*Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control.* James C. Spall

ISBN: 0-471-33052-3

# CHAPTER 9

# EVOLUTIONARY COMPUTATION I: GENETIC ALGORITHMS

This and the next chapter (Part II) focus on methods of evolutionary computation (EC). The role of evolution in the study of biology and, more generally, the life sciences and demographics is paramount. Essentially, evolution acts as a type of natural optimization process based on the conceptually simple operations of competition, reproduction, and mutation. The term EC (sometimes called *evolutionary algorithms*) refers to a class of stochastic search and optimization methods built on the mathematical emulation of natural evolution.

This chapter focuses on the most popular method of EC—genetic algorithms (GAs). The next chapter includes a summary of other evolutionary approaches, including some commentary on the distinctions between GAs and these other approaches. A warning is in order, however. The area of EC is undergoing rapid development with a concurrent lack of specific "industry standard" approaches. This makes the attempt to summarize the essence of this large and changing field in only two book chapters especially audacious, but let us push on regardless!

After some general remarks in Section 9.1, we summarize some of the history and motivating applications of GAs and other EC methods (Section 9.2). Then, in Section 9.3, we describe the coding process for the parameters being optimized, leading to the main evolution-like operations of the GA (Section 9.4) and the standard form of the algorithm in Section 9.5. Some modern extensions of the algorithm and practical suggestions (e.g., constraints) to enhance performance are presented in Section 9.6 and examples are given in Section 9.7. Concluding remarks are given in Section 9.8.

## 9.1 INTRODUCTION

Genetic algorithms (GAs) are the most popular of the evolutionary computation (EC) algorithms. Although GAs are general search algorithms, a leading (perhaps *the* leading) application is search as applied to function optimization.

The treatment here will reflect this application. Hence, we consider the familiar problem of minimizing $L(\boldsymbol{\theta})$ subject to $\boldsymbol{\theta} \in \Theta$. We continue in the spirit of Chapter 8 in focusing on the *global* optimization problem. Further, like several other methods considered thus far, the GA is not restricted to only continuous-variable problems. In particular, the elements of $\boldsymbol{\theta}$ can be real-, discrete-, or complex-valued. Whereas the annealing algorithms of Chapter 8 are based on analogies to the physical cooling of substances, the GA is based loosely on principles of natural evolution and survival of the fittest. In fact, in GA terminology, an equivalent *maximization* criterion, such as $-L(\boldsymbol{\theta})$ (or its analogue based on an encoded form of $\boldsymbol{\theta}$), is often referred to as the *fitness function* to emphasize the evolutionary concept of the fittest of a species having a greater likelihood of surviving and passing on its genetic material.

While the dominant use of GAs has been in optimization, it is worth mentioning at least some of the related applications. One is automatic programming (genetic programming), where the algorithm automatically adapts software to perform certain tasks (see Koza, 1992). Another application involves the use of GAs to study human social systems, where one might be interested in investigating the evolution of societies, including the impact of government policies, resource shortages, and human interaction with the environment. A GA might provide a simulation-based means for making policy recommendations. As stated in the preface of the 1992 update to the seminal Holland (1975), "GAs are a tool for investigating the phenomena generated by *complex adaptive systems*—a collective designation for nonlinear systems defined by interactions of large numbers of adaptive agents (economies, political systems, ecologies, immune systems, developing embryos, brains, and the like)." The goal in this chapter is more modest (but *not that* modest!) in focusing on the familiar mathematical problem of minimizing $L(\boldsymbol{\theta})$ (maximizing the fitness function).

A fundamental difference between GAs and all other algorithms considered up to now is that GAs work with a *population* of potential solutions to the problem. The previous algorithms worked with one candidate solution (say, $\hat{\boldsymbol{\theta}}_k$) and moved toward the optimum by updating this one estimate. GAs simultaneously consider multiple candidate solutions to the problem of minimizing $L$ and iterate by moving this population of candidate solutions toward (one hopes) a global optimum. The terms *iteration* and *generation* are used interchangeably to describe the process of transforming one population of solutions to another. If the GA is successful, the population of solutions will cluster at the global optimum after some number of iterations, as illustrated in Figure 9.1 for a population of size 12 with problem dimension $p = 2$.

One iteration of a GA involves obtaining (possibly noisy) loss function measurements (or fitness function values) at each of the candidate solutions in the population and then using this collection of loss measurements to move the population to a new set of candidate solutions. The population-based approach lends itself to parallel processing since any loss measurement for a candidate

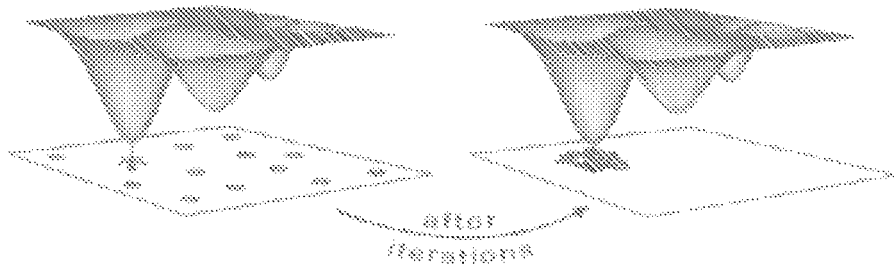


**Figure 9.1.** Minimization of multimodal loss function. Successful operations of a GA with a population of 12 candidate solutions clustering around the global minimum after some number of iterations (generations).

solution can be determined independently of the loss measurements for the other solutions. With this population-based structure in mind, the fundamental steps of a GA or other EC algorithm may be reduced to:

0. **Initialization.** Select an initial population size and values for the elements of the population; encode the elements of $\theta$ in a manner convenient for the algorithm operations (e.g., encode in bit form).
1. **Mixing.** Mix the current population elements according to algorithmic analogues of principles of evolution and produce a new set of population values.
2. **Evaluation.** Measure the performance of the new population values and determine if the algorithm should be terminated. Return to step 1 if further improvement is required and/or the budget of function evaluations has not been expended.

More detailed steps for the GA are given in Section 9.5, after the basic encoding and mixing operations are defined.

The use of a population versus a single solution fundamentally affects the range of practical problems that can be considered. In particular, the GA tends to be best suited to problems where the loss function evaluations are computer-based calculations such as complex function evaluations or simulations. This contrasts with the single-solution approaches discussed in the prior chapters, where the loss function (or possibly gradient) evaluations may represent computer-based calculations *or* physical experiments. Population-based approaches are not generally suited to working with real-time physical experiments because of the parallel structure of the algorithm and the fact that all of the function evaluations are assumed to come from the same generating mechanism. Implementing a GA with physical experiments requires that either there be multiple identical experimental setups (parallel processing) or that the single experimental apparatus be set to the same state prior to each population

member's loss evaluation (serial processing). These situations occur in relatively few practical settings.

As an example, consider the estimation of parameters associated with a feedback control mechanism. If there are many identical control systems available for training, then a population-based approach is feasible. On the other hand, if there is only one system available for training, then to obtain meaningful loss (or fitness) values it is expected that the system can be returned to the same state prior to running the experiment for each element of the population. This is not the usual situation given the inherent dynamic nature of the process. Note that some uses of GAs in control have been reported (e.g., Krishnakumar and Goldberg, 1992; Lennon and Passino, 1999). These approaches typically use the GA as an off-line mechanism for system identification (model estimation) or they replace physical experiments with computer-based simulations, hence putting them in the class of problems with computer-based loss evaluations for which population-based methods are well suited. Note also that nonpopulation methods (e.g., stochastic approximation) must also cope with the inability to return to the same state, but their nonparallel nature makes this somewhat easier (see, e.g., the control applications in Sections 3.2, 6.2, and 7.6).

Specific values of $\boldsymbol{\theta}$—often in some encoded form—are referred to as *chromosomes*. A chromosome is composed of genes, which play a role in the algorithm's iteration process in a way analogous to the biological role of genes in natural evolution. For our purposes , we consider a gene as being those encoded parts of the chromosome (typically, bits or groups of bits) that are associated with each element in $\boldsymbol{\theta}$. Hence a $p$-dimensional $\boldsymbol{\theta}$ has $p$ genes in the chromosome representation. (Some GA references also employ other biologically motivated terms such as allele, phenotype, genotype, and locus, but we will not need those terms here.) The central idea in a GA is to move a set (population) of chromosomes from an initial collection of values to a point where the fitness function is optimized. The population size (number of chromosomes in the population) will be taken as $N$.

Initial attention toward the GA came largely with those in the fields of computer science and artificial intelligence, but more recently interest has extended to essentially all branches of business, engineering, and science where search and optimization are of interest. To a greater extent than other methods in stochastic search and optimization, GAs are shrouded in a certain mystique. The widespread interest in GAs appears to be due to this mystique, to the success in solving some difficult optimization problems, and to the intriguing and sometimes quite surprising results that arise from the approximate computer-based emulation of natural evolution. Unfortunately, some interest also appears to be due to a regrettable amount of "hype" and exaggerated claims. While GAs (and more generally, EC) are important tools within stochastic optimization, there appears to be no formal evidence of consistently superior performance—relative to other appropriate types of stochastic algorithms—in any broad, identifiable class of problems.

