CHAPTER 11

Evolutionary Optimization

Since early in the history of evolutionary computation, simulated evolution has been used as a search mechanism to find optimal solutions to problems of interest. In the 1950s, some used evolutionary methods to design computer programs that would control a robot [Friedman, 1956] or craft a simple computer program [Friedberg *et al*., 1958]. In the 1960s, others used evolutionary methods to design physical devices [Rechenberg, 1965] or create predictive models of time series data [Fogel *et al*., 1966]. Still others sought evolutionary methods for generating strategies in games [Reed *et al*., 1967; Fogel and Burgin, 1969]. The examples here are meant only to be illustrative and not exhaustive. There is a rich and interesting history to the field of evolutionary computation, as described in *Evolutionary Computation: The Fossil Record* [Fogel, 1998]; however, in virtually all of these cases, evolution was used as a foundation for searching a space of possible solutions to a problem using methods of random variation and selection.

In speaking about evolutionary algorithms used for optimization, or in other words *evolutionary optimization*, it is important to recognize that a problem to be solved must be well defined. This means that any possible solution to the problem must be comparable to another possible solution. It is not enough to identify merely the most desired outcome as "the solution." Often in the real world, we are not presented with sufficient resources or there are other constraints that prevent us from obtaining that "golden future." For a problem to be well defined, at least in the sense that is meant here, any possible solution to the problem must be assessable relative to all others.

Most often, the comparisons between two or more candidate solutions are based on quantitative measures of how well a proposed solution meets the needs of the problem. For example, you may have a problem of finding a minimum path from some point to another point separated by obstacles. Any proposed path that avoids the obstacles and gets from the first point to the other point has a certain distance. That distance provides a quantitative measure of the quality of the path. In this case, as in golf, smaller numbers are better. Other cases might be designing a schedule for a factory that maximizes profitability, or finding a new potential drug that binds to a specific protein with minimum free energy.

Fundamentals of Computational Intelligence: Neural Networks, Fuzzy Systems, and Evolutionary Computation, First Edition. James M. Keller, Derong Liu, and David B. Fogel.

Evolutionary algorithms are quite flexible, however, and it is not always the case that purely quantitative measures of a solution's "fitness" are required. For example, suppose that you may want to find a blend of coffee that you find most enjoyable from a mixture of five different types of coffee beans. You could try random combinations of blends and seek to evolve the overall blend that you like best. You could rate each blend on a numeric scale, say from 0 to 10 with 10 being the best coffee you've ever tasted. Then you could make a quantitative comparison between blends. (There's a problem here though because the best coffee that you've ever tasted might be surpassed by a new blend that is the new best coffee that you've ever tasted, and by our scoring system, each would receive a score of 10, unless you retaste and rescore coffees at each generation.) But instead you could simply rank the blends in order of preference, or in the most primitive case simply assert that you like one blend more than another. (See Herdy [1997] for an example of evolving coffee blends.)

The use of qualitative or even fuzzy descriptors of measures of fitness are most often found in what's called *interactive evolutionary computation*, in which a human provides a judgment about the quality of proposed solutions. Putting a human in the loop of assessing the merits of alternative solutions is typically slow relative to evolutionary algorithms that can compute a quantitative measure of a solution's quality. Nevertheless, there are certain applications that require a human's assessment, such as judging which type of music or art is preferred [Takagi, 2001], or the quality of a hearing aid or tinnitus masking device [Fogel, 2008].

This chapter focuses on quantitative evolutionary optimization, in which there is a numeric description—a function—that operates on a potential solution and returns either a single real number or multiple numbers that describe the value of the solution. Within evolutionary optimization, as with all engineering, there are essentially two forms of optimization problem.

One form is numeric. For example, find the point (x, y) such that $f(x, y) = x^2 + y^2$ is minimized, which we saw in Chapter 10. Here, the solution space is $\Re^2$ and $f(x, y)$ can be used as a measure of solution quality (lower is better because it's a minimization problem).

The other form is combinatoric. For example, given a collection of tools, each with a certain weight, find the combination that has the greatest combined weight that will fit in a bag that can hold only 25% of the weight of all the tools combined. This is called a *knapsack problem*. In this case, you aren't searching for a point in $\Re^n$, but rather for a combination of items that can be listed. In the case here, the order of the listing doesn't matter; the tools can go in the bag in any order. In other problems, the order of presentation of the items makes a big difference. For example, think about optimizing the arrival of supplies at a construction site.

This chapter provides an introduction to both numeric and combinatorial evolutionary optimization. It also describes some of the mathematical properties of representation and selection operators, and of evolutionary algorithms broadly. Some important extensions of the basic application of evolutionary algorithms for optimization are also covered, including handling constraints and allowing the evolutionary algorithm to learn how to optimize its own search parameters in a process called *self-adaptation*.

11.1 GLOBAL NUMERICAL OPTIMIZATION

11.1.1 A Canonical Example in One Dimension

In the previous chapter, we saw an example that used an evolutionary algorithm to address a simple problem in two dimensions. Let's briefly consider something even simpler first: the case of searching for a point $x \in \Re$, such that $f(x) = x^2$ is minimized. Since the function $f(x)$ is quadratic, finding the minimum by calculus methods is straightforward.[1] But suppose we didn't know that the function $f(x)$ was actually $f(x) = x^2$. Suppose it was just a "black box" that responded with a number any time we put a number in the box. We put the number 4 in the box and the box says 16. We put the number -4 in the box and again the box says 16. We need to find a number that minimizes what the box generates. This is called *black box optimization*.

An evolutionary approach to finding the minimum number could be as follows. Suppose we form a population of candidate solutions, $x_1, \ldots, x_\mu$, where there are μ "parents." We select these parents at random from a portion of real numbers, say uniformly between a lower limit of -100 and an upper limit of $+100$. In pseudocode, this would be as follows:

```
i = 0;
repeat
   i = i + 1;
   x[i]= U(−100,100);
until (i == μ);
```

where $U(-100, 100)$ is a uniformly distributed random variable over the interval $(-100,100)$.

Each of these parents is varied randomly to create more solutions, $x_{\mu+1}, \ldots, x_{\mu+\lambda}$, where there are λ "offspring."[2] We could conduct this random variation in many ways, but one typical method is to add a random number from a standard Gaussian distribution (i.e., a mean of zero and standard deviation of 1) to a parent to create an offspring.

For the sake of simplifying notation, assume that in this case $\lambda = \mu$ and thus each parent creates one offspring (although there is no limitation in evolutionary algorithms about the number of offspring that can be created from a parent). Then, in pseudocode, the process of creating offspring from parents would be as follows:

[1] Take the derivative $f'(x) = 2x$ and set it equal to zero and solve for $x\,(x = 0)$. Then take the second derivative $f''(x) = 2$ and note that it is positive, thus the point $x = 0$ is a minimum of $f(x)$.

[2] The notation of μ parents and λ offspring is standard nomenclature in evolutionary algorithms and developed within the offshoot of evolutionary computation known as evolution strategies that emerged in Germany in the 1960s. It's important not to be confused by the use of μ, which has a long history in statistics of describing the mean of a population or a random variable.

```
i = 0;
repeat
   i = i + 1;
   x[μ + 1] = x[i] + N(0,1);
until (i == μ);
```

where $N(0, 1)$ denotes a standard Gaussian random variable (also known as a "standard normal").

At this point, we have 2μ random ideas about what to put in the black box. We now have to test each idea and see what the box says. In pseudocode, this would be as follows:

```
i = 0;
repeat
   i = i + 1;
   score_x[i] = f(x[i]);
until (i == 2μ);
```

We then rank order the 2μ solutions in terms of their scores from lowest to highest. The μ best-ranking solutions then become the new parents for the next generation. In pseudocode, the process is as follows:

```
InitializePopulation;
repeat
   CreateOffspring;
   ScoreEveryone;
   SelectNewParents;
until (done);
```

The question of when to halt this procedure is often a matter of how much time is available to compute a solution or what level of quality is required. Here, we might continue the process of variation and selection for some number of generations g, or until the value of the best solution is lower than a threshold, say, 10^{-6}. This threshold would work in our problem if we had a hunch that zero was the minimum (which it is for $f(x) = x^2$).[3] If you construct a simple evolutionary algorithm for this problem, you'll find that it quickly locates solutions that have a score of less than 10^{-6}, but not as quickly as some other search methods, such as bisection or gradient search, and certainly not as quickly as calculus if we knew the function inside the black box ahead of time.

11.1.2 A Canonical Example in Two or More Dimensions

Having reviewed the canonical example in a single dimension, it is easy to extend it to the canonical case of two (or more dimensions), as we did in the example in

[3] Alternatively, a stopping rule could be to halt when improvement from one generation to the next is below a threshold, or improvement over several generations is below a threshold.

Chapter 10. Parents are chosen from the space $\mathfrak{R}^n$, where n is the number of dimensions. Offspring can be created by randomly varying each dimension of the parent, and also by combining or averaging across parents. Then, all the solutions are assessed and the best are retained to be parents of the next generation.

With regard to creating offspring from parents, traditionally, methods that use a single parent to create a single offspring are described under the heading of *mutation*, whereas methods that seek to combine multiple parents to create offspring are described with the term *recombination*. There are various forms of recombination that have their origins in inspiration from nature.

One method of recombination is called *crossover*. This operates on the following two solutions:

$$x_{11}, x_{12}, \ldots, x_{1n}$$

$$x_{21}, x_{22}, \ldots, x_{2n}$$

where x_{12} denotes the second parameter of the first solution and n is the number of parameters (dimensions). A crossover point is selected, usually at random, and two new solutions are created by splicing the first part of the first solution with the second part of the second solution, and vice versa. For example, suppose the crossover point was 3, then the two offspring would be

$$x_{11}, x_{12}, x_{23}, \ldots, x_{2n}$$

$$x_{21}, x_{22}, x_{13}, \ldots, x_{1n}$$

This "one-point" crossover operator has the sometimes undesirable property of forcing segments that are near each end of the solution vector to remain together. Thus, a multipoint[4] crossover operator can be employed, which treats the solution vectors more like rings in which sections can be exchanged, rather than strings in which a transition is made from one to the other. The limiting form of this, called *uniform crossover* [Fraser and Burnell, 1970; Syswerda, 1991], selects one component from either parent at random without regard to maintaining continuous segments and exchanges them.

Another form of recombination is *blending*. This averages parameters of parent solutions when creating offspring. For example, the two parents

$$x_{11}, x_{12}, \ldots, x_{1n}$$

$$x_{21}, x_{22}, \ldots, x_{2n}$$

could create

$$(x_{11} + x_{21})/2, (x_{12} + x_{22})/2, \ldots, (x_{1n} + x_{2n})/2$$

In general, there is no need to use a simple arithmetic mean; a weighted arithmetic mean or even a geometric mean may be useful in certain circumstances.

[4] This is called "n-point" in evolutionary algorithm literature, but here n refers to the number of crossing points not the dimension of the problem.

Recombination, in terms of both crossover and blending, can be extended to more than two parents. There are no restrictions that evolutionary operators follow the mechanics of the operations as found in nature.

It's almost always the case that evolutionary algorithms are employed when an optimization problem has multiple dimensions rather than just one dimension. In pseudocode, the process can be described as

```
InitializePopulation;
repeat
   CreateOffspring;     //mutate and/or recombine
   ScoreEveryone;
   SelectNewParents;
until (done);
```

11.1.3 Evolution versus Gradient Methods

If the problem at hand presents a smooth, convex, continuous landscape (e.g., $f(x, y) = x^2 + y^2$), then gradient or related methods of optimization will be faster in locating the single optimum point. On the other hand, if the problem presents a landscape with multiple local optima (e.g., $f(\mathbf{x}) = An + \sum_{i=1,\ldots,n} [x_i^2 - A\cos(2\pi x_i)]$, where A is a constant, see Figure 11.1), then gradient methods will likely fail to find the global

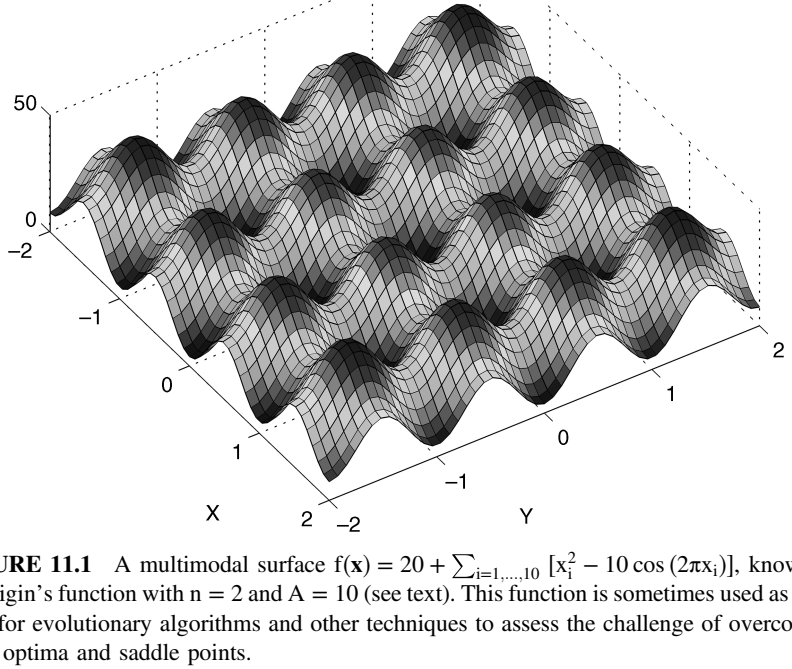

FIGURE 11.1 A multimodal surface $f(\mathbf{x}) = 20 + \sum_{i=1,\ldots,10} [x_i^2 - 10\cos(2\pi x_i)]$, known as Rastrigin's function with $n = 2$ and $A = 10$ (see text). This function is sometimes used as a test case for evolutionary algorithms and other techniques to assess the challenge of overcoming local optima and saddle points.

optimum solution because each gradient is associated only with the local optima created by the effects of the cosine function.[5] This may present an opportunity for evolutionary optimization to be used effectively. If the landscape is discontinuous and/or not smooth, then gradient-based approaches may be inapplicable—it may not be possible to compute gradients—and thus other "generate and test" methods such as evolutionary algorithms again may present an opportunity for addressing the problem successfully.

