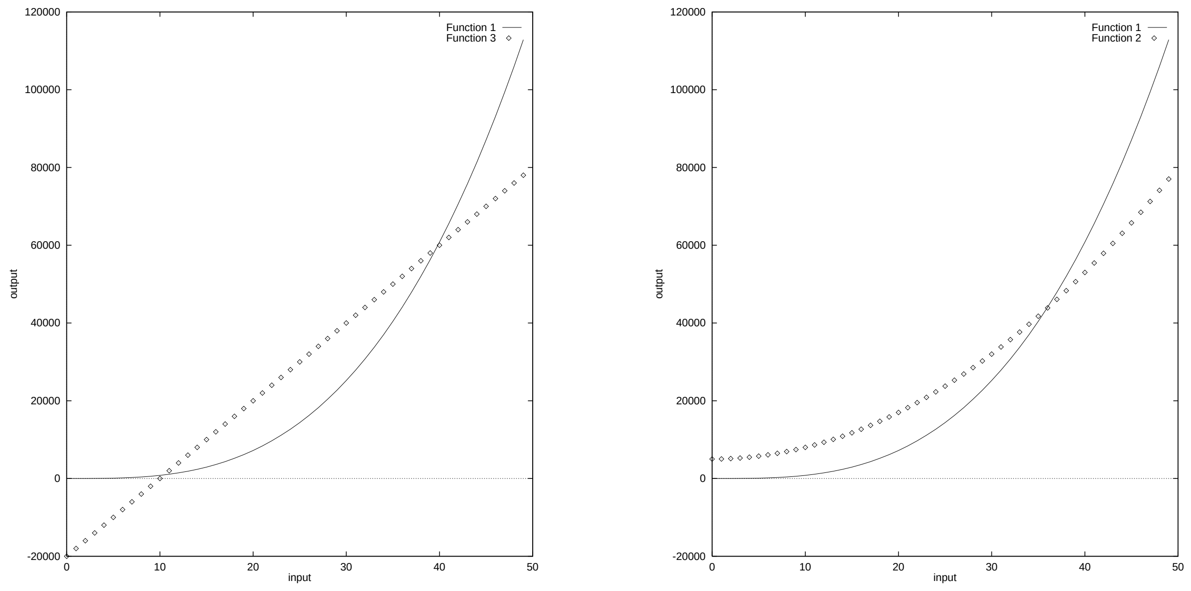


Figure 7: *The leftmost graph shows the linear approximation F3 plotted against the target F1. The rightmost graph shows the quadratic approximation F2 plotted against the target F1.*

been able to rediscover several patented circuits; in another case, genetic programming has been able to find circuits to accomplish a task that many electrical engineers thought was impossible [29].

In addition to being applied to Lisp programs, genetic programming has also been applied to other specialized languages. One such system is AIM-GP: Automatic Induction of Machine-code Genetic Programming. One of the advantages of AIM-GP is that this system can execute as much as 2 or 3 orders of magnitude faster than other genetic programming implementations because learning occurs at the machine code level.

The AIM-GP system represents individuals as machine code programs. AIM-GP uses C code operations to act directly on registers. This means that, in effect, AIM-GP generates a subset of C as its program output [5]. One can still constrain the operations on registers to produce effects similar or identical to higher level primitives often used in GP. For example, one might use a sequence of code to compute the Cosine of some value. In this case, a high level mutation could introduce this block of code or alter how it is applied.

8 Parallel Evolutionary Algorithms

Evolutionary Algorithms are easily parallelized. One of the simplest things that can be done is to evaluate the populuation in parallel. There have also been several mechanisms and selection strategies developed to support this type of parallelism. From a practical point of view, there is also another form of parallelism that is extremely easy to implement and that offers the potential to significantly improve search. This is the parallel *Island Model*.