## 9.2 SOME HISTORICAL PERSPECTIVE AND MOTIVATING APPLICATIONS

### 9.2.1 Brief History

There has been interest in the application of biological evolutionary principles to mathematical search and optimization since at least the 1950s, with several early papers bearing a loose resemblance to more recent methods of EC. The cornerstones of modern EC—evolution strategies, evolutionary programming, and GAs—were developed independently of each other in the 1960s and 1970s. Rechenberg (1965) introduced evolution strategies for optimization of continuous variables, with subsequent development by Schwefel (1977) and others. This approach has gained a considerable following. Evolutionary programming, as described in Fogel et al. (1966), treats the candidate solutions in the population as symbols via finite-state machines. In its original configuration, this approach was largely limited to small problems. Subsequent development by the Fogel father–son team (Lawrence and David Fogel) and others has strengthened the approach.

Chapter 10 provides a more detailed discussion of the contrasts between GAs and other EC approaches. We now mention a few of the historical highlights associated with GAs, directing the reader to De Jong (1988), Goldberg (1989, Chap. 4), Mitchell (1996, Chap. 1), Michalewicz (1996, pp. 1–10), and Fogel (2000, Chap. 3) for more complete historical discussions.

In the 1960s and 1970s, John Holland and his colleagues at the University of Michigan introduced GAs in the notation and formulation of today, leading to publication of the seminal monograph *Adaptation in Natural and Artificial Systems* (Holland, 1975). Although it was recognized from the outset that GAs were suited to optimization problems, Holland and his associates considered even broader issues. They developed GAs as algorithmic representations of complex adaptive processes, ranging from biological systems to economies to political systems. In fact, in the preface to the 1992 reprinting/update of the 1975 monograph, Holland states in reference to the 1975 monograph "About the only change I would make would be to put more emphasis on improvement and less on optimization."

Nevertheless, current applications for GAs appear to be overwhelmingly aimed at optimization. This and the next chapter reflect that emphasis. (And, of course, there may be little practical difference between "improvement" and optimization since optimization ultimately underlies the analysis and modeling associated with adaptation and learning.) The main difference is one of emphasis, where optimization often focuses on the question of convergence of an algorithm, while Holland's adaptive system emphasizes the intermediate behavior as learning is taking place.

On the heels of the original Holland monograph was the doctoral dissertation of one of Holland's students, De Jong (1975). Among the results in this dissertation were a large number of numerical test cases. De Jong

systematically studied the tuning process for GAs, establishing some reasonable ranges for some of the critical algorithm coefficients. These ranges continue to be cited today. One of the major innovations from the Holland team was the attempt to put GAs on a theoretically strong footing via the notion of schemas. This sometimes-controversial notion is discussed in Chapter 10. After the 1975 Holland monograph and De Jong dissertation, there was a relatively quiescent period of about a decade, with only a trickle of publications on GAs, most apparently by first- or second-generation students of Holland.

In the mid-1980s, activity picked up on GAs in parallel with activity in other dormant artificial intelligence concepts such as neural networks and fuzzy logic. The first large-scale conference devoted to GAs, the International Conference on Genetic Algorithms, was held in 1985 in Pittsburgh, Pennsylvania. The first full-fledged textbook on GAs, Goldberg (1989), was by one of Holland's most prolific students. This book is a lucid and accessible treatment of the subject based on information through the late 1980s. Much of the insight in this book continues to be relevant to the practical implementation of GAs (an update to parts of the 1989 book is Goldberg, 2002).

### 9.2.2 Early Motivation

The early interest in general adaptation and learning typically resulted in an optimization problem of some form. Holland (1975) and De Jong (1975) considered adaptation and learning problems in game playing, pattern recognition, and automatic programming, all areas remaining of great interest today despite the huge gains in algorithmic understanding and computational power. As an illustration of some of the issues that early GA researchers encountered, let us elaborate a bit on the problem of game playing.

The seemingly lighthearted task of building algorithms for game playing (checkers, chess, etc.) has led to many advances in artificial intelligence and adaptive control. Such problems have provided a good testbed for GAs and other EC algorithms as well as for machine learning methods such as the temporal difference algorithm (Chapter 11). Serious applications of game playing arise in areas such as law enforcement, strategic planning for the military, and economics. GAs were recognized in Holland (1975) and De Jong (1975) as appropriate algorithms for constructing game-playing strategies. One of the major difficulties in such tasks is the need to adapt to the strategy of an opponent. One typically will not know the opponent's strategy in advance, and hence the *adaptive* aspect of GAs for adjusting to an opponent's strategy—as revealed in the course of the game—is especially relevant.

The solution to game-playing problems typically runs into the same combinatorial problems seen with the traveling salesperson problem in Section 8.3. Even in the idealized case where one knows the opponent's strategy in advance, the evaluation of all possible solutions quickly gets out of hand. For example, there are $\alpha^m$ possible strategies that need to be tested in playing a game involving $m$ moves and $\alpha$ options at each move. Even a small game, say $m = 20$

and $a = 10$, would require over 3000 years of computation if each strategy can be evaluated in only $10^{-9}$ second. A typical chess game, of course, is much more complex than this (greater $m$ and $a$) and has the great complicating aspect of imperfect knowledge of an opponent's strategy.

The population-based aspect of GAs is consistent with a parallel evaluation of different strategies. The GA can consider a range of different strategies and combine aspects of successful strategies via its mixing operations (the mixing is referred to more formally as crossover). An intelligent evolution-based search can yield a good—and possibly optimal—solution with the explicit evaluations of only a fraction of the number of possible strategies.

## 9.3 CODING OF ELEMENTS FOR SEARCHING

### 9.3.1 Introduction

As mentioned in Section 9.1, an essential aspect of GAs is the encoding of $\theta$ for performing the GA operations and the associated decoding to return to the natural problem space in $\theta$. Usually, one works with a string representation to facilitate the operations of the GA (a string being a list of numbers). There are many ways that $\theta$ can be encoded in strings. Standard binary (0, 1) bit strings have historically been the most common (bit = *b*inary dig*it*). Other encoding methods include gray coding (which also uses (0, 1) strings, but differs in the way the bits are arranged) and multiple character encodings (> 2 elements in the string alphabet), including the full 10-character representation associated with the computer-based floating-point representation of the real numbers in $\theta$. The 10-character coding is often referred to as *real-number coding* since it operates as if working with $\theta$ directly. Based largely on successful numerical implementations, this natural representation of $\theta$ has grown more popular over time.

The issue of coding in GAs is an area of active research and there is some controversy regarding the prominent role assigned to bit-string encoding. Some references discussing alternative coding schemes include Davis (1996, Chap. 4), Michalewicz (1996, Chap. 5), Tang et al. (1996), Mitchell (1996, Sects. 5.2 and 5.3), Koehler et al (1998), and Fogel (2000, Sects. 3.5 and 4.3). In this section we first consider the standard bit coding and then consider the other two approaches mentioned above, gray coding and real-number coding.

### 9.3.2 Standard Bit Coding

The binary (0, 1) (i.e., bit) representation has traditionally been the most popular coding (Mitchell, 1996, p. 156; Davis, 1996, pp. 62–64). There appear to be several technical and other reasons for this popularity: (i) historical precedent, beginning with the seminal work of Holland (1975); (ii) relative ease of

implementation of the genetic operations described in Section 9.4; (iii) familiarity with binary (0, 1) manipulations, especially in light of their prominence in computer science where much of the work on GAs has taken place; (iv) rules of thumb that have developed for implementation under bit coding, including picking some of the important algorithm coefficients; and (v) theory associated with schemas suggesting that binary coding maximizes the amount of useful information being processed at each generation (versus other coding schemes). Items (i) through (iv) are largely self-explanatory. The schema theory mentioned in (v) is taken up in Section 10.3.

Because few definitions of $\boldsymbol{\theta}$ in practice are naturally in terms of bits, we now discuss the bit encoding ($\boldsymbol{\theta} \rightarrow$ bits) and decoding (bits $\rightarrow \boldsymbol{\theta}$) operations based on the elements of $\boldsymbol{\theta}$ being defined in terms more natural for the problem at hand. Of course, since one is always working with finite accuracy in solving an actual problem, a real-number definition for the elements of $\boldsymbol{\theta}$ is equivalent in practice to a discrete-valued definition.

Strictly speaking, the *encoding* process may not be needed in some applications since an implementation can be randomly initialized in bit space with all subsequent operations carried out in the bit space until the final decoding into $\boldsymbol{\theta}$. Encoding is, however, useful to understand as a guide to the relationship between $\boldsymbol{\theta}$ and the chromosomes in a GA. Further, encoding will be required for GA implementations when it is desirable to initialize certain elements in the population based on prior knowledge of good values of $\boldsymbol{\theta}$. Encoding can also be used to enforce constraints specified in the natural $\boldsymbol{\theta}$ space. The decoding operation is, of course, always required in converting a final GA solution to a physically meaningful $\boldsymbol{\theta}$.

We discuss one approach for both encoding and decoding below (many other approaches are possible). As is customary in GAs, let us assume that the individual elements of $\boldsymbol{\theta}$ are bounded above and below, and that the analyst knows the values of such bounds (hence, the constraint domain $\Theta$ is known to be a $p$-dimensional hypercube). Note that most of the other algorithms presented in this book have not had such a requirement (which is particularly relevant in conducting a fair comparison of algorithms). For ease of notation, we present the encoding/decoding procedures for a scalar $\theta$. Obviously, the same procedures apply to an individual element of a vector $\boldsymbol{\theta}$, in which case the procedures are associated with one gene in the chromosome. Following the procedure described below generally leads to a different number of bits representing each of the genes (corresponding to each of the elements in $\boldsymbol{\theta}$). This is standard in GA implementation.

