1 A Gentle Introduction to Genetic Algorithms

In this chapter, we introduce genetic algorithms: what they are, where they came from, and how they compare to and differ from other search procedures. We illustrate how they work with a hand calculation, and we start to understand their power through the concept of a schema or similarity template.

WHAT ARE GENETIC ALGORITHMS?

Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics. They combine survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm with some of the innovative flair of human search. In every generation, a new set of artificial creatures (strings) is created using bits and pieces of the fittest of the old; an occasional new part is tried for good measure. While randomized, genetic algorithms are no simple random walk. They efficiently exploit historical information to speculate on new search points with expected improved performance.

Genetic algorithms have been developed by John Holland, his colleagues, and his students at the University of Michigan. The goals of their research have been twofold: (1) to abstract and rigorously explain the adaptive processes of natural systems, and (2) to design artificial systems software that retains the important mechanisms of natural systems. This approach has led to important discoveries in both natural and artificial systems science.

The central theme of research on genetic algorithms has been *robustness*, the balance between efficiency and efficacy necessary for survival in many different environments. The implications of robustness for artificial systems are manifold. If artificial systems can be made more robust, costly redesigns can be reduced or eliminated. If higher levels of adaptation can be achieved, existing systems can perform their functions longer and better. Designers of artificial systems—both software and hardware, whether engineering systems, computer systems, or business systems—can only marvel at the robustness, the efficiency, and the flexibility of biological systems. Features for self-repair, self-guidance, and reproduction are the rule in biological systems, whereas they barely exist in the most sophisticated artificial systems.

Thus, we are drawn to an interesting conclusion: where robust performance is desired (and where is it not?), nature does it better; the secrets of adaptation and survival are best learned from the careful study of biological example. Yet we do not accept the genetic algorithm method by appeal to this beauty-of-nature argument alone. Genetic algorithms are theoretically and empirically proven to provide robust search in complex spaces. The primary monograph on the topic is Holland's (1975) *Adaptation in Natural and Artificial Systems.* Many papers and dissertations establish the validity of the technique in function optimization and control applications. Having been established as a valid approach to problems requiring efficient and effective search, genetic algorithms are now finding more widespread application in business, scientific, and engineering circles. The reasons behind the growing numbers of applications are clear. These algorithms are computationally simple yet powerful in their search for improvement. Furthermore, they are not fundamentally limited by restrictive assumptions about the search space (assumptions concerning continuity, existence of derivatives, unimodality, and other matters). We will investigate the reasons behind these attractive qualities; but before this, we need to explore the robustness of more widely accepted search procedures.

ROBUSTNESS OF TRADITIONAL OPTIMIZATION AND SEARCH METHODS

This book is not a comparative study of search and optimization techniques. Nonetheless, it is important to question whether conventional search methods meet our robustness requirements. The current literature identifies three main types of search methods: calculus-based, enumerative, and random. Let us examine each type to see what conclusions may be drawn without formal testing.

Calculus-based methods have been studied heavily. These subdivide into two main classes: indirect and direct. Indirect methods seek local extrema by solving the usually nonlinear set of equations resulting from setting the gradient of the objective function equal to zero. This is the multidimensional generalization of the elementary calculus notion of extremal points, as illustrated in Fig. 1.1. Given a smooth, unconstrained function, finding a possible peak starts by restricting search to those points with slopes of zero in all directions. On the other hand,

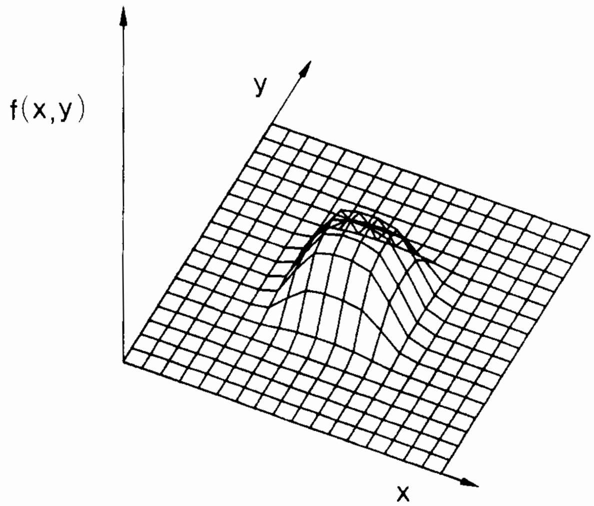

FIGURE 1.1 **The single-peak function is easy for calculus-based methods.**

direct (search) methods seek local optima by hopping on the function and moving in a direction related to the local gradient. This is simply the notion of ***hill-climbing:*** to find the local best, climb the function in the steepest permissible direction. While both of these calculus-based methods have been improved, extended, hashed, and rehashed, some simple reasoning shows their lack of robustness.