11.2 COMBINATORIAL OPTIMIZATION

To illustrate the use of evolutionary algorithms for combinatorial optimization, let's consider the canonical case of addressing the traveling salesman problem. The problem is as follows. There are n cities. The salesman starts at one of these cities and must visit each other city once and only once and then return home. The salesman wants to do this in the shortest distance. The problem then is to determine the best ordering of the cities to visit. This is a difficult problem because the total number of possible solutions increases as a factorial function of the number of cities. More precisely, for n cities, the total number of possible solutions is $(n-1)!/2$. For a small problem such as $n = 10$, there are 181,440 different paths to choose from. For $n = 100$, the number of different paths is on the order of 10^{150}. Figure 11.2 shows the relationship between the number of cities and the number of possible tours. By way of comparison, as mentioned earlier, there are about 10^{18} s in the approximately 13 billion year history of the universe. So, when there are so many possibilities to choose from, enumerating them all and determining the best one is infeasible.

The traveling salesman problem is NP-hard. There are no known methods for generating solutions in a time that scales as a polynomial function of the number of cities, n. There are heuristics that are available for this canonical form of the traveling salesman problem, and some of these can be effective even for large n given certain cases of the general problem but for the time being let's focus on an evolutionary approach to the problem.

First we must create a data structure to represent a solution. One such structure is a list of cities to be visited in order. For example, given the seven cities, a potential solution is

$$[1, 3, 2, 5, 7, 6, 4]$$

in which the salesman is presumed to start at city 1 and return to city 1 after visiting city 4. Other data structures are also possible. For example, since the salesman's circuit is a series of links between cities (more formally *edges* on a *graph*), the representation could be in the form of a series of links, such as

$$[(1, 3), (3, 2), (2, 5), (5, 7), (7, 6), (6, 4), (4, 1)]$$

[5] This function is known as Rastrigin's function and was introduced in evolutionary algorithm research many decades ago as a test case to determine how effective evolutionary methods could be on functions that present many local optima. At the time of this writing, there is a nice illustration of Rastrigin's function at en.wikipedia.org/wiki/Rastrigin_function.

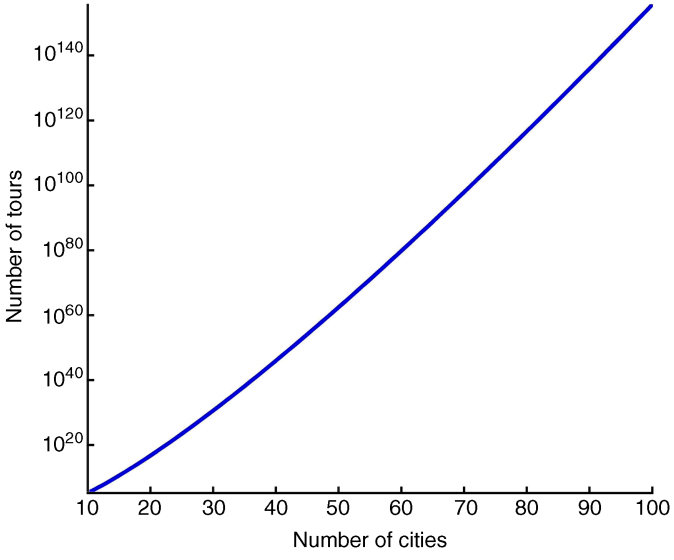

FIGURE 11.2 The number of possible tours in a traveling salesman problem as a function of the number of cities. There are $(n-1)!/2$ possible tours for n cities. The formula is derived noting that whenever you start, there are $(n-1)$ possible next cities, then $(n-2)$, then $(n-3)$, and so forth; however, it doesn't matter which direction you traverse the tour, thus the $(n-1)!$ options can be divided by 2. Problems with 70 cities are already close to presenting 10^{100} possible options.

Some people may find this more intuitive; however, since the ordered list of cities is quite straightforward, so for our purposes here, we'll focus on that representation.

Next, we must be able to score any potential solution. Here, the score associated with any solution, f, is the length of the total path through the cities:

$$f = \sum_{i=1}^{n-1} d(c_i, c_{i+1}) + d(c_n, c_1)$$

where $d(a, b)$ is the distance between city a and city b, and c_i is the ith city. Lower scores are better.

Then we must determine a method for creating offspring solutions from parent solutions. Given the representation of an ordered list, the following are some of the potential methods:

1. *Select and replace:* Choose a city at random along the list and replace it at a random place along the list.
2. *Invert:* Choose two cities along the list at random and invert the segment between those cities.
3. *Protect and randomize:* Choose a segment of the list to be passed from the parent to the offspring intact, and then randomize the remaining cities in the list.

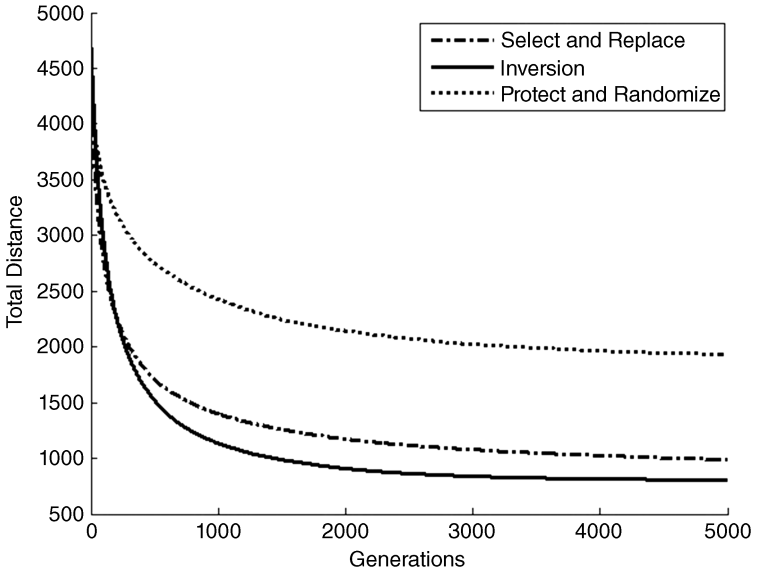

FIGURE 11.3 The best tour length for each average of 50 trials of a (50 + 50) evolutionary algorithm using permutation encoding on a 100-city traveling salesman problem as a function of the number of generations. The legend shows the results of different mutation operators: dotted = protect and randomize, dashed = select and replace, solid = invert a segment. At 5000 generations, the mean best tour lengths based on these mutation operators were 1932.6, 988.4, and 807.0 units, respectively.

Figure 11.3 shows a comparison of results for these three operators. Each curve represents the average score of the best solution in the population at the specified generation. The results suggest that the two-point inversion operator generates better solutions faster than the other two mutation operators on this problem. (We'll discuss how to use statistics to make conclusions about these sorts of comparisons in later examples.)

Let's consider as well a recombination operator we could try on the traveling salesman problem. One possibility is an operator called partially mapped crossover (PMX).[6] The operator works on two parents by choosing a segment of the first parent to move directly to the offspring. It then moves the feasible parts from the second parent to the offspring. Finally, it assigns the remaining values to the offspring based on the indexing in the first parent. Here's an example.

Suppose we have two parents

$$[3, 5, 1, 2, 7, 6, 8, 4]$$

[6] PMX is also called partially matched crossover but, as introduced in Goldberg and Lingle [1985], it was described as partially mapped crossover.

and

$$[1, 8, 5, 4, 3, 6, 2, 7]$$

and suppose that the segment [1, 2, 7] is selected to be saved from parent 1. Then, at this step the offspring could be written as

$$[+, +, 1, 2, 7, +, +, +]$$

where the + symbol in a placeholder for an undetermined component of the solution. Next, the feasible elements of parent 2 are copied to the offspring. These are the values that do not already appear in the offspring. So, the offspring becomes

$$[+, 8, 1, 2, 7, 6, +, +]$$

Finally, we look at the first position, which was a 1 in parent 2 but we can't copy that because 1 already appears in the offspring. So, we look at where 1 appears in parent 1 and see that the corresponding value for parent 2 in that position is 5. So, 5 goes in the first place in the offspring. Similarly, 4 goes in the seventh place and 3 goes in the eighth place, and we end up with

$$[5, 8, 1, 2, 7, 6, 4, 3]$$

Figure 11.4 shows the average results on 50 trials of applying the PMX operator on the same 100-city traveling salesman problem instead of the other mutation operators. It also shows the results of combining PMX with inversion, with the probability of having an inversion set at 0.5 per offspring. The results show that the PMX operator

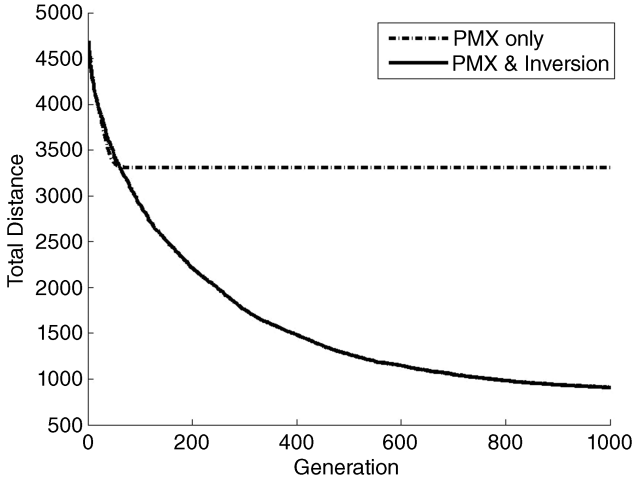

FIGURE 11.4 The average of 50 trials with the same 100-city traveling salesman problem as studied previously comparing the best tour as a function of the number of generations for PMX and PMX plus a 0.5 probability of applying inversion. On average, PMX alone stalls at a relatively poor solution. By including inversion, the search for improved solutions can continue.

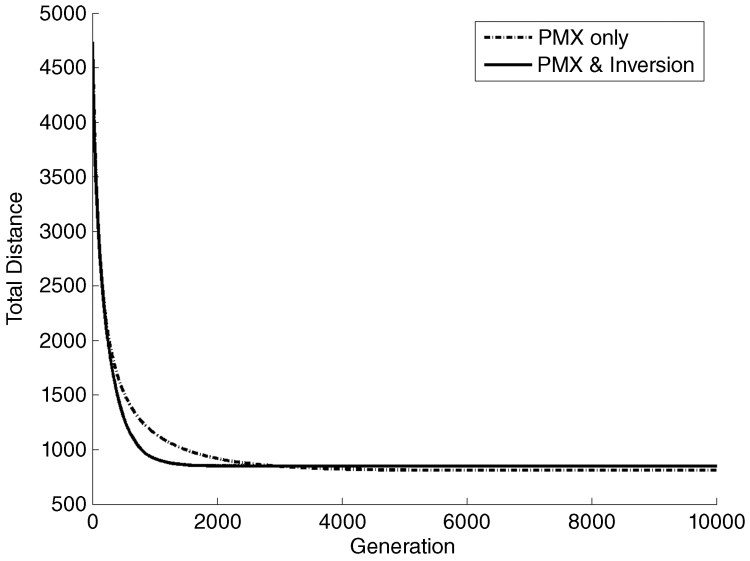

FIGURE 11.5 The average of 20 trials on the 100-city traveling salesman problem comparing PMX + inversion (0.5 probability/offspring) with inversion alone. The combination of PMX and inversion offers more rapid improvement on average for about the first 3000 generations. After that, the results suggest relying on inversion alone.

alone stalls at a mean tour length that's just less than 3000 units. It does this because selection has eliminated the variation in the population and the PMX operator cannot generate anything new. It can only recombine existing solutions. Once those solutions look the same, PMX (or any standard crossover operator) becomes incapable of searching further. The results show that adding in the possibility of mutating offspring via inversion allows the evolutionary search to proceed toward improved solutions.