The approach described below maps the minimum value of $\theta$ (scalar element) to [0 0 0 ... 0] and the maximum value to [1 1 1 ... 1]. A direct result of this desirable framework is that, in general, the mapping between a floating-point number and the bit representation is not one-to-one. In particular, a given floating-point number (to within the specified accuracy) may be represented by more than one bit representation (Exercise 9.4). In the description below, let $b$ be

the number of bits representing a scalar floating-point number (such as one element in a vector $\boldsymbol{\theta}$). Later we use $B$ to represent the number of bits in a chromosome, which encompasses the $p$ genes corresponding to all of the elements of $\boldsymbol{\theta}$; so $B \geq b$.

**Encoding (scalar $\theta$)**

1. Let $\theta_{\min}$ and $\theta_{\max}$ be such that $\theta_{\min} \leq \theta \leq \theta_{\max}$ and let $m$ represent the maximum number of positions after the decimal point that is relevant to the problem at hand ($m < 0$ denotes positions before the decimal). Choose a number of bits $b \geq 1$ such that $b$ is the smallest number satisfying $10^m(\theta_{\max} - \theta_{\min}) \leq 2^0 + 2^1 + 2^2 + \ldots + 2^{b-1} = 2^b - 1$, the maximum value in a standard nonnegative integer representation for a string of length $b$ bits.
2. Let $d = (\theta_{\max} - \theta_{\min})/(2^b - 1)$. The bit representation varies from [0 0 0...0] for $\theta_{\min}$ to [1 1 1...1] for $\theta_{\max}$. Each increase in $\theta$ by an amount $d$ increases the bit representation by one unit.
3. For the given $\theta$, calculate $round[(\theta - \theta_{\min})/d]$, where the operator $round[\cdot]$ rounds off the argument to the nearest integer. Represent the integer $round[(\theta - \theta_{\min})/d]$ using the standard binary representation $[a_1\ a_2\ \ldots\ a_b]$, where the elements $a_i$ are either 0 or 1.

**Decoding (scalar $\theta$)**

1. Assume a $b$-bit representation $[a_1\ a_2\ \ldots\ a_b]$ derived as in the steps above.
2. To within the specified accuracy ($m$), the value for $\theta$ is given by

$$\theta = \theta_{\min} + \frac{\theta_{\max} - \theta_{\min}}{2^b - 1} \sum_{i=1}^{b} a_i 2^{b-i} \ . \tag{9.1}$$

Let us now give an example of an encoding and a decoding for the elements of $\boldsymbol{\theta}$ when $p = 2$.

**Example 9.1—Encoding and decoding.** Let $\boldsymbol{\theta} = [t_1, t_2]^T \in [-4.00, 10.00] \times [1000, 4500]$ (so $t_1$ lies between –4.00 and 10.00 while $t_2$ lies between 1000 and 4500); $m = 2$ for the first element of $\boldsymbol{\theta}$ and $m = -2$ for the second element. For the first element, $b = 11$ since $2^{10} - 1 = 1023 \leq 10^m(\theta_{\max} - \theta_{\min}) = 10^2(10.00 - (-4.00)) \leq 2^{11} - 1 = 2047$. For the second element, we pick $b = 6$ since $2^5 - 1 = 31 \leq 10^{-2}(4500 - 1000) \leq 2^6 - 1 = 63$. Now if $\boldsymbol{\theta} = [-2.31, 4300]^T$, an encoding would be [0 0 0 1 1 1 1 0 1 1 1; 1 1 1 0 1 1], where the semicolon separates the genes for the two elements of $\boldsymbol{\theta}$. So, for example, the gene corresponding to the first element of $\boldsymbol{\theta}$ has $d = 0.00684$ and an integer value $round[(-2.31 - (-4.00))/d] = 247 = 2^0 + 2^1 + 2^2 + 2^4 + 2^5 + 2^6 + 2^7$. Applying the decoding formula (9.1), we recover the values of $\boldsymbol{\theta}$: –4.00 +

$(10.00-(-4.00))(2^0+2^1+2^2+2^4+2^5+2^6+2^7)/(2^{11}-1) = -2.311$ and $1000 + (4500-1000)(2^0+2^1+2^3+2^4+2^5)/(2^6-1) = 4277.8$. To the specified levels of accuracy (expressed via $m$), these recovered values are identical to the values of interest, –2.31 and 4300. ❑

### 9.3.3 Gray Coding

Gray coding also uses the (0, 1) alphabet in its string representation. It differs from the standard bit coding above in that floating-point values that are adjacent to each other, where adjacent is relative to the decimal accuracy chosen, vary by only one bit in the (0, 1) string. The primary motivation for gray coding is that adjacent floating-point values (i.e., the natural values in $\boldsymbol{\theta}$) may have very different representations when using the standard bit form above. This may cause the GA to need an undue number of iterations to move only a small distance in the natural problem space. For example, if a scalar $\theta$ is integer-valued, a move of one unit from $\theta = 7$ to $\theta = 8$ requires that all four bits in the representation [0 1 1 1] be changed. Given that the GA operates by flipping individual bits within the chromosome string (see Sections 9.4 and 9.5), the probability of simultaneously changing several bits to produce a small desired change is typically small.

A dual consequence of the requirement to change many bits to move a small amount in the space of floating-point numbers is the Hamming cliffs problem. Here, one finds that small changes to a binary code can cause very large changes in the elements of $\boldsymbol{\theta}$. For instance, in the simple four-bit example of the preceding paragraph, a change of only one bit from [0 0 0 0] to [1 0 0 0] causes the scalar $\theta$ to change over half its available range of [0, 15]. Although this may occasionally work to the algorithm's advantage (by forcing the search to cover a wider area and possibly jump away from local minima), it is generally accepted that Hamming cliffs introduce undesirable instability in the search process.

Gray coding is an alternative scheme that tends to make the bit representation more closely match the characteristics of the natural problem space. In gray coding, adjacent floating-point values differ by only one bit in the chromosome representation. Hence, it is expected that small changes in $\boldsymbol{\theta}$ can be achieved more easily. Further, a gray code tends to moderate (but not completely eliminate) the Hamming cliffs problem.

There is no unique gray code. In fact, Rana and Whitley (1997) conjecture that there are $O(b!2^b)$ different gray codes, where $b$ is the number of bits in the representation. One convenient procedure for translating standard binary coding to gray coding is described in Michalewicz (1996, p. 98). Table 9.1 shows the binary and gray codes for integers 0 through 10 using this translation procedure. Relative to the standard code, it is apparent that there is a more gradual change in the gray-coded representation as the integer representation changes. Unlike some of the changes in the standard code, each change in integer by one unit causes a change in the gray code of only one bit.

**Table 9.1.** Comparison of standard binary code and gray code.

| Integer | Standard Binary | Gray |
|---|---|---|
| 0 | 0 0 0 0 | 0 0 0 0 |
| 1 | 0 0 0 1 | 0 0 0 1 |
| 2 | 0 0 1 0 | 0 0 1 1 |
| 3 | 0 0 1 1 | 0 0 1 0 |
| 4 | 0 1 0 0 | 0 1 1 0 |
| 5 | 0 1 0 1 | 0 1 1 1 |
| 6 | 0 1 1 0 | 0 1 0 1 |
| 7 | 0 1 1 1 | 0 1 0 0 |
| 8 | 1 0 0 0 | 1 1 0 0 |
| 9 | 1 0 0 1 | 1 1 0 1 |
| 10 | 1 0 1 0 | 1 1 1 1 |

### 9.3.4 Real-Number Coding

We conclude this section with a discussion of direct real-number (floating-point) coding. Here, of course, there is no coding per se because we work directly with the floating-point-valued elements of $\boldsymbol{\theta}$ as implemented on a computer. So a convenient real-number coding scheme is to simply have each of the $N$ chromosomes in the population correspond directly to one $\boldsymbol{\theta}$ vector (hence each of the $p$ genes in the chromosome correspond to the floating-point number for one element of $\boldsymbol{\theta}$). This is the form of coding used in other population-based methods to be covered in Chapter 10 (evolution strategies and evolutionary programming). There is accumulating evidence that this natural form often offers performance equal or superior to that available with the binary-based coding methods described above.

Several of the advantages of the real-number implementation are: (i) the natural physical interpretation of the solution; (ii) the relative ease of handling constraints more general than the component-wise constraints described above for binary coding (see Section 9.6); (iii) the simplification of several implementation issues; and (iv) the growing number of numerical studies pointing to superior performance (e.g., Davis, 1996, Chap. 5; Michalewicz, 1996, Chap. 5; Salomon, 1996; and Fogel, 2000, Sect. 3.5). In addition to the above, some partial theoretical justification for real-number coding is given in Salomon (1996). This reference considers multimodal continuous $\boldsymbol{\theta}$ optimization problems with loss functions that are separable:

$$L(\boldsymbol{\theta}) = L_1(t_1) + L_2(t_2) + \ldots + L_p(t_p),$$

where $\theta = [t_1, t_2, \ldots, t_p]^T$ and the $L_i(\cdot)$ are some nonlinear functions. In the parlance of GAs (and biology), such separable loss functions are said to have no *epistasis* (epistasis refers to interaction among genes in a chromosome, tantamount here to interaction among the elements of $\theta$). For such $L$, real-number coding can reduce run times by a factor of $B^{b-1}$ (relative to standard bit coding), where $b$ is the same for all parameter elements $t_i$ and $B = pb$ is the length of the chromosome (number of bits). Salomon (1996) also suggests qualitatively similar behavior for nonseparable loss functions.