First, both methods are local in scope; the optima they seek are the best in a neighborhood of the current point. For example, suppose that Fig. 1.1 shows a portion of the complete domain of interest; a more complete picture is shown in Fig. 1.2. Clearly, starting the search or zero-finding procedures in the neighborhood of the lower peak will cause us to miss the main event (the higher peak). Furthermore, once the lower peak is reached, further improvement must be sought through random restart or other trickery. Second, calculus-based methods depend upon the existence of derivatives (well-defined slope values). Even if we allow numerical approximation of derivatives, this is a severe shortcoming. Many practical parameter spaces have little respect for the notion of a derivative and the smoothness this implies. Theorists interested in optimization have been too willing to accept the legacy of the great eighteenth and nineteenth-century mathematicians who painted a clean world of quadratic objective functions, ideal constraints, and ever present derivatives. The real world of search is fraught with discontinuities and vast multimodal, noisy search spaces as depicted in a less calculus-friendly function in Fig. 1.3. It comes as no surprise that methods depending upon the restrictive requirements of continuity and derivative existence are unsuitable for all but a very limited problem domain. For this reason and

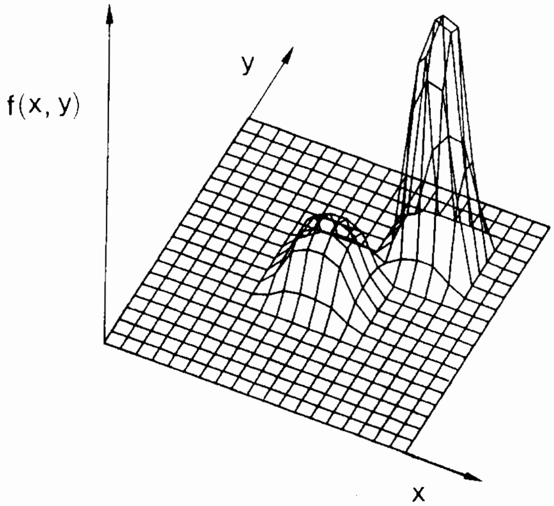

FIGURE 1.2 **The multiple-peak function causes a dilemma. Which hill should we climb?**

because of their inherently local scope of search, we must reject calculus-based methods. They are insufficiently robust in unintended domains.

Enumerative schemes have been considered in many shapes and sizes. The idea is fairly straightforward; within a finite search space, or a discretized infinite search space, the search algorithm starts looking at objective function values at every point in the space, one at a time. Although the simplicity of this type of

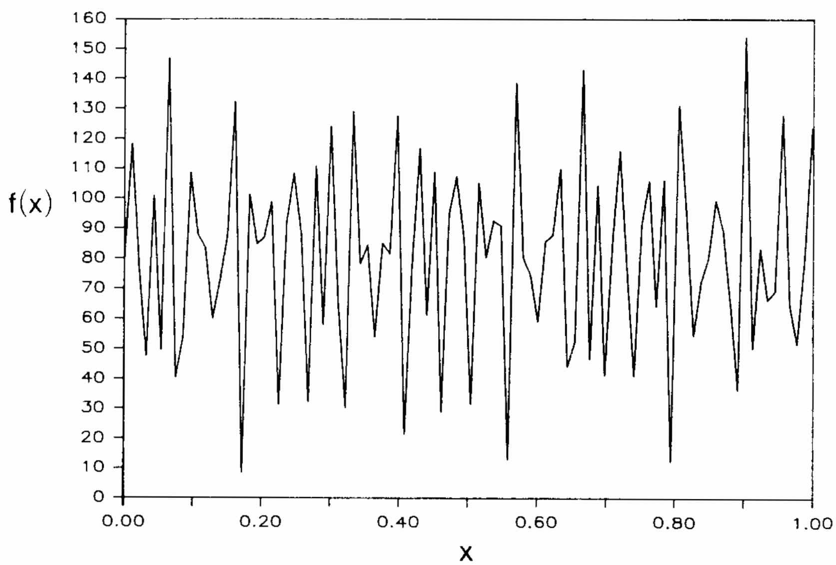

FIGURE 1.3 **Many functions are noisy and discontinuous and thus unsuitable for search by traditional methods.**

algorithm is attractive, and enumeration is a very human kind of search (when the number of possibilities is small), such schemes must ultimately be discounted in the robustness race for one simple reason: lack of efficiency. Many practical spaces are simply too large to search one at a time and still have a chance of using the information to some practical end. Even the highly touted enumerative scheme *dynamic programming* breaks down on problems of moderate size and complexity, suffering from a malady melodramatically labeled the "curse of dimensionality" by its creator (Bellman, 1961). We must conclude that less clever enumerative schemes are similarly, and more abundantly, cursed for real problems.