An island model is a coarse grain parallel model. Assume we wish to use 64 processors and

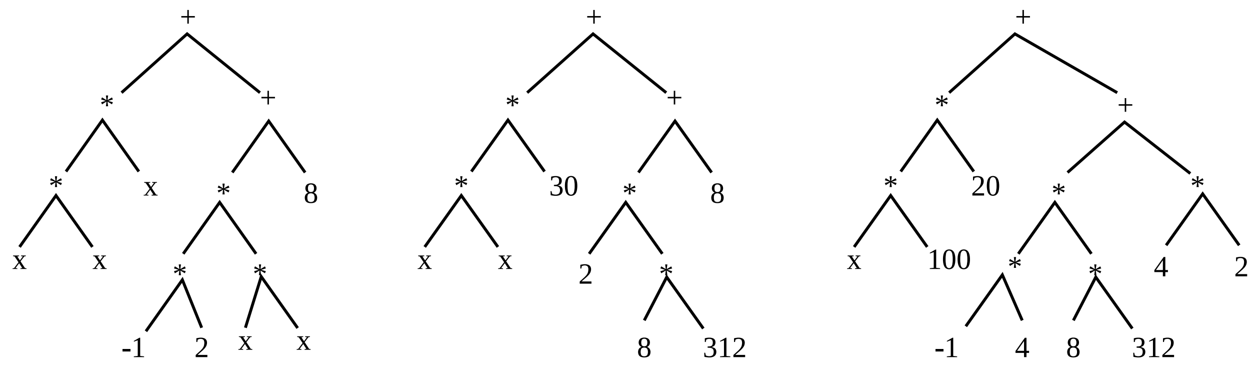


Figure 8: *Three different trees for functions F1, F2 and F3. Similarities between trees that approximate the target function areis an advantage when searching program space.*

6,400 strings. One way to do this is to break the total population down into 64 subpopulations of 100 strings each. Each one of these subpopulations could then execute as a normal evolutionary algorithm. It could be a canonical genetic algorithm, evolution strategy or Genitor. But occasionally, perhaps every five generations or so, the subpopulations would swap a few strings. This *migration* allows subpopulations to share genetic material [52, 20, 42, 44]. Note that the implementation cost is extremely minimal. This model can easily be implemented on a network of workstations and has very minimal communication costs since the migration of individuals between islands is limited.

The search in every subpopulation will be somewhat different since the initial populations will impose a certain sampling bias that will cause them to have a different trajectory through the search space. Thus, having different subpopulations acts as a means of maintaining and exploiting diversity in the overall population. By introducing migration, the Island Model is able to exploit differences in the various subpopulations. If a large number of strings migrate each generation, then global mixing occurs and local differences between islands will be driven out. If migration is too infrequent, it may not be enough to prevent each small subpopulation from prematurely converging.

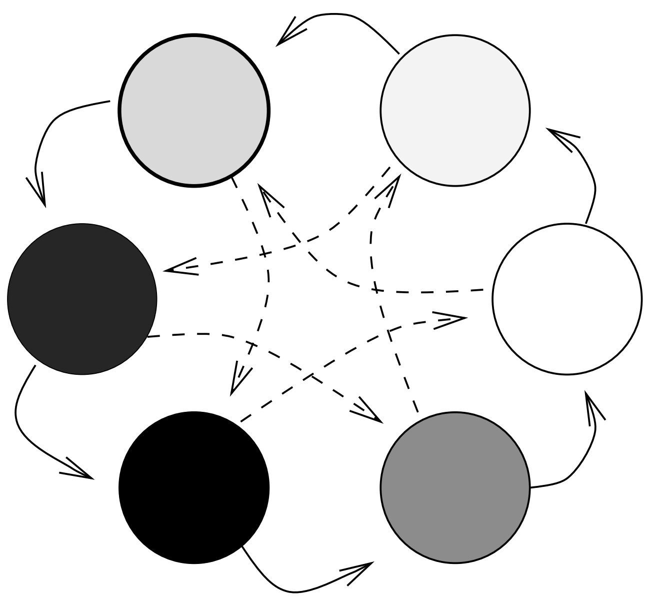
Running an Island Model on a single processor (without the parallelism) is often more effective than running a single population evolutionary algorithm with the same cumulative population size.

9 The Evaluation of Evolutionary Algorithms

When should an evolutionary algorithm be used? For example, when an optimization problem is encountered, when should one consider the use of an evolutionary algorithm?

Evolutionary algorithms are what are known as *weak methods* in the Artificial Intelligence community. Weak methods do not exploit domain specific knowledge. Evolutionary algorithms are also an example of what is known as a *blind search method.*

For many domains, there may be a good deal of domain specific knowledge. Methods that exploit domain knowledge will almost always out-perform methods that are blind. This leads



An Island Model Genetic Algorithm

Figure 9: *An example of an Island Model evolutionary algorithm. Migration is only allowed occassionally between the islands. The migration is typically between different islands at different points in time.*

to two observations: 1) if one has a domain specific method that exploits domain knowledge, *use it*; 2) if one is still interested in trying some form of evolutionary computation, try to add domain knowledge into the evolutionary algorithm.