## 9.4 STANDARD GENETIC ALGORITHM OPERATIONS

Before presenting the steps of the algorithm, we must define the basic operations that go into the iterative process. For consistency with the GA literature, assume that $L(\theta)$ has been transformed to a fitness function with higher values being better. A common transformation is to simply set the fitness function to $-L(\theta) + C$, where $C \geq 0$ is a constant that ensures that the fitness function is nonnegative on $\Theta$ (nonnegativity is only required in some GA implementations). Hence, the operations are described for a *maximization* problem. (Section 9.6 includes some discussion about converting from loss functions to fitness functions in nonstandard cases where the simple transformation, fitness = $-L(\theta) + C$, is not sufficient.) Further, it is assumed that the fitness evaluations are noise-free. Unless otherwise noted, the operations below apply with any coding scheme for the chromosomes.

### 9.4.1 Selection and Elitism

After evaluating the fitness function for the current population of chromosomes, a subset of chromosomes is selected to use as parents for the succeeding generation. This operation is where the survival of the fittest principle arises, as the parents are chosen according to their fitness value. While the aim is to emphasize the fitter chromosomes in the selection process—so that the offspring tend to have even higher fitness—one must be careful about overdoing this emphasis. Too much priority given to the fittest chromosomes early in the optimization process tends to reduce the diversity needed for an adequate search of the domain of interest, possibly causing premature convergence and preventing the algorithm from uncovering chromosomes that are globally optimal.

Associated with the selection step is the optional "elitism" strategy (first described in De Jong, 1975, pp. 101–106), where the $N_e < N$ best chromosomes (as determined from their fitness evaluations) are placed directly into the next generation. This guarantees the preservation of the current $N_e$ best chromosomes. Note that the elitist chromosomes in the original population are also eligible for selection and subsequent recombination. De Jong's method appended the elitist

chromosomes to the current population, creating a temporary population of size larger than $N$. We use the more common method where the elitist elements are included *within* the $N$ chromosomes.

As with the coding operation for $\theta$, many schemes have been proposed for the selection process. We now describe one of the most popular methods—*roulette wheel selection* (also called *fitness proportionate selection*). In this selection method, the fitness functions must be nonnegative on $\Theta$. Other approaches are summarized in Section 9.6. In the roulette wheel approach, an individual's slice of a Monte Carlo-based roulette wheel is an area proportional to its fitness. The wheel is spun $N - N_e$ times and the parents are chosen based on where the pointer stops. For a given generation (iteration), this can be implemented in software as follows:

1. Sum the total of the $N$ fitness values; call this sum $S_f$.
2. Generate a uniformly distributed random variable on the interval $[0, S_f]$ (recall that the fitness values are nonnegative).
3. Go through the population, returning the chromosome whose fitness added to the sum of the previous fitnesses is greater than or equal to the random number. Perform this roulette sampling $N - N_e$ times.

The following example illustrates roulette selection.

**Example 9.2—Roulette selection.** Table 9.2 depicts an example of the roulette selection for ten chromosomes with two chromosomes set aside as elite chromosomes ($N = 10$, $N_e = 2$). Part (a) of the table shows the fitness value and running total, leading to $S_f = 3.50$. Part (b) shows the $N - N_e = 8$ draws from the uniform [0, 3.50] distribution and the corresponding selected chromosomes.

**Table 9.2.** Roulette parent selection process.

**(a)** Example chromosomes and their fitness values when $N = 10$.

| Chromosome | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Fitness | 0.10 | 0.20 | 0.05 | 0.45 | 0.25 | 1.00 | 0.10 | 0.80 | 0.05 | 0.50 |
| Cumulative sum | 0.10 | 0.30 | 0.35 | 0.80 | 1.05 | 2.05 | 2.15 | 2.95 | 3.00 | 3.50 |

**(b)** Roulette choice for $N - N_e = 8$ parent chromosomes to form next generation. Elite chromosomes 6 and 8 are placed directly into the next population.

| Random no. | 0.34 | 2.96 | 0.86 | 3.38 | 2.27 | 1.33 | 1.72 | 0.36 |
|---|---|---|---|---|---|---|---|---|
| Selected chromosome | 3 | 9 | 5 | 10 | 8 | 6 | 6 | 4 |

Note that the chromosome with the greatest fitness value (chromosome 6) received multiple selection, reflecting its fitter status, whereas there are only two selections (3 and 9) from among the five lowest-rated chromosomes (1, 2, 3, 7, and 9). The individual parent pairs for forming the next generation of eight chromosomes to be added to the two elite chromosomes would be chromosomes 3 and 9, 5 and 10, 8 and 6, and 6 and 4 (note that it is possible in general for both parents to be the same chromosome). ❑

### 9.4.2 Crossover

The crossover operation creates offspring of the pairs of parents from the selection step. A crossover probability, say $P_c$, is used to determine if the offspring will represent a blend of the chromosomes of the parents. If no crossover takes place, then the two offspring are clones of the two parents. If crossover does take place, then the two offspring are produced according to an interchange of parts of the chromosome structure of the two parents. Figure 9.2 illustrates this for the case of a nine-bit representation of the chromosomes. Case A shows one-point crossover, where the bits appearing after one randomly chosen dividing point in the chromosome are interchanged. Case B shows two-point crossover, where only the middle section is interchanged. In general, one can have a number of splice points up to the number of bits minus one, $B - 1$.

Note that the crossover operator also applies directly with real-number coding since there is nothing intimately connected to binary coding in single- or multi-point crossover. All that is required are two lists of compatible symbols. For example, one-point crossover applied to the chromosomes ($\theta$ values) [2.1, –7.4, 4.0, 3.9 | 6.2, –1.5] and [–3.8, 3.3, 9.2, –0.6 | 8.4, –5.1] yields the two children: [2.1, –7.4, 4.0, 3.9, 8.4, –5.1] and [–3.8, 3.3, 9.2, –0.6, 6.2, –1.5]. Another possible crossover operator under real-number coding is to average the two parents, yielding one child (Davis, 1996, pp. 66–69, uses this with success).

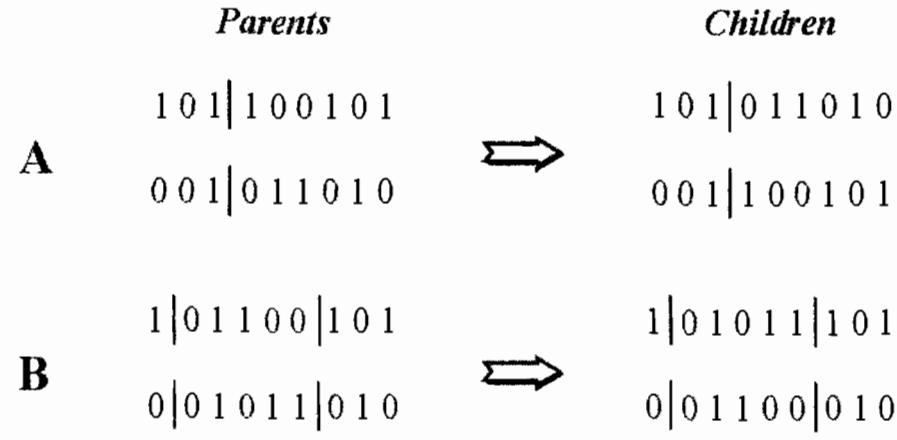


**Figure 9.2.** Crossover operator under bit coding. Case A shows one splice point; case B shows two splice points.

### 9.4.3 Mutation and Termination

Since the initial population may not contain encoded information rich enough to find the solution via crossover operations alone, the GA also uses a mutation operator where the chromosomes are randomly changed. For the binary coding, the mutation is usually done on a bit-by-bit basis where a chosen bit is flipped from 0 to 1, or vice versa. Mutation of a given bit only occurs with small probability $P_m$ to preserve the good chromosomes created through crossover. Figure 9.3 depicts a mutation in the third bit for the first child chromosome appearing in Figure 9.2.

Real-number coding requires a fundamental change in the mutation operator above. This follows from the loss of a uniqueness property associated with binary coding. With a (0, 1)-based coding, an opposite is uniquely defined, but with a real number, there is no clearly defined opposite (e.g., it does not make sense to "flip" the 7.63 element). One type of mutation operator is simply to add small independent normal (or other) random vectors to each of the chromosomes (the $\boldsymbol{\theta}$ values) in the population (see, e.g., Salomon, 1996). In a manner more closely corresponding to the bit-based mutation above, some real-number mutation operators work on the individual elements in each of the $\boldsymbol{\theta}$, changing a given element if a particular probability threshold is crossed (e.g., Michalewicz, 1996, p. 103; Davis, 1996, pp. 66–69). There appears to be little evidence that these per-element methods are superior to the simpler approach of adding a random vector to the $\boldsymbol{\theta}$.