It's natural to want to compare the results of the PMX + inversion approach with inversion alone to see if PMX is offering any benefit to the search. Figure 11.5 shows the results of that comparison through 10,000 generations. Up to generation 3000, the combination of PMX and inversion generated faster improvement than inversion alone, on average. After the 3000th generation, inversion alone did better on average at fine-tuning toward better solutions.

At this point, if you are thinking that perhaps it would make sense to use different variation operators at different times during an evolutionary search, rather than rely only on a fixed set of operators in a certain framework (e.g., PMX + inversion at 0.5 probability/offspring), that's great! We'll cover more on the self-adaptation of variation operators later in this chapter.

Returning to the broader concept of combinatorial optimization problems, there is a wide variety of these problems that can be addressed using evolutionary algorithms. Some of these are extensions of canonical problems, such as the traveling salesman problem. For example, consider the problem of optimizing the routing of delivery trucks for a major logistics company. Each truck must be loaded optimally in order to provide

each driver the opportunity to deliver the products on time. Not all customers are equally important; some deserve higher priority. Not all drivers are capable of handling the same equipment: Some are rated to drive larger vehicles. Determining the best way to allocate the materials to the trucks, assigning the drivers to their vehicles, and routing the vehicles to their destination present a complicated real-world problem.

There are other forms of combinatorial problems that are related to optimal subset selection problems. These are common in mathematical regression, in which you must explain observed data using independent variables but must decide which subset of variables provide the most explanatory power. They arise in many machine learning applications in the form of determining the appropriate number of neurons and topology for a neural network, or the number and shape of membership functions in a fuzzy control system. Similarly, data structures of variable size such as finite-state automata and symbolic expression trees also pose opportunities for combinatorial optimization.

11.3 SOME MATHEMATICAL CONSIDERATIONS

Any optimization approach should be amenable to analysis. Computers have increased in speed so dramatically that much of the analysis that used to be performed analytically (i.e., by mathematical theorem and proof) can be done effectively by empirical observation. Put another way, it's often possible to arrive at a good understanding of the mathematical properties of an approach through statistical estimation based on repeated sampling of a given procedure on a particular problem of interest. This approach is limited, however, to that particular set of observations and it can be difficult to generalize those results to other problems.

Early in the development of evolutionary computing techniques, there were broad beliefs that certain choices would offer superior performance generally. For example, one such belief was that binary representation would offer an intrinsic advantage over other representations regardless of the problem and that a selection method called *proportional selection* would provide an optimal way to create offspring in the search for improved optimization performance [Holland, 1975]. These and other specifics will be addressed in more detail in this section; however, it's important to note that mathematical analysis has shown that there truly is no single best approach to computational (i.e., computer-based) problem solving generally [Wolpert and Macready, 1997; see also Corne and Knowles [2003] and DuenezGuzman and Vose [2013]. Extending this "no free lunch" principle, it's important to note that there is always a challenge of generalizing the results of an evolutionary algorithm (or any algorithm) on a particular problem to other problems, even though they may appear to be closely related. The onus is on the researchers to effectively make the case if they want to claim a generalized result via empirical observations.

11.3.1 Convergence

Prior to the fast modern-day PC computers, extensive efforts were made to determine the mathematical properties of evolutionary algorithms both broadly and on specific

function optimization problems of interest. These efforts continue in order to more precisely describe the behavior of evolutionary algorithms in certain cases; however, for a basic understanding of general properties of evolutionary algorithms, it is important to review fundamental issues regarding their convergence properties.

11.3.1.1 Convergence with Probability 1 Evolutionary algorithms can be constructed in the framework of a Markov chain, which provides the mathematical tools to show that these algorithms can converge to an optimum solution with probability 1. Relevant publications can be found in Eiben *et al.* [1991]; Fogel [1992b]; Davis and Principe [1993]; and Rudolph [1994], and one accessible proof is found in Fogel [2006]. Rather than repeat the proof here, the essence of it is to consider different configurations of an evolving population to be states in a Markov chain.

If you don't know what a Markov chain is, it's a stochastic process, which is a fancy way of saying that it's a time-indexed sequence of random variables. The Markov chain is defined over a set of states (e.g., "awake" and "asleep"). The probability of transitioning from one state to another is time invariant and depends only upon the current state. For example, here's a Markov chain for "Awake and Asleep."

$$\begin{array}{c} \\ \text{Awake}[t] \\ \text{Asleep}[t] \end{array} \begin{array}{c} \begin{array}{cc} \text{Awake}[t+1] & \text{Asleep}[t+1] \end{array} \\ \begin{bmatrix} 0.4 & 0.6 \\ 0.8 & 0.2 \end{bmatrix} \end{array}$$

In this case, if you are in the state Awake at time t, then you have a 0.4 probability of staying in the state Awake and a 0.6 probability of transitioning to the state Asleep. If you are already in the state Asleep, then you have a 0.8 probability of transitioning to the state Awake, and a 0.2 probability of staying in the state Asleep.

Since the transition probabilities depend on the states in the system and don't depend on any particular time, the probability of being in a state of two time steps in the future can be found by multiplying the probability matrix (described by the letter P) by itself (squaring the matrix). Iterating from one point in time to another, say, k steps in the future, in a Markov chain is accomplished by raising P to the kth power. In our case, the basic transition matrix is written as

$$P = \begin{bmatrix} 0.4 & 0.6 \\ 0.8 & 0.2 \end{bmatrix}$$

and thus the two-step transition matrix is

$$P^2 = \begin{bmatrix} 0.64 & 0.36 \\ 0.48 & 0.52 \end{bmatrix}$$

So, if you are in the state Awake at a time t, then there is a 0.64 probability that you will be in the state Awake at time t + 2. It's natural to think about what state you would

be in if time went to infinity. In our case,

$$P^{\infty} = \begin{bmatrix} 0.5714 & 0.4286 \\ 0.5714 & 0.4286 \end{bmatrix}$$

That is, no matter which state you are in at time t, there is (approximately) a 0.5714 probability that you will be in the state Awake as time goes to infinity, and a 0.4286 probability that you will be in the state Asleep as time goes to infinity.

We can apply this same principle to study the long-term behavior of some evolutionary algorithms in which the time history of how the population arrived at its present state is not pertinent to the population's future trajectory, and also the probabilities for transitions for one configuration to another are fixed (stationary). If an evolutionary algorithm is constructed with a form of selection called *elitist selection*, in which the absolute best solution(s) in the population is always retained into the next generation, and if the algorithm is constructed such that a mutation operation can reach any state with nonzero probability (e.g., applying a Gaussian random mutation on all parameters), then the transition matrix for the Markov chain can be written as

$$P = \begin{bmatrix} 1 & 0 \\ R & Q \end{bmatrix}$$

where P is the transition matrix, 1 is a 1×1 identity matrix describing the absorbing state that has the global best solution, R is a strictly positive (all entries > 0) $t \times 1$ submatrix, Q is a $t \times t$ transition submatrix, 0 is a $1 \times t$ matrix of zeros, and t is a positive integer based on the size of the state space. Essentially, if the population already contains the best possible solution, then it is in the state "1" and will stay there forever. If the population doesn't contain the best solution, then the submatrix R describes the probabilities of transitioning to "1" in the next step, and submatrix Q describes the probabilities of transitioning elsewhere.

This is a special case of a more general transition matrix

$$P = \begin{bmatrix} I_a & 0 \\ R & Q \end{bmatrix}$$

where I_a is an $a \times a$ identity matrix.

In the limit, as k tends to infinity,

$$\lim_{k \to \infty} P^k = \begin{bmatrix} I_a & 0 \\ (I_t - Q)^{-1} R & Q \end{bmatrix}$$

The components "0" indicate that given infinite time, there is zero probability that the chain will be in a state that is not an absorbing state, and the absorbing state(s) was defined as a state(s) that contains a global optimum due to elitist selection. So, this

means that there is convergence with probability 1 to a global optimum. Special properties of the matrix entries R and Q provide for this result, implying a stronger form of convergence called *complete convergence* [Hsu and Robbins, 1947].[7]

This mathematical result is of limited utility because no one has infinite time to wait to discover a globally optimal solution; however, it may be useful to recognize that without elitist selection and without a mutation operation that can reach all possible states, this proof of convergence with probability 1 does not hold. In particular, if crossover is substituted for mutation, then the result does not guarantee convergence to a global optimum, but rather only to a homogenous state in which all solutions are identical and therefore no new solutions are possible.

11.3.1.2 Premature Convergence Evolutionary algorithms that rely predominantly on crossover or other recombination methods to generate offspring can sometimes suffer from what is termed *premature convergence*, which occurs when a population becomes homogeneous at a solution that is not the global optimum (or is less than desirable). The term is often used incorrectly to describe the effect of converging to a local optimum, but the origin of the term applies directly to the case in which no further progress is likely because the population lacks diversity, which effectively stalls an evolutionary algorithm that is heavily reliant on crossover or recombination. (We saw an example of premature convergence when exploring the use of the PMX operator on the traveling salesman problem. The PMX operator became ineffective when the population became homogeneous.)

In some studied cases of evolutionary algorithms that rely heavily on crossover, the likelihood of converging prematurely to a given solution has been shown to be related to the quality of the solution (i.e., there was more likelihood of stalling at a point if that point was of higher quality) [Spears and De Jong, 1997]. The most common methods for overcoming premature convergence include restarting from a new randomized population, using heuristics to move the population to a new collection of points (e.g., by hill climbing), or redesigning the approach. Early literature in evolutionary algorithms often recommended very high rates of crossover and very low rates of mutation [Holland, 1975; Goldberg, 1989], which made premature convergence more likely. Observing repeated premature convergence in an evolutionary algorithm suggests, at least, reconsidering the variation operators that are being used and giving consideration to modifying the probabilities of applying those operators, or creating new variation operators that are better tailored to the problem.

11.3.2 Representation

Designing an evolutionary optimization algorithm requires choices of representation, selection, and variation operators. With regard to representation, as mentioned early in this chapter, for many years in the early formulations of evolutionary algorithms, there

[7] Rudolph [1994] showed that some nonelitist evolutionary algorithms can also converge on strictly convex functions.

was a general belief that it would be beneficial to represent solutions using binary strings.

For example, when facing an optimization problem in $\Re^n$, instead of treating solutions directly as a vector $[x_1, \ldots, x_n]$ where $\mathbf{x} \in \Re^n$, the solution would be transformed into a series of bits $[x_1, \ldots, x_k]$, where k defined the length of the bit string. The greater the degree of desired precision, the larger the value of k would need to be. The belief was that longer strings generated more opportunities for an evolutionary algorithm to explore the subspace of possible solutions via substrings, and that this would provide a greater "information flow" [Holland, 1975].[8]

Many problems do not lend themselves easily to a description in binary strings. For example, consider representing a traveling salesman problem as a series of 1's and 0's. This is anything but straightforward. Suppose there are five cities. An intuitive representation would be an ordered list of cities, such as

[1 2 3 4 5]

Encoding these in binary could be done as

[001010011100101]

with each three-bit segment corresponding to the number of a city. But this representation is not easily varied by mutation or recombination. For example, mutating the fifth bit from 1 to 0 yields

[001000011100101]

and then the second city to visit is city "zero," which doesn't exist. Similarly, crossing two such bit strings would almost certainly generate offspring that would not correspond to legal tours of the available cities.[9]

[8] As a historical footnote, this belief originated in the subfield of evolutionary computing known as "genetic algorithms." The core idea of this approach was to view solutions in terms of *building blocks* that can be assembled via crossover. This is of course possible in some problem constructions, but not in others. When real-valued encodings were first tried on problems in $\Re^n$ using this approach, it violated the "building block hypothesis" of genetic algorithms; however, placing more emphasis on mutation and less on crossover was often successful with this real-valued representation and the results were published. This led to an approach called a "real-valued genetic algorithm." That is a misnomer because the core genetic algorithm concept of building block construction within parameters does not take place in the real-valued representation on $\Re^n$. In this case, it's essentially equivalent to what emerged in other branches of evolutionary computation, such as *evolution strategies* [Rechenberg, 1973], *evolutionary programming* [Fogel, 1990], and see Bremermann *et al*. [1966] and others. Differentiating between evolutionary approaches (e.g., genetic algorithms versus evolution strategies) is of dubious scientific value in modern evolutionary computing.

[9] In an ingenious procedure, Grefenstette *et al*. (1985) proposed a binary encoding for the traveling salesman problem that was amenable to recombination based on the order in which cities were removed from a list. But the procedure did not ultimately provide better optimization performance than comparative methods.

Fortunately, the notion that binary representations are universally better than other representations is false. In fact, there is no "best" representation across all problems, and under some conditions there is a provable mathematical equivalence of representations of different cardinality [Fogel and Ghozeil, 1997; see also Radcliffe [1992] and Battle and Vose [1993]]. Thus, the choice of a representation for a particular problem is often a matter of which provides the greatest intuition to the practitioner as the problem solver.