One of the most simple tests to do *before* one attempts to apply an evolutionary algorithm is to try some form of local search. In local search, a neighborhood structure is defined around every point in the search space. Search then proceeds from a point and by testing all of the neighbors for an improving move. Any point where all of the neighbors are inferior is a *local optimum.*

An easy way to do local search is to apply a bit climber. This is especially true if a genetic algorithm is going to be used that also utilizes a bit encoding. In this case, the neighborhood is defined by flipping the L bits of the string representing the current point in the search space. This neighborhood is also known as the *Hamming Distance-1* neighborhood; the entire search space is then *Hamming Space.*

Dave Davis'algorithm called Random Bit Climbing (RBC) is a local search algorithm that climbs in Hamming Space [7]. A random permutation is generated that determines the order in which bits are flipped. Each improving move is accepted. After every bit has been tested, a new permutation is generated for the next pass. If RBC has checked every bit in the string and no improvement is found, a local optimum has been found and RBC is restarted from a new random point in the search space.

Other methods that might be used include such simple methods as forms of line search [49] and the Nelder Mead simplex methods (c.f. [40]). None of these methods requires gradient information. If gradient information is available, then some form of nonlinear gradient based search should be attempted. Whitley et al. [49] provide a more in-depth discussion of the evaluation of evolutionary algorithms for optimization and search problems.

In the case of genetic programming, finding some reasonable comparative method may or may not be simple. In the case of classification problems, neural networks may be a reasonable alternative to genetic programming. But in specialized domains, such as circuit design, it may not be easy to find an obvious method that can be easily compared against genetic programming. In some cases, it may be possible to use some form of local search.

10 Conclusions

There is a large body of literature covering evolutionary algorithms. Some topics not covered include the use of hybrid evolutionary algorithms that combine local search or some other heuristic search methods. Such methods can be used to improve the initial population or to improve each offspring that is produced. Evolutionary algorithms also have been applied with a good measure of success to scheduling and other combinatorial optimization problems. A special issue of the journal *Evolutionary Computation* [10] covers scheduling applications.

Major conferences in the area include the *Genetic and Evolutionary Computation Conference* (GECCO), *Parallel Problem Solving from Nature* (PPSN), and the *IEEE Congress on Evolutionary Computation.* Smaller high quality venues include the *Foundations of Genetic Algorithms* theory workshops and the *European Conference on Genetic Programming* (Euro-GP).

Acknowledgements

The author acknowledges the support of the Colorado Advanced Software Institute (CASI) and the Air Force Office of Scientific Research (AFOSR), Air Force Materiel Command, USAF, under grant number F49620-00-1-0144. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Thanks to Jon Rowe find Lisa Kennedy for reading a draft of this paper.

References

[1] A.E. Eiben and T. Bäck. Empirical Investigation of Multiparent Recombination Operators in Evolution Strategies. *Journal of Evolutionary Computation*, 5(3):345–365, 1997.

[2] T. Bäck. *Evolutionary Algorithms in Theory and Practice.* Oxford University Press, 1996.

[3] T. Bäck, F. Hoffmeister, and H.P. Schwefel. A Survey of Evolution Strategies. In L. Booker and R. Belew, editors, *Proc. of the 4th Int'l. Conf. on GAs*, pages 2–9. Morgan Kaufmann, 1991.

[4] James Baker. Reducing Bias and Inefficiency in the Selection Algorithm. In John Grefenstette, editor, *GAs and Their Applications: 2nd Int'l. Conf.*, pages 14–21. L. Erlbaum Assoc., 1987.

[5] W. Banzhaf, P. Nordin, R.E. Keller, and F.D. Francone. *Genetic Programming: An Introduction.* Morgan Kaufmann, San Francisco, CA, 1998.

[6] James R. Bitner, Gideon Ehrlich, and Edward M. Reingold. Efficient Generation of the Binary Reflected Gray Code and Its Applications. *Communications of the ACM*, 19(9):517–521, 1976.