As discussed in Section 1.2, there is no easy method for automatically stopping most (all?) stochastic search and optimization algorithms with a guarantee of being close to an optimum. GAs are no exception. The one obvious means of stopping a GA is to end the search when a budget of fitness (equivalently, loss) function evaluations has been spent. Alternatively, termination may be performed heuristically based on subjective and objective impressions about convergence. In the case where noise-free fitness measurements are available, criteria based on fitness evaluations may be most useful.

For example, suppose that $F^*$ and $F_*$ represent the maximum and minimum fitness values over the $N$ population values within a generation. Then a criterion suggested in Schwefel (1995, p. 145), which is based on the population elements yielding nearly the same fitness value, is to stop the search when $|F^* - F_*| \leq \eta$ for some $\eta > 0$. Alternative criteria based on the decoded population elements themselves (i.e., the $\boldsymbol{\theta}$ values) may be useful, especially if the fitness measurements are only available with noise (where it would be too

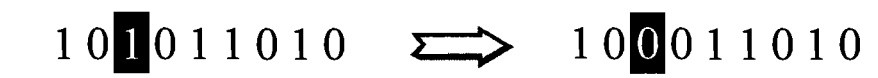


**Figure 9.3.** Mutation operator affecting one bit in a binary coding.

costly to use averaging to approximate the values $F^*$ and $F_*$ at each iteration). For example, terminate if the normed difference between all decoded population elements within one generation is less than some $\eta > 0$, or terminate if the best $\boldsymbol{\theta}$ value in the population changes negligibly over several generations.

## 9.5 OVERVIEW OF BASIC GA SEARCH APPROACH

There are *many* variations of the GA, making it difficult to present one "standard" form. Nevertheless, this section presents the steps of a basic form of the algorithm. These are essentially the steps described in Holland (1975, Chap. 6) with the following two exceptions. First, we do not include the inversion operator due to its relative lack of effectiveness and resulting lack of application in modern GAs (see, e.g., Davis, 1996, p. 21; Mitchell, 1996, p. 161). Second, we include the elitist strategy described in Subsection 9.4.1 as part of the basic GA (elitism did not appear in Holland, 1975). Section 9.6 discusses some popular enhancements to the basic algorithm.

### Core GA Steps for Noise-Free Fitness Evaluations

**Step 0** **(Initialization)** Randomly generate an initial population of $N$ chromosomes and evaluate the fitness function (the conversion of $L(\boldsymbol{\theta})$ to a function to be maximized for the encoded version of $\boldsymbol{\theta}$) for each of the chromosomes.

**Step 1** **(Parent selection)** Set $N_e = 0$ if elitism strategy is not used; $0 < N_e < N$ otherwise. Select with replacement $N - N_e$ parents from the full population (including the $N_e$ elitist elements). The parents are selected according to their fitness, with those chromosomes having a higher fitness value being selected more often.

**Step 2** **(Crossover)** For each pair of parents identified in step 1, perform crossover on the parents at a randomly (perhaps uniformly) chosen splice point (or *points* if using multi-point crossover) with probability $P_c$. If no crossover takes place (probability $1 - P_c$), then form two offspring that are exact copies (clones) of the two parents.

**Step 3** **(Replacement and mutation)** While retaining the $N_e$ best chromosomes from the previous generation, replace the remaining $N - N_e$ chromosomes with the current population of offspring from step 2. For the bit-based implementations, mutate the individual bits with probability $P_m$; for real coded implementations, use an alternative form of "small" modification (in either case, one has the option of choosing whether to make the $N_e$ elitist chromosomes candidates for mutation).

**Step 4** **(Fitness and end test)** Compute the fitness values for the new population of $N$ chromosomes. Terminate the algorithm if the stopping criterion is met or if the budget of fitness function evaluations is exhausted; else return to step 1.

The above steps are general enough to govern many (perhaps most) implementations of GAs. Many details remain to be specified before one can actually implement a GA in software. Several versions of the GA are available at the book's Web site. The performance of a GA typically depends greatly on the implementation details, just as we have seen with other stochastic optimization algorithms. Some of these practical implementation issues are taken up in the next section.

## 9.6 PRACTICAL GUIDANCE AND EXTENSIONS: COEFFICIENT VALUES, CONSTRAINTS, NOISY FITNESS EVALUATIONS, LOCAL SEARCH, AND PARENT SELECTION

As suggested above, the core steps in Section 9.5 leave much to be specified before implementation. Further, a straightforward implementation of the core algorithm is often insufficiently powerful to address serious, "industrial strength" problems. This section summarizes some of the modifications, enhancements, and tricks that have evolved to address implementation needs. It is obviously impossible to cover all but a sliver of these variations in this section, but we hope to convey the flavor of the kinds of variations possible. More detailed discussions are given in Mitchell (1996, Chap. 5), Michalewicz (1996, Chaps. 4–6), Davis (1996, Part I), Fogel (2000, Chaps. 3 and 4), Goldberg (2002, Chap. 12), and other references mentioned below.

As with other stochastic optimization methods, the choice of algorithm-specific coefficients has a significant impact on performance. In fact, with the GA, there are more coefficients to be set and decisions to be made than with any of the algorithms seen thus far. Included in the decisions and coefficients to be set to run a GA are: the choice of chromosome encoding, the population size ($N$), the probability distribution generating the initial population, the strategy for parent selection (roulette wheel or otherwise), the number of splice points in the crossover, the crossover probability ($P_c$), the mutation probability ($P_m$), the number of retained chromosomes in elitism ($N_e$), and some termination criterion. De Jong (1975, especially Chaps. 3 and 4) devoted a considerable fraction of his dissertation to exploring these issues, and developed broad guidelines for picking some of these quantities. These and other guidelines are reviewed in Mitchell (1996, pp. 175–177) and Weile and Michielssen (1997).

Obviously, there is not complete unanimity on the selection of the GA coefficients, but there is some consensus that the population size, crossover probability, and mutation rate for a standard bit implementation satisfy $20 \leq N \leq 100$, $0.60 \leq P_c \leq 0.95$, and $0.001 \leq P_m \leq 0.01$, respectively. Absent prior information suggesting other values, these ranges may provide reasonable starting points for an implementation. Ideally the values for these (and possibly other) GA coefficients should be allowed to evolve over the iterations. Davis (1996, Chap. 7) presents an adaptive method that changes some of the coefficient

settings over the course of a run. Of course, such adaptation complicates the algorithm and may not yield a benefit worth the complication.

Constraints on $L(\boldsymbol{\theta})$ (or the equivalent fitness function) and/or $\boldsymbol{\theta}$ are obviously of major importance in practical problems, as discussed many times earlier in this book. The bit-based implementation of GAs provides one natural (but limited) way of imposing component-wise lower and upper bounds on the elements of $\boldsymbol{\theta}$ (i.e., a hypercube constraint). More general approaches to handling constraints are discussed in Michalewicz (1996, Chap. 7 and Sects. 4.5 and 15.3). This reference shows how certain techniques in deterministic nonlinear programming can be extended to the GA context. Included in the suite of approaches are many variations of the penalty function method that was mentioned in Section 7.7 here. Real-number-based implementations of GAs are typically more adaptable to general constraints using techniques similar to those of nonlinear programming. A summary of available methods for handling constraints in GAs is given in Petridis et al. (1998) and Michalewicz and Fogel (2000, Chap. 9).

One of the major themes in this book has been optimization with noise in the loss function measurements. Unfortunately, there appears to be relatively little formal analysis of GAs in the presence of noise, although the application and testing of GAs in such cases was carried out as early as De Jong (1975, p. 203). Several of the references considering the problem are Rana et al. (1996), Nissen and Propach (1998), and Stroud (2001), all of which present numerical results from test cases. As with the search algorithms considered in earlier chapters, there is a fundamental tradeoff of (i) more accurate information for each function input (typically, via an averaging of the inputs) and fewer function inputs versus (ii) less accurate ("raw") information to the algorithm together with a greater number of inputs to the algorithm.

The above-mentioned references reveal performance characteristics that we have seen in other contexts with noisy loss measurements: namely, that short- and long-run results can be significantly degraded and that results can be surprising. GAs are sometimes claimed as being especially appropriate for noisy environments due to implicit averaging through the population-based structure. However, some such claims seem to ignore the additional cost per iteration (in loss evaluations) of a population-based method relative to the cost of a standard nonpopulation method (such as a random search algorithm [Subsection 2.2.2]).

There appears to be no rigorous comparison of GAs with other algorithms regarding relative robustness to noise. In fact, because of the discrete nature of certain decision steps in the algorithm, GAs may suffer relative to stochastic approximation-type algorithms that are designed for noise. That is, as with random search in Chapter 2 and simulated annealing in Chapter 8, noise fundamentally changes decisions made within the algorithm when the decisions are based on testing whether one fitness value is better than another. For example, the presence of nontrivial noise may alter parent selection and choices for elitism (see Exercise 9.3). Although there have been successful implementations of GAs with noisy measurements, these successes are problem

specific and/or involve noise with a very special structure (e.g., Baum et al., 2001). Regarding noise, Michalewicz and Fogel (2000, p. 325) state: "There really are no effective heuristics to guide the choices to be made that will work in general." Some numerical comparisons are given in Section 9.7.