Some important aspects to consider when selecting a representation include the following:

1. The representation should optimally provide immediate information about the solution itself. For example, in the traveling salesman problem, the list of cities is suggestive of the solution.
2. The representation should be amenable to variation operators that are well understood for their mathematical properties and can exhibit a gradation of change. This means that variation operators should be available to make both small changes and big changes to any given parent(s), and that the likelihood of these different-sized changes can be controlled. For example, when searching for $\mathbf{x} \in \Re^n$ such that $f(\mathbf{x})$ is minimized, using a Gaussian variation operator on $\mathbf{x}$ allows generating offspring that are close to or far from $\mathbf{x}$, and this can be controlled by changing the standard deviation in each dimension (see Section 11.5).
3. Unless the objective is to explore the utility of a novel representation, utilizing representations that have been studied and for which results have been published may allow more systematic and meaningful comparisons.

With experience, you can gain better intuition about the effects of different representations, and how they are coupled with variation operators in order to search a solution space (landscape) for successively better answers to a problem of interest.

11.3.3 Selection

Selection describes either the process of eliminating solutions from an existing population or making proportionally more offspring from certain parents. Some common forms of selection include plus/comma, proportional, tournament, and linear ranking. Each of these has different effects on the likelihood of particular individuals to survive as parents or create offspring, and thus each has conditions that favor or disfavor its utility.

11.3.3.1 Plus/Comma Selection The notation $(\mu + \lambda)$ and (μ, λ) are now commonplace in evolutionary algorithms and refers to the two cases in which (i) μ parents create λ offspring and the best μ individuals are selected from among the $\mu + \lambda$ to be parents of the next generation, or (ii) μ parents create λ offspring and the best μ individuals are selected only from among the λ offspring to be parents for the next generation. Thus, in "plus" selection, all parents and offspring compete to be

parents for the next generation, whereas in "comma" selection, the parents die each generation and a surplus of offspring must be created. Some variations of these approaches include (i) the case of $(\mu + 1)$, which is sometimes referred to as "steady-state" or "continuous" selection, and (ii) allowing parents to survive for some maximum number of generations g before being removed in the comma selection process.

11.3.3.2 Proportional Selection As the name infers, proportional selection picks parents for reproduction in proportion to their relative fitness. (The procedure is sometimes also called *roulette wheel selection.*) Thus, the procedure is constrained to maximization problems on strictly positive fitness scores. If it were desired to find the minimum of $f(x, y) = x^2 + y^2$ using proportional selection, the problem would need to be turned first into a maximization problem, such as find the maximum of $1/f(x, y)$. The probability of selecting an individual in the population for reproduction is determined as

$$p_i = f(i) / \sum_{j=1}^{\mu} f(j)$$

where p_i is the probability of selecting the ith individual, there are μ existing individuals, and f(i) is the fitness of the ith individual. Rather than working directly on the fitness values, proportional selection can work on the relative ranking of solutions (thus making it applicable to minimization problems). Proportional selection is applied to individuals generally by selecting one individual for mutation, or two (or more) individuals for recombination until the population size for the next generation has been filled.

11.3.3.3 Tournament Selection There have been different forms of tournament selection in the history of evolutionary algorithms, but the more common one selects a subset of size q (often $q = 2$) from the existing population and selects the best of those q individuals to be a member of the next generation. The process is repeated until the population is filled. The process can be conducted with or without replacement, that is, individuals that are selected out of a q-tournament can be given an opportunity (or not) to be selected again in another q-tournament. As with proportional selection, tournament selection allows the possibility that solutions that are less than best can propagate into a future generation.

11.3.3.4 Linear Ranking Selection Linear ranking selection maps individuals to selection probabilities according to a prescribed formula based on the rank of the solution (see the earlier remark on proportional selection based on ranking in Section 11.3.3.2). There are many variations of this approach, but one early approach [Baker, 1985] assigned a probability to the ith ranked individual as

$$p_i = (\eta^+ - (\eta^+ - \eta^-)[i - 1]/[\mu - 1])/\mu$$

where p_i is the probability of selecting the ith individual, μ is the number of individuals in the population (sometimes described with λ, see [Bäck, 1994]), and

η^+ and η^- are user-controlled constants constrained by $1 \leq \eta^+ \leq 2$ and $\eta^- = 2 - \eta^+$. For example, if $\mu = 100$ and $\eta^+ = 1.5$, then $\eta^- = 0.5$, and the probability of selection of the ith-ranked solution in the population is

$$p_1 = (1.5 - (1)(0/99))/100 = 0.0015$$

$$p_2 = (1.5 - (1)(1/99))/100 = 0.00148989\ldots$$

$$p_{100} = (1.5 - (1)(100/99))/100 = 0.0048989\ldots$$

Thus, better solutions are favored over lesser solutions.

11.3.3.5 Example Let's examine the effects of different selection operators when combined with a simple evolutionary algorithm to search for the minimum of Rastrigin's function in two dimensions. The function is $f(x, y) = 20 + x^2 - 10\cos(2\pi x) + y^2 - 10\cos(2\pi y)$. We'll initialize a population of 30 candidate solutions uniformly at random from $[-10, 10]$ in each dimension. Variation will be accomplished by a Gaussian mutation, zero mean, with standard deviation equal to 0.25 applied to each dimension. We'll create 60 offspring from 30 parents and compare the results of a (30, 60) selection with those from roulette wheel selection and tournament selection with $q = 2$. Figure 11.6

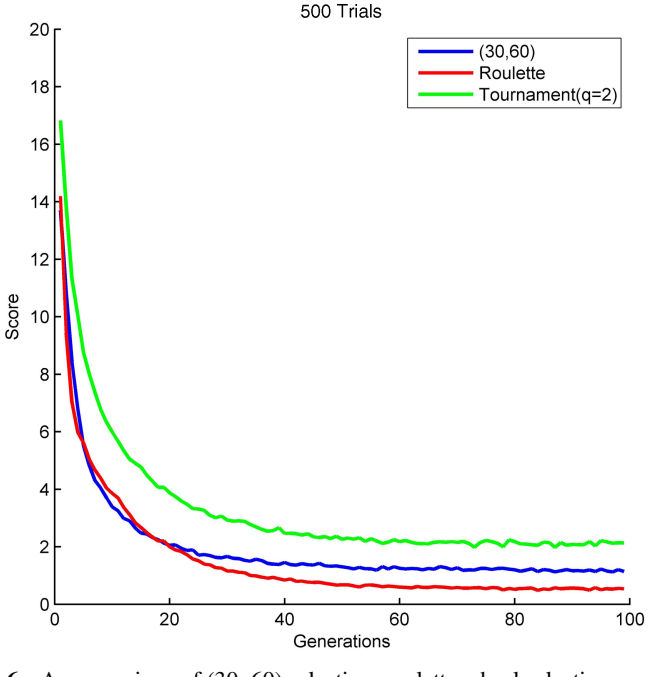

FIGURE 11.6 A comparison of (30, 60) selection, roulette wheel selection, and tournament selection with $q = 2$ on Rastrigin's function in two dimensions. The results, averaged over 500 trials, favor the use of roulette wheel selection in this case.

shows the average results of 500 trials with each method through the first 100 generations. Tournament selection with $q = 2$ evidenced the slowest rate of improvement, while roulette wheel selection found better solutions on average than did the (30, 60) selection method. It's important to also know the variation of the data in order to assess whether or not these differences are meaningful.

11.3.3.6 Considerations There are various extensions to these basic forms of selection. For example, *elitist* selection can be applied, which automatically ensures that the best solution in a population is retained in the next generation. Nonlinear ranking methods can be used [Michalewicz, 1992]. Generally, however, there is a continuum of selection methods from those that are "soft" to those that are "hard." The harder the selection, the faster the better solutions can overtake the population.

This is often described in terms of *takeover time*, which is the expected number of generations required to fill a population with copies of what is initially a single best individual when applying only selection (no variation). As shown by Bäck [1994], for typical settings of the parameters in the procedures above and on given test cases, the order of selection strength from weakest to strongest is proportional selection, linear ranking selection, tournament selection, and then plus/comma selection.

If you are using an evolutionary algorithm that relies heavily on recombination, then diversification is required for the population to search for new solutions (see premature convergence (Section 11.3.1.2)). In this case, it may be advisable to try a weaker form of selection so as to maintain more diversification; however, this may serve as a drag on progress toward an optimal solution. Stronger selection may result in faster optimization, but that optimization may be only toward a local optimum. You must weigh the trade-offs between different rates of progress and the suitability of alternative variation operators that require diversification to greater or lesser degree as those trade-offs pertain to the problem at hand.[10]

11.3.4 Variation

Variation operators provide the means for searching the solution space for improved solutions, or potentially for weaker solutions that could lead to improved solutions. (The latter comes in the case of having a weak selection method that allows less-than-best solutions to survive and be the basis of further exploration.) There are many traditional variation operators, some of which have already been discussed, such as binary mutation, Gaussian mutation, or one-point, n-point, or uniform crossovers. Although some people go into the problem with a set of variation operators in mind

[10] As a historical note, for many years, proportion selection was viewed as the optimal selection method based on analysis in Holland [1975]. It was claimed that proportional selection generated "minimum expected losses" when sampling from competing random distributions with different means and variances (analogous to sampling from different subspaces). This was later shown to be false by counterexample [Rudolph, 1997] and by mathematical derivation [Macready and Wolpert, 1998] (i.e., the analysis in Holland [1975] was flawed). Each person must decide which form of selection to employ based on the problem. There is no generally optimal form of selection.

and then seek to adjust the representation and selection operators, it's clear that the choice of variation operators goes "hand in glove" with the choice of representation.

11.3.4.1 Real-Valued Variation When searching $\Re^n$, it is typical to use a Gaussian mutation operator. Recall that a Gaussian distribution is defined by two parameters: the mean and standard deviation (or variance). When applying a Gaussian mutation, the mean is set typically to zero, providing for an unbiased search in the neighborhood of a parent. The size of the neighborhood is defined by the standard deviation, σ; the smaller the standard deviation, the smaller the search neighborhood, although for any $\sigma > 0$, there is a nonzero probability of a Gaussian random variable returning any value between $-\infty$ and ∞. But the probability of a large move away from 0 may be very small. For example, with $\sigma = 1$, the probability of a zero-mean Gaussian random variable returning a value greater than 1 or less than -1 is approximately 0.32. For $\sigma = 0.5$, this probability is approximately 0.05. Aspects of controlling the setting of σ will be described in Section 11.5; however, note that the progress that a search based on a Gaussian random mutation operator will make is highly dependent on the value(s) of σ (in each dimension).

At times, it may be desirable to have a larger probability of creating a greater distance between an offspring and its parent. Rather than increasing the value of σ in a Gaussian mutation, an alternative is to employ a Cauchy-distributed random mutation. A Cauchy random variable is constructed by taking the ratio of two independent identically distributed Gaussian random variables. (For example, using a random number generator, create two independent Gaussian-distributed numbers and divide one by the other.)

The Cauchy distribution has the interesting mathematical properties of being symmetric and yet having no explicit mean or standard deviation (i.e., the expected value of the random variable is undefined and so are all higher moments of the distribution). The Cauchy distribution has "fatter tails" than a corresponding Gaussian distribution, thus the probability of mutating a real-valued parameter to a greater extent is markedly greater. This can be helpful for escaping from locally optimal solutions. Section 11.5 addresses approaches to trading-off between Gaussian and Cauchy mutation operators in real-valued evolutionary optimization.

11.3.4.2 Multiparent Recombination Operators Earlier discussion highlighted the use of one-point, n-point, and uniform crossovers, as well as *blending* recombination that averages components of multiple individuals. Note that there is no limit to relying on two parents. Nature provides inspiration for evolutionary algorithms but it is up to the designer to find what inspires him or her. In some cases, it may be beneficial to recombine elements or blend parameters of three or more solutions. When applying blending recombination, averaging components across multiple solutions, note that the result is an offspring that essentially estimates the mean of the population. The greater the number of individuals that contribute to that blending, the more reliably the offspring will estimate the population mean (centroid). When a population is contained in a locally optimal region of the search space, a blending operator can accelerate convergence toward the local optimum. This comes at the expense of sacrificing searching outside of the local region.

11.3.4.3 Variations on Variable-Length Structures Certain representations employ data structures of variable length. For example, consider the case of evolving a neural network that can adapt not only its weights and bias terms but also the number of nodes it uses and the feedback loops that it employs (if any). The data structure used might well be of variable length. As another example, consider the case of evolving a collection of fuzzy membership functions that are used in a fuzzy controller. The number of fuzzy functions, as well as their shape and location, could be subject to evolutionary adaptation. When considering a representation that poses a variable-length data structure, you can consider an array of variation operators with the idea of ensuring the possibility of a thorough search of the solution space. Three other common examples involve finite-state machines, symbolic expressions, and difference equations.