[7] Lawrence Davis. Bit-Climbing, Representational Bias, and Test Suite Design. In L. Booker and R. Belew, editors, *Proc. of the 4th Int'l. Conf. on GAs*, pages 18–23. Morgan Kaufmann, 1991.

[8] Lawrence Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York, 1991.

[9] Ken DeJong. Genetic Algorithms are NOT Function Optimizers. In L. Darrell Whitley, editor, *FOGA - 2*, pages 5–17. Morgan Kaufmann, 1993.

[10] D. Montana (Guest Editor. Special Issue on Evolutionary Algorithms for Scheduling. *Evolutionary Computation*, 6(1), 1998.

[11] L. Eshelman and D. Schaffer. Preventing Premature Convergence in Genetic Algorithms by Preventing Incest. In L. Booker and R. Belew, editors, *Proc. of the 4th Int'l. Conf. on GAs.* Morgan Kaufmann, 1991.

[12] Larry Eshelman. The CHC Adaptive Search Algorithm. How to Have Safe Search When Engaging in Nontraditional Genetic Recombination. In G. Rawlins, editor, *FOGA -1*, pages 265–283. Morgan Kaufmann, 1991.

[13] D. B. Fogel. *Evolutionary Computation.* IEEE Press, 1995.

[14] D.B. Fogel. *Evolving Artificial Intelligence.* PhD thesis, University of California, San Diego, San Diego, CA, 1992.

[15] D.B. Fogel and W. Atmar. Comparing genetic operators with Gaussian mutation in simulated evolutionary processes using linear systems. *Biological Cybernetics*, 63:111–114, 1990.

[16] L.J. Fogel, Owens A.J., and M.J. Walsh. *Artificial Intelligence Through Simulated Evolution.* John Wiley, 1966.

[17] D. Goldberg and K. Deb. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In G. Rawlins, editor, *FOGA -1*, pages 69–93. Morgan Kaufmann, 1991.

[18] David Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[19] David Goldberg. A Note on Boltzmann Tournament Selection for Genetic Algorithms and Population-oriented Simulated Annealing. Technical Report Nb. 90003, Department of Engineering Mechanics, University of Alabama, 1990.

[20] Martina Gorges-Schleuter. Explicit Parallelism of Genetic Algorithms through Population Structures. In H.P. Schwefel and Reinhard Männer, editors, *Parallel Problem Solving from Nature*, pages 150–159. Springer/Verlag, 1991.

[21] R. Heckendorn, S. Rana, and D. Whitley. Polynomial time summary statistics for a generalization of MAXSAT. In *GECCO-99*, pages 281–288. Morgan Kaufmann, 1999.

[22] R. Heckendorn, S. Rana, and D. Whitley. Test Function Generators as Embedded Landscapes. In *Foundations of Genetic Algorithms FOGA-5*. Morgan Kaufmann, 1999.

[23] Robert B. Heckendorn, L. Darrell Whitley, and Soraya Rana. Nonlinearity, walsh coefficients, hyperplane ranking and the simple genetic algorithm. In *FOGA - 4*, San Diego, California, 1996.

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

[25] John H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, Cambridge, MA, second edition, 1992.

[26] J.D. Schaffer and L. Eshelman. Real-Coded Genetic Algorithms and Interval Schemata. In L. Darrell Whitley, editor, *FOGA - 2.* Morgan Kaufmann, 1993.

[27] J. Koza, D Goldberg, D. Fogel, and R. Riolo. *Genetic Programming 96: proceedings of the first annual conference.* MIT Press, Cambridge, MA, 1996.

[28] J. Koza, F.H. Bennet III, D. Andre, and M.A. Kleane. *Genetic Programming III: Darwinian Invention and Problem Solving.* Morgan Kaufmann, San Francisco, CA, 1999.

[29] J. Koza, F.H. Bennet III, W. Mydlowec, M.A. Kleane, J. Yu, and Oscar Stiffelman. Searching for the impossible using genetic programming. In *GECCO-99*, pages 1083–1091. Morgan Kaufmann, 1999.

[30] John Koza. *Genetic programming: A paradigm for genetically breeding computer population of computer programs to solve problems.* MIT Press, Cambridge, MA, 1992.