One of the most powerful extensions of GAs is to strengthen their local search capabilities. In this way, one aims to cope with the common problem of a GA bogging down after doing a credible job of isolating the approximate location of a global optimum. Michalewicz (1996, Chap. 6) suggests a means by which the GA mutation operator can be modified to strengthen local search capabilities and provides the successful results of three numerical experiments on optimal control problems. As described in the following paragraph, an even easier and more powerful approach for continuous $\boldsymbol{\theta}$ problems may simply be to append a local optimization technique such as one of the classical deterministic methods mentioned in Chapter 1. Alternatively, one of the stochastic approximation methods of Chapters 4–7 can be used, especially if the loss (or gradient) measurements are only available in the presence of noise.

In particular, the GA can be turned off when it has stopped making significant progress toward a solution. Then, one of the local algorithms may be invoked using the GA solution (perhaps the mean of the population values) as the initial condition. These local algorithms are likely to be faster since they work directly (e.g., conjugate gradient or Robbins–Monro/stochastic gradient) or indirectly (e.g., simultaneous perturbation stochastic approximation [SPSA]) with the important gradient information that can dramatically speed convergence. An interesting hybrid form of GA/gradient descent is given in Blackmore et al. (1997). Their *congregational descent* algorithm is based on running a *population* of gradient descent algorithms. A significant benefit is that the differential equation theory discussed in Section 4.3 can be used to establish convergence properties of the algorithm. This includes theory for the *rate* of convergence.

Considerable effort has gone into alternative schemes for the selection of the chromosomes in the population that will produce offspring for the next generation. The essential aim is to favor those chromosomes producing better fitness values while preserving enough diversity to allow for an adequate search of the domain of interest $\Theta \subseteq \mathbb{R}^p$ to uncover possible hidden optima. The roulette wheel approach was discussed in Subsection 9.4.1. Part of the motivation for alternatives to the roulette wheel is to cope with problems resulting from large differences in the relative magnitudes of the fitness function, including possible premature convergence. This arises from the selection scheme putting too much emphasis on the better chromosomes encountered early in the iteration process.

One way of dealing with this is to rescale the fitness values so that they do not vary many orders of magnitude over the domain of interest (e.g., if $-L(\boldsymbol{\theta}) > 0$ for all $\boldsymbol{\theta} \in \Theta$ is the fitness value, then using $\log(-L(\boldsymbol{\theta}))$ instead of $-L(\boldsymbol{\theta})$ may be effective). Methods such as *rank selection* (Baker, 1985) and *tournament selection* (Goldberg and Deb, 1991) are also designed to help cope with possible premature convergence. In rank selection, only the ranks 1 to $N$ of the

chromosomes (instead of their fitness values) are used in a proportional selection scheme. Analogous to the median (versus the mean) in statistics, rank selection is less sensitive to extreme values and thereby less likely to be dominated by such values early in the iteration process.

Tournament selection is similar to rank selection in that fitness values are only used to determine whether one chromosome is better than another. It differs in that chromosomes are compared in a "tournament," with the better chromosome being more likely to win. The tournament process is continued by sampling (with replacement) from the original population until a full complement of parents has been chosen. The most common tournament method is the binary approach, where one selects two pairs of chromosomes and chooses the two parents according to which chromosome has the higher fitness in each of the two pairs. Example 9.3 illustrates the process and Examples 9.4 and 9.5 in Section 9.7 use this method with success in some numerical studies. Mitchell (1996, Sect. 5.4) provides a good survey of several other selection methods.

**Example 9.3—Binary tournament selection.** Let us use the example in Table 9.2. To pick the first parent, we uniformly generate two integers in {1, 2,...,10}, say 2 and 7. For the second parent, the same process yields (say) 6 and 7. Then using the fitness values in Table 9.2(a), we find parent 1 = chromosome 2 and parent 2 = chromosome 6. ❑

## 9.7 EXAMPLES

This section reports on three example implementations with GAs. The first two of these are numerical studies and the third is a conceptual problem. As is certainly clear to the reader by now, any set of numerical studies provides only limited insight into the general performance characteristics of an algorithm. Nevertheless, with the dearth of theory providing usable performance insight for GAs, simulation experiments seem to be the primary tool available for gaining insight into expected performance. This section attempts to use representative implementations of GAs on problems that span a range of the kind of problems encountered in practice. Obviously, countless other references contain numerical studies of GAs, including many of the citations in this chapter. There is no pretense that the numerical studies here represent a comprehensive evaluation of GAs. There are three versions of the GA used: two versions are bit-coded, one with roulette selection (Section 9.4) and one with tournament selection (Section 9.6), and the other version is real-coded with tournament selection. The GA studies here are conducted with MATLAB programs available at the book's Web site (**`GAbit_roulette`**, **`GAbit_tourney`**, and **`GAreal_tourney`**).

To be consistent with the remainder of this book, we describe the problems in terms of minimization for loss functions, even though the internal mechanics of the GAs are in terms of maximization of fitness functions. The examples here are representative of three important classes of optimization

problems: smooth (continuous) loss functions with one minimum, smooth loss functions with many minima, and nonsmooth (discrete) loss functions.

**Example 9.4—Skewed-quartic loss function.** Consider the skewed-quartic loss function in Example 6.6 (Section 6.7) with $p = 10$: $L(\boldsymbol{\theta}) = \boldsymbol{\theta}^T \boldsymbol{B}^T \boldsymbol{B}\boldsymbol{\theta} + 0.1\sum_{i=1}^{10}(\boldsymbol{B}\boldsymbol{\theta})_i^3 + 0.01\sum_{i=1}^{10}(\boldsymbol{B}\boldsymbol{\theta})_i^4$, where $(\cdot)_i$ represents the $i$th component of the argument vector $\boldsymbol{B}\boldsymbol{\theta}$, and $\boldsymbol{B}$ is such that $p\boldsymbol{B} = 10\boldsymbol{B}$ is an upper triangular matrix of 1's. This unimodal function has a high degree of parameter interaction (epistasis) and a high degree of skewness (the ratio of maximum to minimum eigenvalue of the Hessian at $\boldsymbol{\theta}^*$ is approximately 65). A single minimum occurs at $\boldsymbol{\theta}^* = \mathbf{0}$. The problem of determining a global minimum from among multiple local minima is not relevant here.

This study compares several implementations of GAs against SPSA (Chapter 7) in cases with both noise-free and noisy loss measurements. As discussed in Schwefel (1995, pp. 158–159), a difficulty in performing a fair comparison of GAs to many other algorithms (including basic SPSA) is that the GA may require more prior information to implement. In particular, with GAs, one specifies a hypercube containing the solution, information that may not be available to many other algorithms. To compensate for this advantage, we use a constrained version of SPSA.

The GAs are initialized with $N$ values of $\boldsymbol{\theta}$ uniformly distributed in the hypercube $\Theta = [-1.6383, 1.6384]^{10}$, with the endpoints chosen so that $m = 4$ and $b = 15$ (postdecimal accuracy and number of bits, respectively) for all elements of $\boldsymbol{\theta}$. The slightly peculiar endpoints are chosen so that each increment in the bit representation accounts for exactly 0.0001 in the corresponding element in $\boldsymbol{\theta}$. It is not obvious how one initializes a single-path search method (such as SPSA) to provide a fair comparison with a population-based method such as a GA. The approach here is to pick an initial condition for the single-path method that has the same mean distance to $\boldsymbol{\theta}^*$ as the elements in the initial population in a GA. In this vein, the SPSA solution is constrained to lie in $\Theta$ and the algorithm is initialized by picking $\hat{\boldsymbol{\theta}}_0$ such that the distance to $\boldsymbol{\theta}^*$ is the same as the mean distance of the GA population to $\boldsymbol{\theta}^*$ ($\hat{\boldsymbol{\theta}}_0 = 0.936 \times [1, 1, \ldots, 1]^T$ is used here).

Table 9.3 summarizes the results of the study based on 50 independent replications of optimization runs, each based on 2000 loss function measurements. All algorithm coefficients for the GAs and SPSA ($N$, $P_c$, $a_k$, etc.) are numerically tuned to achieve approximately optimal performance. Mutation for the real-coded GA is implemented by adding a small $p$-variate normal random vector to the nonelite members of the population (in the same general manner as Example 9.5 below). For the bit-based GAs, we used the formula in Exercise 9.6 to set the number of generations at the smallest number such that the total number of *expected unique* function evaluations is at least 2000. This accounts for the chromosomes retained through elitism and the fact that some offspring are identical to the parents. For the roulette selection implementation,

the fitness values are of the form $-L(\boldsymbol{\theta}) + C$, where $C$ is changed at each iteration to be the highest loss value in the population (so all fitness values at each iteration are nonnegative) and $\boldsymbol{\theta}$ is the decoded form of the bit representation. The tournament selection implementation simply uses fitness = $-L(\boldsymbol{\theta})$ since there is no requirement of nonnegativity.

Table 9.3 includes results for both noise-free and noisy loss measurements. In the latter, the noise in the loss function measurements at any value of $\boldsymbol{\theta}$ is given by $[\boldsymbol{\theta}^T, 1]\boldsymbol{z}$, where $\boldsymbol{z} \sim N(\boldsymbol{0}, \sigma^2 \boldsymbol{I}_{11})$ is independently generated at each $\boldsymbol{\theta}$. This relatively simple noise structure represents the usual scenario where the noise values in the loss measurements are dependent on $\boldsymbol{\theta}$; the last ($z_{11}$) term in $\boldsymbol{z}$ provides some degree of independence at each noise contribution and ensures that the loss measurements (i.e., the $y(\cdot)$) always contain noise of variance at least $\sigma^2$ (even if $\boldsymbol{\theta} = \boldsymbol{0}$). Approximate 90 percent confidence regions are shown below the loss values, computed according to the $t$-distribution method of Appendix B.