Finite-State Machines Finite-state machines have long been used in evolutionary algorithms for predicting sequences of symbols (time series prediction). Fogel *et al*. [1966] employed Mealy/Moore finite-state automata for this purpose. A finite-state machine was defined by its number of states, its starting state, the input/output function for each state, and the input/state transition function for each state. Suppose that the available symbols were $\{0, 1\}$, then one finite-state machine could have the following characteristics (Figure 11.7):

(a) Two states.
(b) State 1 is the start state.
(c) In state 1, input of 0 yields 0, input of 1 yields 1; in state 2, input of 0 yields 1, input of 1 yields 0,
(d) In state 1, input of 0 transitions to state 2, input of 1 remains in state 1; in state 2, input of 0 remains in state 2, input of 1 transitions to state 1.

There are five modes of mutation that follow naturally from the description of the finite-state machine:

(a) Add a state.
(b) Delete a state.
(c) Change the start state.

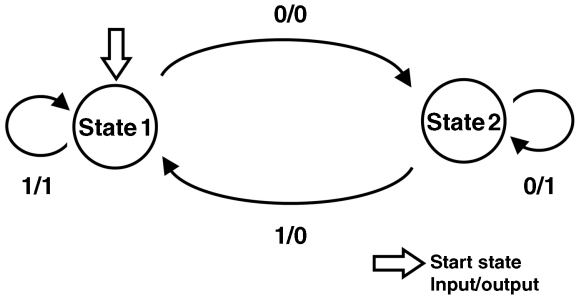

FIGURE 11.7 An example of a finite-state machine as described in the text.

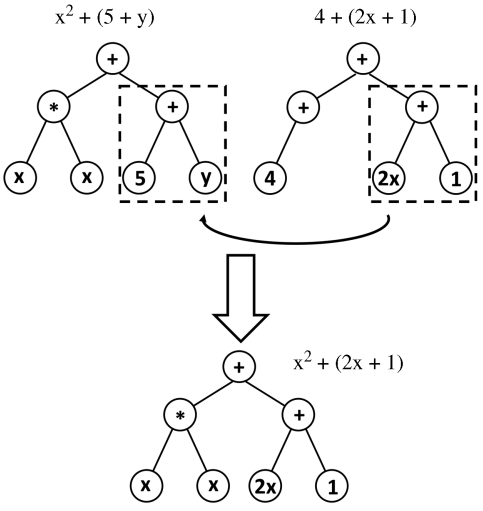

FIGURE 11.8 An example of subtree crossover on two symbolic expressions. In the example, it's desired to have a function that represents $f(x) = x^2 + 2x + 1$. This is accomplished by swapping the right branches of the trees.

(d) Change an input–output relationship.

(e) Change an input–state transition relationship.

By employing nonzero probabilities of applying each of these mutation operators, a parent finite-state machine can create a very different offspring (in terms of its input–output behavior) or one that is very similar. Note that the effects on the sequences that a finite-state machine may generate (i.e., the "behavior" of the machine) vary across these mutation operators. The effect of deleting a state may be much greater than merely adjusting the output a state generates for some particular input.[11]

Symbolic Expressions Similarly, consider evolving a solution to a problem using a symbolic expression (s–expression).[12] The data structure is of variable length and there are opportunities to modify any terminal node and to perform subtree recombination. Suppose the problem was to find a polynomial expression that returned the value of $x^2 + 2x + 1$ for an input of x. Figure 11.8 shows an example of a subtree recombination

[11] In practice, adding and deleting states are limited to a maximum number of states and a minimum of one state, respectively. Also, when deleting states, any transitions that pointed to that deleted state must be redirected. When adding states, it may be desirable to also affect state transitions to ensure that the new state has the possibility of being expressed and thus changing the behavior of the finite-state machine. States that cannot be expressed are sometimes described as *introns* in analogy to sections of DNA that do not translate directly to proteins.

[12] This is often called *genetic programming* (following Koza [1992]), but there's no scientific benefit derived from giving separate names to applying variation operators to different data structures. Similarly, there's no need to describe evolutionary algorithms applied to neural networks, fuzzy systems, or other data structures with some new monikers.

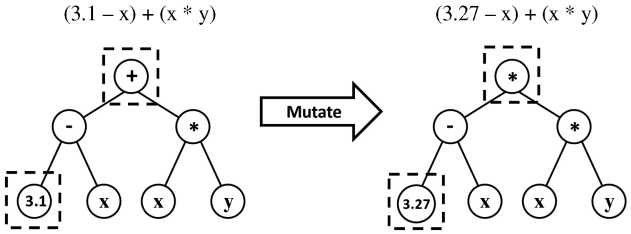

FIGURE 11.9 Mutation can be accomplished on tree-based expressions by changing operators or numeric terms, as indicated in the figure. In addition, variables can be modified, and branches can be grown or eliminated.

that would take two suboptimal expressions and combine them to create the correct formula. Figure 11.9 shows an example of mutating specific values associated with nodes in the expression. It would also be feasible to add/construct new branches in the expression tree or delete existing ones. As with the prior example pertaining to finite-state machines, each of these operations may have a different degree of behavioral change on the expression, and thus the probabilities of applying the operators can be adjusted so as to provide for a more (or less) gradual search of the solution space.

Difference Equations Another example comes in a form of mathematical modeling known as *system identification*. In this case, input–output examples are given for a particular system and the objective is to construct a mathematical model that represents the input–output relationship. This is a typical neural networks application; however, particularly on time series problems, it is more common to address system identification using autoregressive (AR) moving-average (MA) models (or ARMA), which are particular forms of difference equations. These are of the form

$$x[t+1] = a_0x[t] + a_1x[t-1] + \cdots + a_jx[t-j] + e[t] + b_1e[t-1] + \cdots + b_ke[t-k]$$

where x[t] is the observed variable at time t (this is the AR part), e[t] is the random noise occurring at time t (this is the MA part), and $a_0, \ldots, a_j, b_1, \ldots, b_k$ are coefficients. A standard approach employs gradient methods to estimate the coefficients for a model that has a predefined number of lag terms for the AR and MA parts of the model. This is problematic because the best choice of j and k are unknown *a priori*, and thus multiple searches must be conducted over different choices of j and k to provide confidence in a final result. Moreover, some nonlinear time series involve ratios of AR and MA processes, and gradient methods may lead to entrapment at suboptimal solutions (just as backpropagation may do the same when optimizing weights and bias terms of a neural network).

An evolutionary approach to this problem encodes both the number of lag terms and the coefficients as a single solution, such as

$$[2, 1.5, 0.7, 0]$$

where the first integer is the number of lag terms in the AR process, the next two entries are the coefficients of those lag terms, and then the next integer is the number

of lag terms in the MA process. As that is zero, there are no more entries. This specific representation corresponds to the model

$$x[t+1] = 1.5x[t] + 0.7x[t-1] + e[t]$$

The following are the variation operators that follow naturally from this representation:

(a) Vary the number of AR terms
(b) Vary the number of MA terms
(c) Vary the coefficients of any/all terms

In addition, it may be feasible to apply recombination operators to combine two or more models. Note that in cases where the degrees of freedom of a model are allowed to increase as part of the search process, a trade-off should be employed that incorporates the goodness-of-fit of the model to the data and also the number of degrees of freedom. An example is described in Section 12.2.2.

11.3.4.4 Considerations One common approach to evolutionary optimization involves employing a high rate of crossover and low rate of mutation. A possible justification for this approach is a view of evolution in nature proceeding by assembling building blocks of genetic code between parents. This view of evolution is controversial (as it was 20 years ago, see Atmar [1994]), but the effectiveness of believing that a particular problem is amenable to solution by assembling building blocks can be tested empirically.

As an example, let's explore this with a function that poses multiple local optima. The function is

$$f(x,y) = \exp(-[(x-3)^2 + (y-3)^2]/5) + 0.8\exp(-[x^2 + (y+3)^2]/5) + 0.2[\cos(x\pi/2) + \cos(y\pi/2))] + 0.5$$

which is displayed in Figure 11.10. The maximum of the function in this range is 1.6903, which occurs at (3, 3), and let's say we want to find that maximum. In order to assess the possibility for recombining building blocks, we'll choose a binary representation that was familiar within one early school of evolutionary algorithms. The representation takes the real values for x and y and encodes them in eight bits ranging from [00000000] = −5 to [11111111] = 5. With eight bits of precision, there are 2^8 possible values for x and y, and a precision of 0.039. If we wanted more precision, we could increase the number of bits used to represent the x and y values. For this level of precision, the maximum value of the function is 1.6902.

We can compare the effects of one-point, two-point, and uniform crossovers on this function. Recall that one-point and two-point crossovers select and recombine segments of a solution that might serve as building blocks for new solutions. Uniform crossover doesn't do that. It chooses each component (each bit) from each parent with equal probability. We can also compare the effects of regular binary coding and what's called Gray coding, which uses binary but encodes numbers so that two successive numbers differ by only one bit.

Figures 11.11 and 11.12 show a comparison over 200 trials, starting in a range of −5 to 5 in x and y, with 100 parents making 100 offspring under roulette wheel selection for

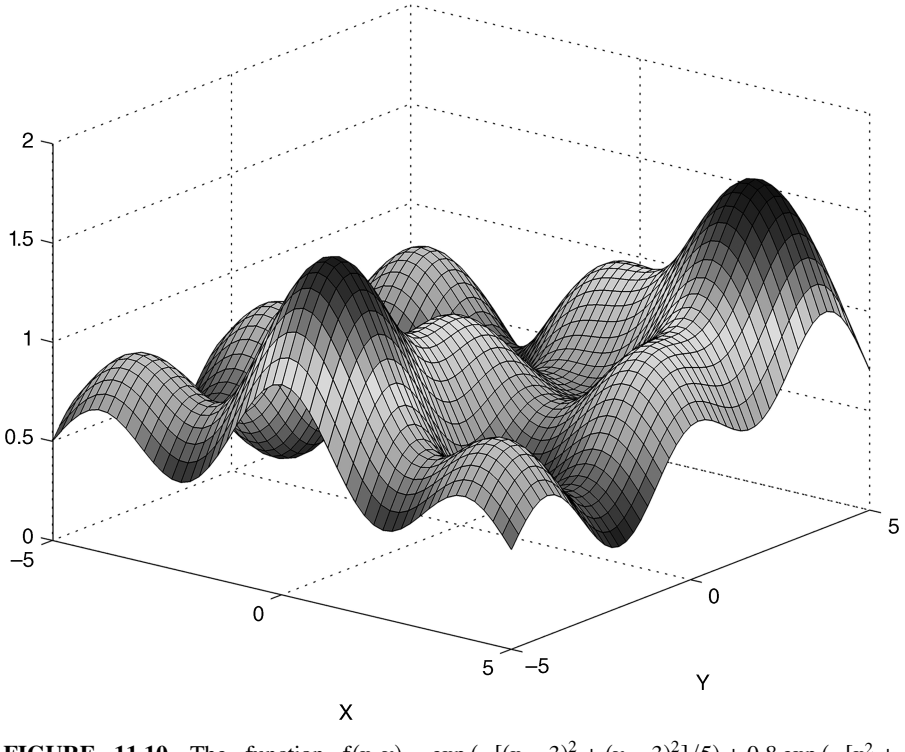

FIGURE 11.10 The function $f(x, y) = \exp(-[(x-3)^2 + (y-3)^2]/5) + 0.8\exp(-[x^2 + (y+3)^2]/5) + 0.2[\cos(x\pi/2) + \cos(y\pi/2))] + 0.5$. It reaches a maximum height of 1.6903.

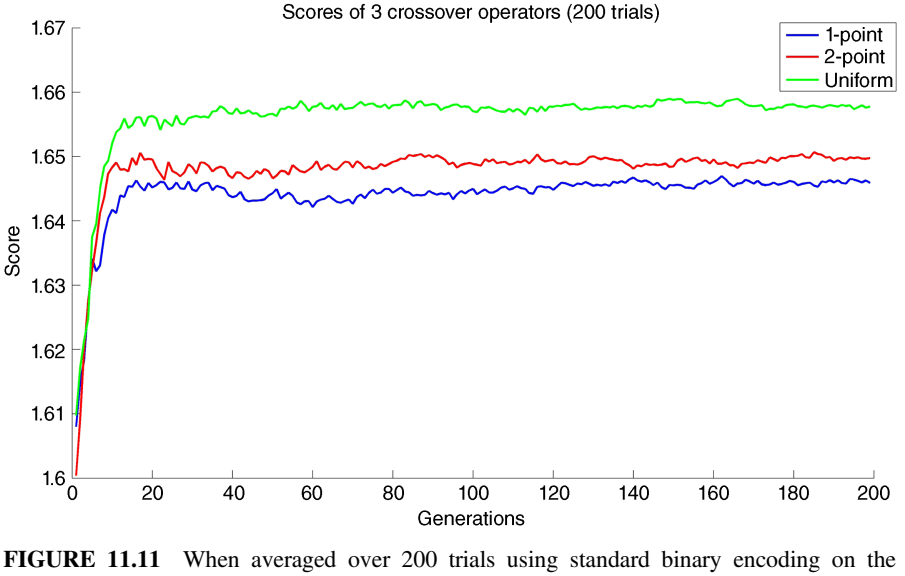

FIGURE 11.11 When averaged over 200 trials using standard binary encoding on the function in Figure 11.10, uniform crossover outperformed two- and one-point crossovers.