[31] K. Mathias, J.D. Schaffer, L. Eshelman, and M. Mani. The effects of control parameters and restarts on search stagnation in evolutionary programming. In G. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *ppsn5*, pages 398–407. Springer-Verlag, 1998.

[32] Keith E. Mathias and L. Darrell Whitley. Changing Representations During Search: A Comparative Study of Delta Coding. *Journal of Evolutionary Computation*, 2(3):249–278, 1994.

[33] U. O'Reilly and F. Oppacher. The troubling aspects of a building block hypothesis for genetic programming. In D. Whitley and M. Vose, editors, *FOGA - 3*, pages 73–88. Morgan Kaufmann, 1995.

[34] R. Poli. Exact Schema Theorem and Effective Fitness for GP with One Point Crossover. In *GECCO-00*, pages 469–476. Morgan Kaufmann, 2000.

[35] R. Poli and W. Langdon. Schema Theory for Genetic Programming with One-Point Crossover and Point Mutation. *Evolutionary Computation*, 6(3):231–252, 1998.

[36] N.J. Radcliffe and P.D. Surry. Fundamental limitations on search algorithms: Evolutionary computing in perspective. In J. van Leeuwen, editor, *Lecture Notes in Computer Science 1000*. Springer-Verlag, 1995.

[37] Soraya Rana, Robert Heckendorn, and Darrell Whitley. A tractable walsh analysis of sat and its implications for genetic algorithms. In *aaai98*, pages 392–397. MIT Press, 1998.

[38] J. D. Schaffer. Some Effects of Selection Procedures on Hyperplane Sampling by Genetic Algorithms. In Lawrence Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 89–130. Morgan Kaufmann, 1987.

[39] Hans-Paul Schwefel. *Numerical Optimization of Computer Models*. Wiley, 1981.

[40] Hans-Paul Schwefel. *Evolution and Optimum Seeking*. Wiley, 1995.

[41] S. Smith. *A learning systems based on genetic adaptive algorithms*. PhD thesis, University of Pittsburgh, 1980.

[42] Timothy Starkweather, L. Darrell Whitley, and Keith E. Mathias. Optimization Using Distributed Genetic Algorithms. In H.P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature*, pages 176–185. Springer/Verlag, 1990.

[43] Gilbert Syswerda. Uniform Crossover in Genetic Algorithms. In J. D. Schaffer, editor, *Proc. of the 3rd Int'l. Conf. on GAs*. Morgan Kaufmann, 1989.

[44] R. Tanese. Distributed Genetic Algorithms. In J. D. Schaffer, editor, *Proc. of the 3rd Int'l. Conf. on GAs*. Morgan Kaufmann, 1989.

[45] D. Whitley. A Free Lunch Proof for Gray versus Binary Encodings. In *GECCO-99*, pages 726–733. Morgan Kaufmann, 1999.

[46] D. Whitley, R. Beveridge, K. Mathias, and C. Graves. Test Driving Three 1995 Genetic Algorithms. *Journal of Heuristics*, 1:77–104, 1995.

[47] Darrell Whitley and Joan Kauth. GENITOR: A Different Genetic Algorithm. In *Proceedings of the 1988 Rocky Mountain Conference on Artificial Intelligence*, 1988.

[48] Darrell Whitley, Keith Mathias, and Larry Pyeatt. Hyperplane Ranking in Simple Genetic Algorithms. In L. Eshelman, editor, *Proc. of the 6th Int'l. Conf. on GAs.* Morgan Kaufmann, 1995.

[49] Darrell Whitley, Keith Mathias, Soraya Rana, and John Dzubera. Evaluating Evolutionary Algorithms. *Artificial Intelligence Journal*, 85:1–32, August 1996.

[50] L. Darrell Whitley. The GENITOR Algorithm and Selective Pressure: Why Rank Based Allocation of Reproductive Trials is Best. In J. D. Schaffer, editor, *Proc. of the 3rd Int'l. Conf. on GAs*, pages 116–121. Morgan Kaufmann, 1989.

[51] L. Darrell Whitley. A Genetic Algorithm Tutorial. *Statistics and Computing*, 4:65–85, 1994.

[52] L. Darrell Whitley and Timothy Starkweather. GENITOR II: A Distributed Genetic Algorithm. *Journal of Experimental and Theoretical Artificial Intelligence*, 2:189–214, 1990.

[53] David H. Wolpert and William G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute, July 1995.