All algorithms show significant improvement relative to the loss values in the first iteration. For the noise-free case, the terminal GA solutions (corresponding to the values in the table) are the best loss values in the population at the last generation. For the noisy case, the GA solution is the population element having the lowest noisy measurement in the final population (the imperfect information resulting from the noise is not too damaging when the population has clustered around the solution). As usual, for evaluation purposes, the values in the table are the true (noise-free) loss values.

Table 9.3 indicates that the two bit-based GA forms are inferior to the real-coded form in the no-noise case; all the GAs are similar in the noisy case. SPSA outperforms the GAs in both the noise-free and noisy cases. Other studies

**Table 9.3.** Sample mean of terminal loss values for GAs and SPSA with skewed-quartic loss function; noise-free and noisy measurement cases; approximate 90 percent confidence intervals shown in [·]. Initial values for the GA populations include loss values both smaller and larger than $L(\hat{\boldsymbol{\theta}}_0) = 3.65$ for SPSA.

| Noise $\sigma$ | GA Bit-coded roulette selection | GA Bit-coded tournament selection | GA Real-coded tournament selection | Constrained SPSA |
|---|---|---|---|---|
| 0 | 0.0036 [0.0031, 0.0041] | 0.0031 [0.0026, 0.0036] | $8.5\times10^{-5}$ [$7.8\times10^{-5}$, $9.2\times10^{-5}$] | $2.7\times10^{-5}$ [$2.1\times10^{-5}$, $3.3\times10^{-5}$] |
| 0.1 | 0.065 [0.056, 0.074] | 0.052 [0.047, 0.057] | 0.057 [0.051, 0.063] | 0.017 [0.006, 0.027] |

of GA relative to SPSA have been conducted; for example, Nandi et al. (2001) show that the two methods perform comparably in a chemical process control application involving neural network training. ❑

**Example 9.5—Many-minima continuous problem.** Consider the loss function $L(\boldsymbol{\theta}) = t_1 \sin(4t_1) + 1.1 t_2 \sin(2t_2)$, $\boldsymbol{\theta} = [t_1, t_2]^T$. Let $\Theta = [0, 10]^2$. As shown in Figure 9.4, this function has many local minima, with one global minimum at $\boldsymbol{\theta}^* = [9.039, 8.668]^T$, corresponding to $L(\boldsymbol{\theta}^*) = -18.555$. Suppose that noise-free loss measurements are available. Let us compare stochastic approximation with injected randomness (Section 8.4) to a GA with real-number coding and tournament selection.

The SA algorithm with injected $N(\mathbf{0}, \boldsymbol{I}_{10})$ randomness is used in conjunction with the SPSA gradient approximation for the input $\boldsymbol{G}_k(\cdot)$. The algorithm uses tuned gains of form C in Section 8.4 (due to Fang et al., 1997): $a_k = a/(k+20)^{\alpha}$ and $b_k = b/[(k+1)^{\alpha/2} \log(k+1)]$. The specific parameter values in the gains are $a = 0.1$, $b = 5$, and $\alpha = 0.602$; the SPSA gradient estimate is formed with $c_k = c/(k+1)^{\gamma}$, $c = 0.001$, and $\gamma = 0.101$ (the indicated $\alpha$ and $\gamma$ are the standard values discussed in Section 7.5 and satisfy conditions C in Section 8.4). In contrast to the fixed initial condition in Example 9.4, the SA replications are initialized randomly (uniformly) within $\Theta$. The GA uses binary tournament selection with coefficients determined from manual tuning. The following settings work well: $N = 80$, $N_e = 2$, $P_c = 0.80$, with a mutation given by an independent, additive $N(\mathbf{0}, 0.05^2 \boldsymbol{I}_{10})$ perturbation applied to the $N - N_e = 78$ nonelite elements in the population (mutation is not applied to the $N_e$ elite

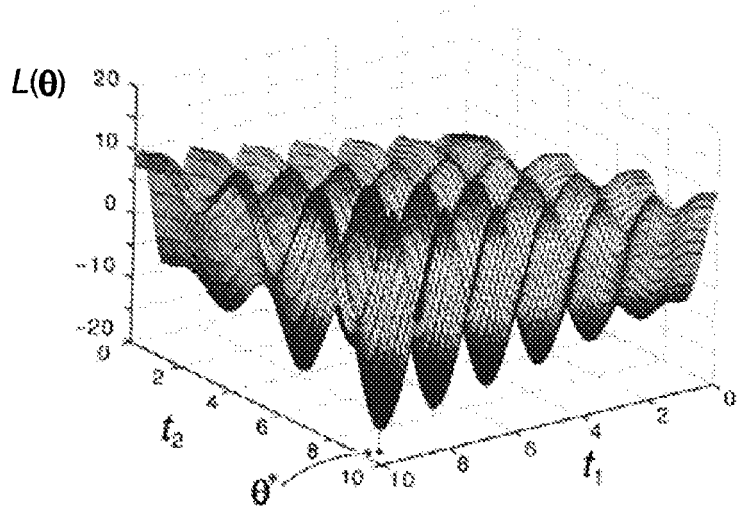


**Figure 9.4.** Multimodal loss function with unique global minimum at $\boldsymbol{\theta}^*$ near $[9.0, 8.7]^T$ (used in Example 9.5). The function is plotted over the domain $\Theta$.

elements). The initial population is distributed uniformly in $\Theta$. The fitness function is simply $-L(\boldsymbol{\theta})$.

The GA does well on this problem, significantly better than the SA algorithm. In 39 of 40 independent replications of 4000 loss measurements per replication, the GA produced a best final chromosome value having a loss value within 0.0005 of $L(\boldsymbol{\theta}^*)$ (the lone bad replication had a loss value greater than 1.5 units away from $L(\boldsymbol{\theta}^*)$). In contrast, in only two of 40 replications is the SA algorithm within 0.0005 of $L(\boldsymbol{\theta}^*)$ and in only five additional replications does the algorithm come within 0.01 of $L(\boldsymbol{\theta}^*)$. These seven "brushes" with $L(\boldsymbol{\theta}^*)$ (the values within 0.01 of $L(\boldsymbol{\theta}^*)$) generally come *before* the final iteration. That is, because there is no analogue of elitism with basic SA as implemented, six of these seven promising solutions are lost before the final iteration. (Of course, in practice, if one has a record of the iterates, it is possible to recover the best values in the iteration process.)[1] ❑

**Example 9.6—Traveling salesperson problem.** This famous problem is described in Section 8.3, including Example 8.1, where simulated annealing is discussed as a solution method. A number of authors have also pursued GA solutions to this problem. Let us summarize some of the issues associated with implementing a GA in the traveling salesperson problem. Numerical studies may be found in countless references, some of which are summarized in Michalewicz and Fogel (2000, Chap. 8). (Among the best numerical results cited is one where a GA finds a tour in a 532-city problem that has a cost within 0.06 percent of the known optimal cost.) To implement a GA, one needs to determine the appropriate coded representation for $\boldsymbol{\theta}$ and the appropriate operations (or analogies) for selection, crossover, and mutation. A central issue in GAs is implementing these steps in a way to maintain legal tours (recall that each city must be visited once and only once).

With respect to representing $\boldsymbol{\theta}$, binary codes are undesirable because of the ease with which they produce illegal tours. A number of possible nonbinary representations for $\boldsymbol{\theta}$ are discussed in Michalewicz and Fogel (2000, Sect. 8.1). The most natural is simply the ordered list of cities to be visited. For example, $\boldsymbol{\theta}$ = [10, 2, 28,...] says that city 10 is followed by city 2, which is followed by city 28, and so on. Once the $\boldsymbol{\theta}$ representation has been determined, there is nothing

[1] As an illustration of the dangers of drawing broad conclusions from numerical studies, a study with similar general characteristics is carried out in Maryak and Chin (2001). In particular, as in Example 9.5, $L(\boldsymbol{\theta})$ is continuous with many local minima and one global minima, and the loss measurements are noise-free ($L$ is the function used in Example 8.4). Here, SPSA without injected randomness is compared with a real-coded GA of the same general form in Example 9.5 (note that SPSA without injected randomness performed more poorly than SPSA with injected randomness in the problem of Example 9.5). In this study, SPSA achieved effective convergence in 10 of 10 replications using 2500 loss measurements per replication; a tuned GA achieved effective convergence in only one of 10 replications using 50,000 loss measurements per replication.

special about the structure of the traveling salesperson problem that requires altering the traditional selection methods (e.g., roulette or tournament) or elitism.