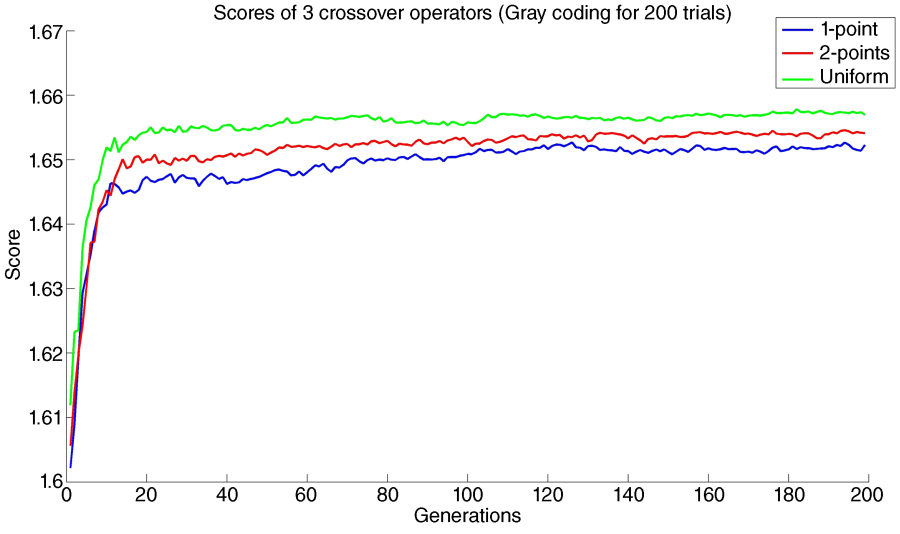

FIGURE 11.12 When averaged over 200 trials using Gray coding on the function in Figure 11.10, uniform crossover again outperformed two- and one-point crossovers, but did not reach as high a score on average as when using standard binary encoding.

each of the three recombination operators using traditional binary encoding and Gray coding. The results indicate that the "building block hypothesis" for bringing together good subsections of solutions does not hold for this problem, as uniform crossover outperforms one- and two-point crossovers under both representations.

One procedure for testing this hypothesis is to compare the effectiveness of an evolutionary algorithm that uses recombination with one that does not on the same problem. Statistical hypothesis testing can then be employed to determine if there is any statistically significant evidence to favor one approach over another. Many test problems in the literature have shown niches where mutation alone has worked better than recombination, and others where it has not [Fogel and Atmar, 1990; Schaffer and Eshelman, 1991; Fogel and Stayton, 1994; Chellapilla, 1997; Luke and Spector, 1998; Spears, 1998; and many others].[13] You might try an evolutionary algorithm that uses a real-valued representation and Gaussian mutation on this function and see how it compares to the results in Figures 11.11 and 11.12.

Another procedure, offered in Jones [1995a], employs what is called *headless chicken crossover*. The procedure takes an existing parent and instead of crossing it over with another existing parent, it crosses the parent with a completely random

[13] Despite more than 20 years of empirical and theoretical comparison between different evolutionary optimization methods showing no single approach works best, it's common to observe complacency and a reliance on a particular approach that is "comfortable" to a researcher. This is true across computational intelligence methods, and it's likely also true in science generally. People tend to start with the tools they are most comfortable with, rather than starting with the problem and asking which tools they should employ (see Michalewicz and Fogel [2006]).

solution. At this point, the mechanism of crossover is still present but the effect of the mechanism is purely a large mutation, since it is just random material that is being swapped with the existing parent to create offspring. There have been cases shown (and this was surprising initially to many) in which headless chicken crossover outperformed the usual one-point or other variations of crossover (e.g., Angeline [1997]; see also White and Poulding [2009]). In those cases, greater attention can be placed on mutation operators, or attention can be placed on reformulating the problem representation in order to facilitate crossover.[14]

One variation operator that has received little attention is *inversion*. As proposed originally [Fraser, 1968; Holland, 1975], this operator inverted the index position of elements in a solution string. For example, given a set of values $x_1, \ldots, x_5$ with an additional subscript for the position of those values, an initial solution might be

$$[x_{11}, x_{22}, x_{33}, x_{44}, x_{55}] \tag{11.1}$$

but after inverting between position 2 and 4, this would be

$$[x_{11}, x_{24}, x_{33}, x_{42}, x_{55}] \tag{11.2}$$

Thus, if the objective were to, say, minimize the value of $f(\mathbf{x}) = x_1^2 + 2x_2^2 + 5x_3^2 + x_4^2 + x_5^2$ then each of (11.1) and (11.2) above would evaluate to the same $f(\mathbf{x})$. Inversion here did not change the values of $x_1, \ldots, x_5$, it only changed the position of these values in the solution string. The suggested purpose of doing this was to ensure that one-point crossover would have more opportunity to find building blocks between elements of a solution regardless of their initial position in the solution string. Given all the research performed on the effectiveness of variation operators, there has been comparatively little research on the value that could be imparted by utilizing this form of inversion in certain cases.

When designing an evolutionary optimization algorithm keep in mind that the number of variation operations applied in creating an offspring can be greater than 1. It is typical to see an approach such as apply crossover with a probability of 0.9 and mutation with a probability of 0.01, or apply one from a list of variation operators each with equal probability. But a greater exploration of a search space, and also sometimes a more gradual one, can be created by repeating variation operators. For example, when mutating a finite-state machine (Section 11.3.4.3), rather than select only one variation operator, or rather than test for the application of each variation operator once, another parameter could be the number of variations to apply. One way to

[14] One of the authors (DF) has seen this many times in more than 30 years of evolutionary algorithm experience: An evolutionary approach does not yield effective results and thus the designer seeks to reformulate the problem to be more amenable *to crossover*. An alternative and potentially more fruitful suggestion is to reformulate the algorithm to be more amenable to yielding *a superior solution*. Crossover is just one of many tools in the virtual Swiss-army knife of the evolutionary algorithmist. Using crossover is not an objective in and of itself.

engineer this is to use a Poisson random variable with a set mean rate, and use a sample from that random variable to determine how many variation operations to apply. This also leads to the concept of having the evolutionary process learn how much variation to apply, which is treated in Section 11.5.

It is easy to become accustomed to a basic plan of applying an evolutionary algorithm for optimization. Some opt for a plan that involves a high rate of crossover between parents and a low rate of mutation. Others opt for no crossover or other form of recombination at all and employ only mutation unless the problem at hand suggests that mutation alone is insufficient or inefficient. With experience, you can demonstrate to yourself that neither of these approaches is optimal generally. It is often beneficial to think imaginatively about how to design variation operators that exploit the characteristics of the objective function being searched in light of the chosen representation and type of selection.

Different variation operators can be effective at different stages of an evolutionary optimization process. That is, it may be that recombination may be most effective early in optimization with mutation serving to fine-tune optimization in a later stage. Or it may be that mutation and recombination actually generate very similar expected rates of progress given the same population and fitness criteria. An example of this is given in Section 11.5.5. The use of static probabilities of applying variation operators has been of limited utility in the experience of one of the authors (DF); in contrast, the concept of having the evolutionary algorithm adjust its own search via reinforcement learning mechanisms has demonstrated utility and is described in more detail in Section 11.5.

11.4 CONSTRAINT HANDLING

Almost all real-world problems are constrained problems. For example, suppose you are designing an optimal schedule for a bus company. They have a specific number of existing buses. That is one constraint. They have a limited budget to purchase additional buses. That is another constraint. They have a limited number of qualified drivers for each different type of bus. That is yet another constraint. Each bus has limited capacity, which is yet another constraint. We could invent more constraints for this problem, including required maintenance, roads that can and cannot be used, the available time for each driver to work each week, and so forth.

Thus, when applying evolutionary algorithms it is important to consider how to treat the constraints of the problem. Some of these constraints are part of the objective, whereas some are part of the parameters of a solution and therefore impose boundary conditions on the search space. Each of these may pose *hard* or *soft* constraints.

A hard constraint is one that, if violated, makes the entire proposed solution worthless. For example, suppose you are designing a new golf club for professional golfers. The United States Golf Association (USGA) requires that all golf clubs be at least 18 in. in length and not more than 48 in. in length. These are hard constraints on your design because anything outside of these limits cannot be used in regulation play.

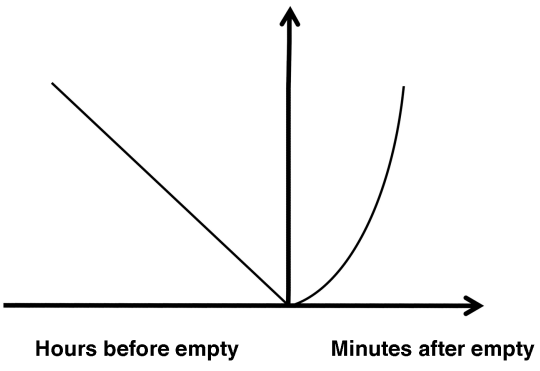

FIGURE 11.13 A sketch of a penalty function to describe the cost of a fuel truck arriving too early or too late to a fuel station to refuel the station. The function describes the cost to the company. If the truck arrives too early, the company pays the driver an hourly wage to sit and wait until the station becomes close to empty. If the truck arrives too late, the company loses the opportunity to sell fuel to customers and also loses customers to its competitors.

A soft constraint is one that can be violated, but there is some imposed penalty for violating it, and perhaps the penalty increases with the degree of violation. For example, suppose you are creating a schedule for fuel trucks to refuel gas stations. A constraint would be to design a schedule in which no station is ever empty, which would mean the oil company would have customers waiting at the gas pumps for a refueling truck. But this is not likely to be a hard constraint. If you crafted a schedule that had a gas station out of fuel for 1 min, it might lead to some bad publicity. In fact, it would be more likely to add to bad publicity the longer the station remained empty. So, a penalty function of the form shown in Figure 11.13 could be used to describe this soft constraint.

It is often helpful to craft an objective function that comes in two parts. The first involves the primary criteria of interest and the second involves a penalty function that treats constraint violations. Sometimes these two parts can be simply added together. For example, in information-theoretic model building, there are many criteria that trade off the goodness-of-fit of a mathematical model to available data and the number of parameters (degrees of freedom) that the model employs. For example, Akaike's information criteria (AIC) [Akaike, 1974] can be written as

$$\mathrm{AIC}(\mathbf{x}) = -2\ln(\mathrm{L}) + 2\mathrm{p}$$

where $\mathbf{x}$ is the parameter vector of the model, L is the likelihood function of the model, and p is the number of model parameters. If you don't know what a likelihood function is, you can think of it as a measure of the goodness-of-fit of the model to the available data. Better models generate lower AIC scores. For each new parameter that is added, the likelihood function will be higher (and thus $-2\ln(\mathrm{L})$ will be lower), but

it has to be sufficiently better to "pay for" the addition two points that the extra parameter will cost. Here, there is no hard constraint on the number of parameters, but each additional parameter comes at a cost.

When a hard constraint is involved, the objective function can be set to an infinitely bad score if the constraint is violated. For example, suppose that in the above AIC minimization problem there was a constraint not to use more than 10 parameters. Then the objective function could be written as

$$f(\mathbf{x}) = \begin{cases} \text{AIC}(\mathbf{x}), & p \leq 10 \\ \infty, & p > 10 \end{cases}$$

The difficulty with this approach is that it sets up areas of the search space that have no information content to direct the evolutionary search to any places of improved performance. Anything in the search space that violates the hard constraint is equally worthless. An alternative is to impose a penalty for violating the hard constraint that increases in effect gradually over successive generations. In this way, the constraint is treated as a soft constraint at first but over time solidifies as a hard constraint. A drawback to this approach is the requirement for tuning a schedule for how to transition the soft constraint to a hard constraint.[15]

Sometimes it is possible to treat constraints in the objective function as constraints in parameters. For example, in the above case, a hard constraint that $p \leq 10$ could be handled by ensuring that any mutation that generates $p > 10$ is set aside, or that mutations to values of p that are greater than 10 are reflected into the feasible range. That is, if a parent had $p = 9$ and a mutation were to make p grow by 3, then it would first become 10, and then it would reflect into the feasible range to 9 and then again to 8. This has advantage of always evaluating feasible solutions, but it can have the disadvantage of introducing boundary effects on variation operators that are sometimes difficult to intuit.

Thinking about problem solving is often more important than attempting problem solving. By thinking about the problem at hand, it is often possible to garner insights about that problem and design-specific operators to address the intricacies and constraints posed therein. For example, Michalewicz *et al.* [1996] studied an n-dimensional problem of maximizing:

$$f(\mathbf{x}) = \left| \left[\sum_{i=1}^{n} \cos^4(x_i) - 2 \prod_{i=1}^{n} \cos^2(x_i) \right] \Big/ \sum_{i=1}^{n} i x_i^2 \right|$$

subject to $\prod_{i=1,\ldots,n} x_i \geq 0.75$, $\sum_{i=1,\ldots,n} x_i \leq 0.75\,n$, and $0 \leq x_i \leq 10$ for $1 \leq i \leq n$. By recognizing that the maximum point was likely to exist on the boundary condition $\prod_{i=1,\ldots,n} x_i = 0.75$, Michalewicz *et al.* constructed an evolutionary algorithm that searched only on that boundary. They initialized the algorithm on the boundary

[15] This is analogous to tuning a schedule for reducing the temperature in a simulated annealing procedure [Kirkpatrick *et al.*, 1983].

condition and then constructed a new "geometric crossover" that combines two parents **x** and **y** as follows:

$$[x_1, \ldots, x_n]$$

$$[y_1, \ldots, y_n] \rightarrow [x_1^{\alpha} y_1^{1-\alpha}, \ldots, x_n^{\alpha} y_n^{1-\alpha}]$$

where α is selected uniformly at random in the interval [0, 1]. When applied to any two solutions on the boundary condition, this operator returns another solution on the boundary. Finally, they also employed a mutation operator that selected two components of an existing solution and multiplied one by a factor $q > 0$ and the other by $1/q$. By tailoring the operators to the problem in this way, Michalewicz *et al.* [1996] reported finding a new best solution for this problem (better than previous approaches that had relied on traditional forms of evolutionary algorithms).

11.5 SELF-ADAPTATION

When considering a simple evolutionary algorithm, say, one that employs only Gaussian mutation to real-valued components of potential solutions, **x**, it is evident that the setting of the standard deviation of the mutation will affect the degree of progress that can be made toward an optimum. For example, consider Figures 11.14 and 11.15, which show the case in one dimension. The function $f(x) = x^2$ is to be minimized and there is a parent solution at $x = 2$. Using a zero-mean Gaussian

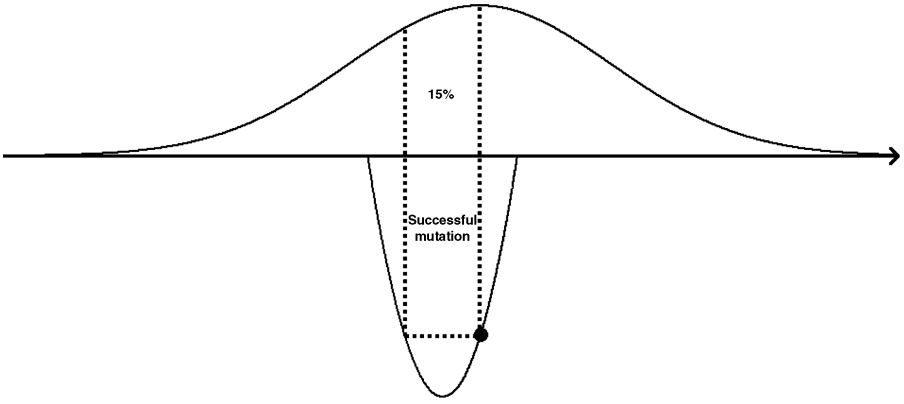

FIGURE 11.14 Suppose that there is a one-dimensional function to be minimized, such as the parabola in the figure, and the current solution is represented by the black dot. Under zero-mean Gaussian mutation, as the step size grows large, the probability of finding a solution that has a better score (corresponding to a lower point on the function) becomes very small. Here the probability is illustrated at about 0.15.

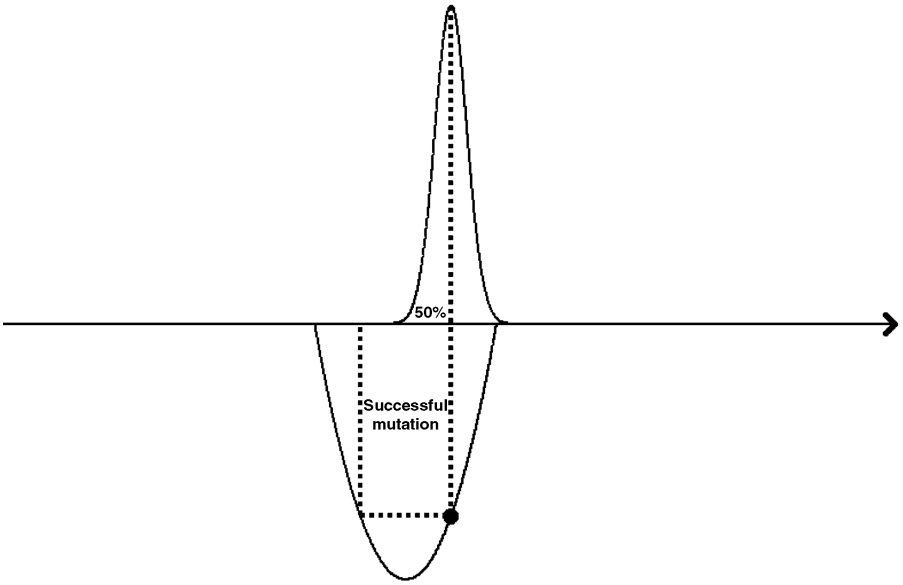

FIGURE 11.15 Suppose we have the same one-dimensional function to be minimized, such as the parabola in the figure, and the current solution is again represented by the black dot, but this time we have a zero-mean Gaussian mutation with a very small standard deviation. As the standard deviation shrinks close to zero, the probability of improving tends to 0.5; however, the amount of improvement at that limit will be close to zero.

distribution for mutation, denoted $N(0, \sigma)$, for increasingly large values of σ, say $\sigma = 10^6$, the likelihood that an offspring will be better than the parent is extremely low (zero in the limit) (see Figure 11.14). In contrast, for infinitesimal values of σ, say $\sigma = 1/$(national debt of the United States), the likelihood that an offspring will be better than the parent approaches 0.5. This increased likelihood of success comes with the drawback of a high likelihood of making very little progress toward the optimum (because the step size is so small) (see Figure 11.15).

The problem shown poses a balance between the step size of the mutation and the expected rate of progress. For very small steps, the likelihood of success can be maximized but the rate of progress is slow. For steps that are too large, the likelihood of success is minimized and the rate of progress is also slow. In between there is an optimum step size for which the likelihood of success is not maximized, but the rate of progress toward the optimum is maximized.

11.5.1 The 1/5 Rule

Rechenberg [1973] studied two minimization problems that are linear and quadratic functions of $\mathbf{x}$. Using a $(1 + 1)$ evolutionary algorithm that relied on zero-mean Gaussian mutation, Rechenberg studied the trade-off between the rate

of improvement (i.e., how quickly the best solution's score decreases as a function of the number of generations) and the probability of improvement (i.e., the likelihood that an offspring will be better than its parent). It was shown that as the number of dimensions $n \rightarrow \infty$, the maximum rate of improvement on the linear function occurred when the probability of improvement was approximately 0.184, and for the quadratic function it was approximately 0.27. Rechenberg selected 0.2 as a compromise between these two values and offered a heuristic called *the 1/5 rule*: The ratio of successful mutations to all mutations should be about 1/5.

Schwefel [1995] suggested a simple method for computing this empirically. After every m mutations, determine the success rate over the previous 10m mutations. If the success rate is less than 0.2, multiply the step size (σ) by 0.85. If the success rate is greater than 0.2, then divide the step size by 0.85. When the success rate is too low (< 0.2), the step sizes will become smaller, and conversely when the success rate is too high (> 0.2), the step sizes will increase.

It is important to recall that the 1/5 rule is a heuristic that was derived only from two simple types of functions, as the number of dimensions tends to infinity, and for a $(1 + 1)$ evolutionary algorithm. There is no general case to make for the utility of the 1/5 rule (see Chellapilla and Fogel [1999] for counterexamples). Still, it illustrates that static parameters for variation operators are very unlikely to lead to the best rates of progress toward optimum solutions.

11.5.2 Meta-Evolution on Real-Valued Parameters

Given the limited utility of the 1/5 rule, more general methods for controlling the degree to which variation operators act coarsely or finely are desired. A common method employed in evolutionary algorithms views the parameters that control the evolutionary search as part of the evolutionary process to be adapted while searching for an optimum. This poses a form of *meta-evolution* in that adaptation takes place on two levels within the evolutionary algorithms: the parameters that are used to evaluate the objective function (*objective parameters*) and the parameters that are used to control variation (*strategy parameters*).

The history of meta-evolutionary approaches extends back to the 1960s [Reed *et al.*, 1967; Rosenberg, 1967; Rechenberg, 1996[16]]. Currently, for real-valued evolutionary optimization, the most common approach follows a procedure offered in Schwefel [1981]. With the problem of minimizing (maximizing) a real-valued function $f(\mathbf{x})$, encode a possible solution as $(\mathbf{x}, \boldsymbol{\sigma})$, where $\mathbf{x}$ is the vector of real-valued objective parameters and $\boldsymbol{\sigma}$ is the vector of strategy parameters, which designate the positive standard deviation to apply a Gaussian mutation in the given dimension.

[16] In personal communication with one of the authors (DF), Ingo Rechenberg remarked that he devised an unpublished self-adaptive approach for evolution strategies in 1967.

Each parent creates an offspring via a two-step process applied across all dimensions, $i = 1, \ldots, n$:

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

where σ_i' is the standard deviation for the offspring in the ith dimension, $N(0, 1)$ is a standard Gaussian random variable sampled once and held at the same value for all i dimensions, $N_i(0, 1)$ is a standard Gaussian random variable sampled anew for each of the i dimensions, τ and τ' are constants that are proportional to $1/\sqrt{2\sqrt{n}}$ and $1/\sqrt{2n}$, respectively, and x_i' is the objective parameter value for the offspring in the ith dimension.

In this process, the standard deviations used in mutating a parent are carried as "genetic" information along with the parent solution. The first step mutates the standard deviations using a lognormal distribution. The second step uses the mutated standard deviations to create the offspring's objective parameters. In this way, standard deviations that lead to improved solutions are retained by selection, while those that lead to solutions of lesser quality perish. In essence, the procedure uses a form of reinforcement learning to update the strategy parameters of the evolutionary search while the search is in progress.

The two-step process above allows the evolutionary algorithm to learn how to shape the Gaussian distribution to better fit the contours of the objective function in the neighborhood of each solution. In this process, the dimensions are treated independently; however, extensions to self-adapting correlated mutations have also been offered in Schwefel [1981] (and for further discussion, see Fogel [2006]). Figure 11.16 provides an

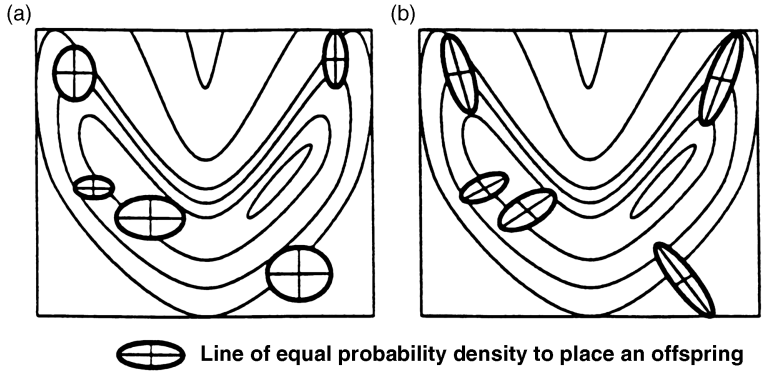

FIGURE 11.16 Self-adapting the mutation distributions on each solution allows new trials to be generated in light of the contours of the response surface. Independent adjustment of the standard deviation in each dimension provides a mechanism for varying the width of the probability contour in alignment with each axis (a). Correlated standard deviations provide a mechanism for generating trials such that the probability contours are not aligned with the coordinate axes (b). (Taken from Fogel [Fogel, 2000, p. 157] and Back *et al.* [1991].)

intuitive picture of what the self-adaptive process can accomplish. In addition, a common form of modern self-adaptation simplifies step (11.3) to be

$$\sigma_i' = \sigma_i \exp(\tau' N_i(0,1)) \tag{11.3}$$

foregoing the use of a "global" Gaussian random variable and relying solely on independent Gaussian random variables for each ith dimension.

11.5.3 Meta-Evolution on Probabilities of Variation Operators

The concept of meta-evolution can be extended to support variation operators that are not in the continuous domain. Self-adaptation can be applied both within a variation operator (controlling how that operator functions) and to determine the likelihood or number of times the variation operator is applied.

For example, when considering a traveling salesman problem, Chellapilla and Fogel [1997b] utilized a permutation representation (an ordered list of cities to visit) and applied self-adaptation to adjust either (i) the length of an inversion operation or (ii) the probability of applying every possible inversion from length 1 to $c/2 - 1$, where there were c cities. Angeline *et al.* [1996] used self-adaptation to adjust the mutation probabilities of varying finite-state machines.