A standard crossover, such as described in Subsection 9.4.2, can easily lead to illegal solutions involving no—or multiple—visits to one city. As with the $\boldsymbol{\theta}$ representation, many proposals have been made for modified crossover operators that are guaranteed to provide valid tours (see the summary in Michalewicz and Fogel, 2000, Sect. 8.1). However, there does not appear to be any compelling reason to use one of these crossover operations in lieu of the simple operations described in Section 8.3 (inversion, translation, and switching). Rather, through use of a *single-parent* inversion operator, offspring can be created that are guaranteed to represent legal tours. In some numerical studies, this choice is shown to be more effective than several different crossover approaches (Michalewicz, 1996, Chap. 10). In particular, two cut points are randomly chosen in the chromosome and the ordering of the tour between these cut points is reversed. For example,

$$[3, 6, 11, 2 \,|\, 1, 9, 12, 5, 4 \,|\, 7, 10, 8]$$

is replaced by

$$[3, 6, 11, 2 \,|\, 4, 5, 12, 9, 1 \,|\, 7, 10, 8].$$

Finally, there are many options for mutation operators in traveling salesperson problems. One is to use mutation in conjunction with the switching operation in Section 8.3. In particular, mutation may be applied to each element (city) in $\boldsymbol{\theta}$ with probability $P_m$; if mutation is invoked, then the given element is interchanged with a randomly selected other element. ❑

## 9.8 CONCLUDING REMARKS

The genetic algorithms emphasized in this chapter are one important implementation of general evolutionary computation. GAs are based on an intriguing algorithmic analogue of the physical evolution of species in a population. The fundamental distinction between GAs and the methods considered in previous chapters is the use of a *population* of solutions. With a population, each algorithm iteration simultaneously updates a number of $\boldsymbol{\theta}$ values (or their bit-coded equivalent). This contrasts with the updating of a single $\boldsymbol{\theta}$.

Despite its intriguing form, a researcher or practitioner might ask whether a GA is going to help solve a problem better than other approaches. That is, with a comparable amount of effort in algorithm design and coding, will a GA be more efficient or otherwise outperform other methods we have studied? There is no easy answer to this question. Certainly, there have been many successful

implementations of GAs (and other evolutionary methods) to challenging practical problems. However, while the connection to natural evolution might suggest that the GA is somehow an optimized algorithm, this line of reasoning appears wrong on at least three counts: (i) There is no proof that natural evolution is an optimized process for species' development (it is difficult to see how a peacock's tail can be the result of an optimized process!); (ii) the no free lunch theorems (Subsection 1.2.2 and Section 10.6) guarantee that "on average" a GA can work no better than other algorithms; and (iii) many numerical studies exist on reasonable problems where other methods soundly outperform GAs.

Certainly, GAs are important tools in the collection of stochastic search and optimization methods. With their population-based structure, they are well suited to parallel processing. However, while there is some theory related to convergence and convergence rates (see Chapter 10), the theory is not as rich as for stochastic approximation and some other stochastic methods. This is unsurprising given the more complex structure of GAs (this relative complexity is evident by comparing the lengths of the GA codes and the codes for some other algorithms at the book's Web site). Further, the justification for the use of GAs with noisy measurements of the loss (fitness) function is thin. As with simulated annealing, the built-in binary decisions (e.g., whether or not to select a particular population element as a parent) suggest a possible lack of robustness to noise, although the redundancy in the population structure might ameliorate this effect at the cost of additional loss measurements.

In summary, GAs and other evolutionary methods provide an effective means for solving some difficult search and optimization problems, including some problems that vex other methods. Nonetheless, in contrast to the implications of some statements in the literature, GAs are not necessarily the only algorithms—or even the best algorithms—for solving certain difficult problems.

## EXERCISES

**9.1** Prove that the bit coding scheme in Section 9.3 yields the stated level of accuracy (i.e., that an encoding–decoding process on an arbitrary number is guaranteed to yield an accuracy neither greater than nor less than the $m$ digits specified).

**9.2** Let $\boldsymbol{\theta} \in \Theta = [-15.000, 15.000] \times [1.2000 \times 10^6, 2.4000 \times 10^6]$. Using the procedure of Subsection 9.3.2, determine the bit encoding for $\boldsymbol{\theta} = [-7.222, 2.1000 \times 10^6]^T$ when $m = 3$ for the first component and $m = -2$ for the second component. Apply the decoding process to recover $\boldsymbol{\theta}$ to within the specified accuracy. (Note: The zeros shown in $\Theta$ and $\boldsymbol{\theta}$ are consistent with the values of $m$.)

**9.3** Suppose that fitness values are observed only in the presence of independent $N(0, 1)$ noise and that $N = 15$ and $N_e = 1$. Suppose that one chromosome in the current population has a true (noise-free) fitness value of 11, while the

other 14 chromosomes have true fitness values of 10. Based on picking the chromosome with the highest noisy fitness measurement as the elite element, what is the probability of choosing one of the "wrong" chromosomes as the elite element for the next generation's population?

**9.4** Suppose that a scalar $\theta$ satisfies $-14 \leq \theta \leq 14$ and that $m = 3$ and $b = 15$. Compare the number of floating-point values to be encoded with the number of possible binary encodings using the procedure of Subsection 9.3.2. How many surplus encodings are there? Give one example of two encodings mapping to the same floating-point number.

**9.5** Using the numbers in Table 9.2(a), perform 500 Monte Carlo replications of (binary) tournament selection. What is the allocation of the resulting 1000 parents among the 10 chromosomes? Compare this allocation with the probabilities associated with choosing each of the chromosomes using roulette selection. Comment on any significant differences.

**9.6** The bit-based GA implementations in Section 9.7 were terminated when the expected number of unique fitness function evaluations first exceeded a specified value, say $n_{\text{fit}}$ (e.g., $n_{\text{fit}} = 2000$ in Example 9.4). Let *ident* represent the cumulative number of times (over all iterations) that a chromosome involved in crossover in the parent generation is identical to a chromosome in the offspring.[2] Assume that the probability of the mutation process returning any chromosome to a value previously obtained is negligible. As in Section 9.3, let $B$ represent the total number of bits in the chromosomes ($B = pb$ if there are $b$ bits per gene). For the bit-based forms of the GA, verify that the algorithm must perform at least

$$\frac{n_{\text{fit}} - N + E(ident)(1-P_m)^B}{(N-N_e)[P_c + (1-(1-P_m)^B)(1-P_c)]}$$

iterations to ensure that the expected number of function evaluations meets or exceeds $n_{\text{fit}}$. (The implementations in **`GAbit_roulette`** and **`GAbit_tourney`** at the book's Web site use the actual observed *ident* instead of $E(ident)$ for determining the number of generations to process to achieve an expected number of fitness evaluations of $n_{\text{fit}}$.)

**9.7** Given that a particular nonelite, binary-coded chromosome has been uniquely selected as a parent (i.e., there is one and only one copy of this chromosome among the parents in one generation), what is a (generally) nonzero lower bound to the probability of this chromosome passing intact to the next generation? (Ignore any negligible probability of mutating back to the

[2] So, if in one iteration (generation), two parents are identical through the selection process (guaranteed to produce two identical offspring) and in the next iteration another two selected parents are identical (usually different parents from the previous identical parents), then the total contribution toward *ident* for these two generations is four. The counter *ident* accumulates the identical parents over all iterations. Note that when two parents are not identical, the crossover probability provides an *upper bound* to the probability of producing two offspring different from the parents.

original chromosome after crossover.) Express the answer in terms of $P_c$, $P_m$, and $B$. Provide a specific calculation of this lower bound when $P_c = 0.70$, $P_m = 0.005$, and $B = 50$.

**9.8** For the $p = 10$ Rosenbrock function first considered in Example 2.5, implement a GA with bit coding and binary tournament selection. Assume that $\Theta$ is given by $[-2.048, 2.047]^{10}$ with $m = 3$ in the bit coding (the strange-looking endpoints yield no "excess" bit representations). Report the mean terminal loss value for 50 independent replications of 4000 loss (fitness) measurements at each replication (the terminal loss for one replication is the lowest loss value in the population at the last iteration). Also report the 90 percent confidence interval (based on the $t$-test of Appendix B) and the values for the GA parameters $N$, $P_c$, $P_m$, and $N_e$. Make some efforts to tune the GA coefficients to enhance performance.

**9.9** Assume the loss function and $\Theta$ as given in Exercise 9.8. As in Example 9.4, let the noise in the loss function measurements used as input to the algorithm be given by $[\boldsymbol{\theta}^T, 1]z$, where $z \sim N(\mathbf{0}, \sigma^2 \boldsymbol{I}_{11})$ is independently generated at each $\boldsymbol{\theta}$. Consider the three noise levels $\sigma = 0$, 0.1, and 2.0. At each noise level, perform 50 replications based on 4000 loss (fitness) function measurements for each replication. Use a GA with real coding and binary tournament selection. Make some efforts to tune the GA coefficients to enhance performance at each noise level. At each noise level, report the sample mean of the (noise-free) terminal loss values, the 90 percent confidence interval (based on the $t$-test of Appendix B), and the values for the GA parameters $N$, $P_c$, $P_m$, and $N_e$. If you did Exercise 9.8, contrast the results at $\sigma = 0$ here with the results for the GA with bit coding.

**9.10** Consider the method in Example 9.4 for initializing a single-path search method (such as stochastic approximation) to provide a fair comparison with a population-based method such as a GA. Consider the domain $\Theta$ and optimum $\boldsymbol{\theta}^*$ of Exercises 9.8 and 9.9.

**(a)** For a uniformly distributed initial GA population on $\Theta$, determine by Monte Carlo or analytical methods the mean (Euclidean) distance of the initial population to $\boldsymbol{\theta}^*$.

**(b)** Determine an initial condition of the form $[-c, 1, -c, \ldots, 1]^T$ that has the same distance to $\boldsymbol{\theta}^*$, where $c > 0$.

**(c)** Determine an alternative initial condition of the form $-[c, c, \ldots, c]^T$ that has the same distance to the solution, where $c > 0$.