Thus, self-adaptation is useful potentially not only for applications of evolutionary optimization in $\Re^n$ but also in virtually any real-world application. It can be used to adjust the likelihood of using mutation, crossover, blending recombination, multi-parent recombination, as well as the types of these operators (e.g., one-point, n-point, uniform crossovers). It is extremely unlikely that any static settings for variation operators will be optimal for solving a problem, and the effectiveness of different variation operators changes during the course of the evolution.

11.5.4 Meta-Evolution on Combinations of Variation Operators

An additional extension of self-adaptive meta-evolutionary methods comes in blending different operators. For example, as mentioned in Section 11.3.4.1, in a real-valued evolutionary optimization, there is a trade-off between using Gaussian variation and Cauchy variation. Instead of hand-tuning the probabilities of using these operators, their effects can be adjusted via self-adaptation, implemented potentially as follows:

(1) $\sigma_i' = \sigma_i \exp(\tau' N_i(0,1))$	//step-size control for the Gaussian component
(2) $s_i' = s_i \exp(\tau N_i(0,1))$	//step-size control for the Cauchy component
(3) $\alpha' = \alpha + N(0,\omega)$	//weight to apportion between Gaussian and Cauchy
(4) $x_i' = x_i + \lvert\sin(\alpha')\rvert N(0,\sigma_i') + (1 - \lvert\sin(\alpha')\rvert) C(0,s_i')$	//update with mutation in part by Gaussian and Cauchy components

where σ_i' and s_i' are the offspring scaling parameters for the Gaussian- and Cauchy-distributed mutations, respectively, τ and τ' are scaling constants, α' is the

offspring's adjustment factor for weighting the effects of Gaussian and Cauchy mutations, ω is a stepsize parameter for mutating α, and $N(0, \sigma)$ and $C(0, s)$ represent Gaussian and Cauchy distributions centered at zero with scaling parameters σ and s, respectively. Using the transform $|\sin(\alpha')|$ returns a value between 0 and 1 and thus weights the contribution of the Gaussian and Cauchy components of the overall variation.[17]

11.5.5 Fitness Distributions of Variation Operators

Applying a variation operator to a parent or set of parents generates offspring based on a probability mass or density function. Choosing among variation operators and changing the parameters of those operators alters the likelihood of sampling each point from the solution space. Thus, a probabilistic distribution of offspring fitness scores can be created given a variation operator and the parent or parents that it operates on. In certain simple cases, this *fitness distribution* can be computed mathematically (e.g., leading to Rechenberg's 1/5 rule), but it can be estimated empirically in general cases. Doing so is computationally intensive, as it requires generating a sufficient number of examples of applying an operator to a parent(s) and determining the results statistically; however, the fitness distribution of an operator can yield insight into setting its parameters, how to make them subject to self-adaptation, and whether or not to include them.

For example, Nordin and Banzhaf [1995] quantified the change in fitness that occurred after applying crossover to machine code expressions that represented regression equations. Figure 11.17 shows the histogram of the percentage of fitness change over 35 successive generations. Positive change denotes an improvement. The figure shows that the most common result was either no change in fitness at all (i.e., the offspring's fitness was the same as the parent's fitness) or a 100% decrease in fitness. In this case, the success rate of crossover was very small, leading to the suggestion that in this case (and others [Nordin *et al*., 1996; Teller, 1996]) crossover was acting as an ineffective macromutation [Banzhaf *et al*., 1998].

Fogel and Jain [2000] showed that for the case of evolving a neural network to perform the XOR function, the fitness distributions of Gaussian mutation and one-point crossover changed as a function of the progress of the evolution. At certain times, the expected progress was the same regardless of which operator was chosen; at others, mutation or crossover was favored. For additional background on fitness distributions, see Altenberg [1995], Grefenstette [1995], Fogel and Ghozeil [1996], and Fogel [2006].

[17] The process suggested here is untested and provides an open area for research, including self-adaptation of ω. See Saravanan and Fogel [1997] and Chellapilla and Fogel [1997a] for other methods of adapting the use of Gaussian and Cauchy variation operators. As an aside, it is easy to imagine applying this form of adaptation to neural networks or fuzzy systems in evolving weights on transfer or membership functions. For example, a neural network can have nodes that are flexible between sigmoid (SIG) and radial-basis functions (RBF) by adapting a parameter α and utilizing it in the manner of $\alpha\text{SIG} + (1 - \alpha)\text{RBF}$.

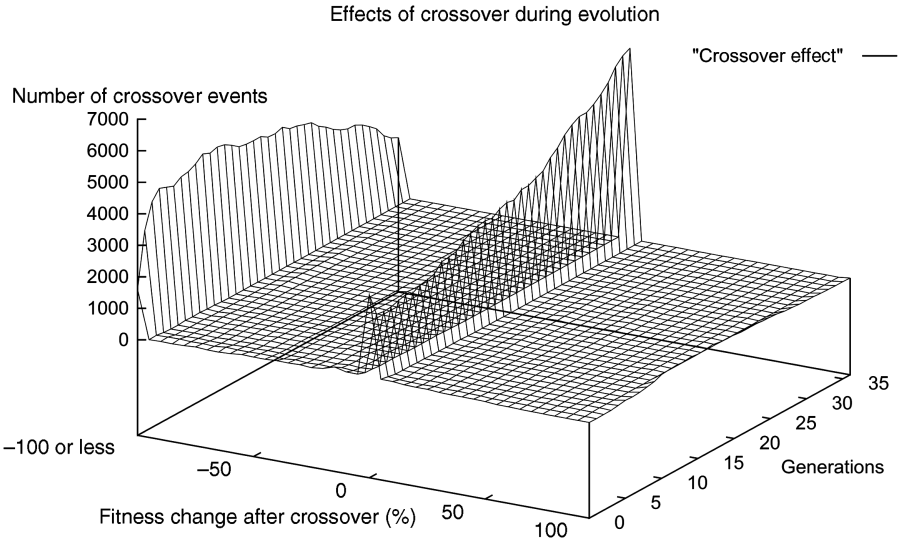

FIGURE 11.17 The fitness change after crossover on problems evolving machine code expressions from Nordin and Banzhaf [1995]. The vast majority of crossovers either resulted in no change in fitness or a large decrease in offspring fitness.

11.6 SUMMARY

Evolutionary algorithms have a long history of being used for optimization [Fogel, 1998]. These procedures have been applied to a wide array of problems in different areas: drug design, finance, video games, robotics, pattern recognition, scheduling, and many more. This chapter has provided an introduction to the application of evolutionary algorithms for optimizing solutions to problems in their canonical forms, including real-valued optimization, combinatorial optimization, and hybrids of these.

The essence of the evolutionary approach to optimization is to utilize a population of contending ideas about how to solve a problem and subject those ideas to random variation and selection in light of objective criteria that provide feedback on the suitability of the solutions. Determining which solutions to maintain as parents follows the concept of natural selection, eliminating the least-fit candidate solutions from the population probabilistically.

Representation, selection, and variation act in concert. There is no best choice to make for these design variables outside of the context of the problem at hand [Wolpert and Macready, 1997; see also English [1996]]. All canonical forms of variation—Gaussian mutation, one-point crossover, binary mutation, recombination by blending—serve not as ends but as beginnings for the evolutionary algorithmist (you!) who must design an effective search of a solution space. Self-adaptive methods can assist in allowing the evolutionary algorithm to learn how to search the solution space via reinforcement of the strategies that have already found better solutions.

It is important to maintain an open mind when designing an evolutionary algorithm. Simple traditional approaches may be sufficient, and if so, they should be used. But imaginative thinking can lead to interesting new approaches (as with the use of "geometric recombination" noted in Section 11.4). For example, consider the problem of optimizing for a solution in $\Re^n$ that minimizes $f(\mathbf{x})$, except that the representation is made in polar coordinates instead of Cartesian coordinates (see Ghozeil and Fogel [1996]). Or consider the problem of designing variation operators that can effectively jump from one locally optimal region directly to another in a given application. These sorts of activities may require a fundamental understanding of the properties of the "landscape" posed by the objective function in light of the variation operators [Jones, 1995b]. (Variation operators define the search neighborhood of each parent regardless of the "shape" of the landscape being searched.) Each of these requires specific investigation and standard approaches as found commonly in literature may not be effective.

Evolutionary algorithms offer opportunities for hybridizing with other methods. For example, an evolutionary algorithm can be run for a number of generations and then a gradient method can be used to approach a local optimum more quickly. Evolutionary algorithms also offer significant flexibility in being able to handle problems with constraints (both hard and soft), with multiple criteria. The chapter has not treated the application of evolutionary algorithms to dynamic and noisy environments. Some examples are provided in Chapter 14. It has also not treated many of the mathematical properties of evolutionary algorithms with the depth that would be found in a textbook dedicated solely for that purpose. Students of evolutionary algorithms who would like a more detailed mathematical treatment should refer to Bäck [1996], Vose [1999], Fogel [2006], and Beyer [2010].

EXERCISES

11.1. Write an evolutionary program that addresses the traveling salesman problem as described in Section 11.2, except that instead of evaluating solutions based on minimizing total length, try the following two objectives: (1) minimize total length but have a large penalty for crossing $x = 0$, and (2) minimize total length but have a large bonus for crossing $x = 0$. You can use any evolutionary approach you like. Have a look at the solutions that emerge for each variation. Do they each reflect the varied objective of the problem?

11.2. Try another variation of the traveling salesman problem in which you designate a specific city as the starting city and all segments that are traversed from north to south (a higher y-coordinate to a lower y-coordinate) cost 1.5 times the distance of the segment. Do the solutions you generate appear different than those that do not incorporate this penalty?

11.3. Work with Rastrigin's function as described in Section 11.1. Choose an evolutionary algorithm that uses a real-valued representation and zero-mean Gaussian mutations. Implement self-adaptation as described in Section 11.5.2

with the simplified version of updating the standard deviations:

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

using $\tau' = 1/\sqrt{2n}$. Explore the rate of optimization on Rastrigin's function in 2 and 10 dimensions as a function of the population size when trying a $(\mu + 5\mu)$ configuration. For different values of μ, plot the best score attained as a function of the number of generations.

11.4. For the same problem as in exercise 11.3, compare the use of self-adaptation to the 1/5 rule (Section 11.5.1). Does the 1/5 rule provide for faster convergence to better solutions on Rastrigin's function in 2 and 10 dimensions than does simple self-adaptation?

11.5. For the problem illustrated in Section 11.3.4.4, determine the fitness distribution of various operators as follows. Start with a real-valued representation, initialization between $[-5, 5]$ in each dimension, and zero-mean Gaussian mutation with a standard deviation of 0.33 with 100 parents making 100 offspring (as in the text). Run the algorithm for 10 generations and then focus on the best parent. Empirically, determine what fitness values would result from mutating this parent with a zero-mean Gaussian mutation using different fixed standard deviations for x and y. Then compare the best Gaussian mutation that you find with a Cauchy distribution. (You can compute a standard Cauchy distribution by taking N_1/N_2 where N_1 and N_2 are independent $N(0, 1)$ random variables.)

11.6. Now try the same problem, but using a binary encoding as shown in the text. Using one-point crossover and a probability of mutation = 0.01/bit, stop the evolution at generation 10 and then determine the fitness distribution of one-point, two-point, and uniform crossovers by selecting the best parent and a random other parent from the population. Does one recombination operator provide a better fitness distribution than another?

11.7. Data for total sunspots are available widely on the Internet. Acquire the data for total sunspots for 2000–2013 and plot the data as a function of year (sunspots on the y-axis, year on the x-axis). Determine the best linear equation to model these data in the form of

$$y = ax + b$$

where a and b are evolved via any evolutionary algorithm that you choose in order to minimize the squared error between each predicted value $(ax + b)$ and the actual value y. Using standard regression software, check how close your values for a and b come to the optimal values. Now try evolving a and b so as to minimize the absolute value of the cube of the error. How much different are a and b from before?

11.8. Develop code structures for finite-state machines and symbolic expressions, as described in Section 11.3.4.3, including variation operators. (You'll need these in Chapter 12.) Test your code by evolving finite-state machines to generate a sequence [100111001] and a symbolic expression to match $x^2 + 2x + 1$. You can use any objective functions that you believe will help the evolution get to the desired outcome.

11.9. For the problem illustrated in Section 11.3.4.4, let's impose a constraint that $x < y - 1$. The domain of (x, y) is still $[-5, 5]$, but now the only feasible candidates for (x, y) must satisfy the constraint. Create a penalty method for addressing this constrained problem and apply an evolutionary algorithm of your choice. How well does it do? Try at least 30 independent trials to assess your results.

11.10. Explain why premature convergence is not necessarily to a point that is locally optima in a continuous space. (That is, the population may converge, but there would be a gradient that would suggest moving to a better point.) When is this more or less likely in evolutionary optimization?

11.11. Indicate the conditions that allow an evolutionary algorithm to converge with probability 1 to a global optimum solution.

11.12. Explain the benefits and drawbacks of self-adaptation in evolutionary algorithms, in terms of adjusting standard deviations in both continuous search and operator probabilities generally.