Random search algorithms have achieved increasing popularity as researchers have recognized the shortcomings of calculus-based and enumerative schemes. Yet, random walks and random schemes that search and save the best must also be discounted because of the efficiency requirement. Random searches, in the long run, can be expected to do no better than enumerative schemes. In our haste to discount strictly random search methods, we must be careful to separate them from randomized techniques. The genetic algorithm is an example of a search procedure that uses random choice as a tool to guide a highly exploitative search through a coding of a parameter space. Using random choice as a tool in a directed search process seems strange at first, but nature contains many examples. Another currently popular search technique, *simulated annealing,* uses random processes to help guide its form of search for minimal energy states. A recent book (Davis, 1987) explores the connections between simulated annealing and genetic algorithms. The important thing to recognize at this juncture is that randomized search does not necessarily imply directionless search.

While our discussion has been no exhaustive examination of the myriad methods of traditional optimization, we are left with a somewhat unsettling conclusion: conventional search methods are not robust. This does not imply that they are not useful. The schemes mentioned and countless hybrid combinations and permutations have been used successfully in many applications; however, as more complex problems are attacked, other methods will be necessary. To put this point in better perspective, inspect the problem spectrum of Fig. 1.4. In the figure a mythical effectiveness index is plotted across a problem continuum for a specialized scheme, an enumerative scheme, and an idealized robust scheme. The gradient technique performs well in its narrow problem class, as we expect, but it becomes highly inefficient (if useful at all) elsewhere. On the other hand, the enumerative scheme performs with egalitarian inefficiency across the spectrum of problems, as shown by the lower performance curve. Far more desirable would be a performance curve like the one labeled Robust Scheme. It would be worthwhile sacrificing peak performance on a particular problem to achieve a relatively high level of performance across the spectrum of problems. (Of course, with broad, efficient methods we can always create hybrid schemes that combine the best of the local search method with the more general robust scheme. We will have more to say about this possibility in Chapter 5.) We shall soon see how genetic algorithms help fill this robustness gap.

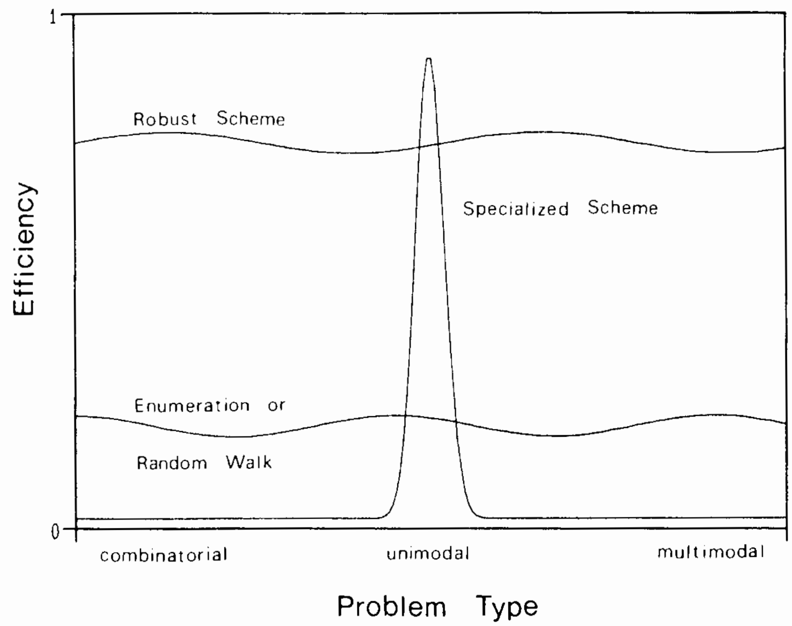

FIGURE 1.4 Many traditional schemes work well in a narrow problem domain. Enumerative schemes and random walks work equally inefficiently across a broad spectrum. A robust method works well across a broad spectrum of problems.

THE GOALS OF OPTIMIZATION

Before examining the mechanics and power of a simple genetic algorithm, we must be clearer about our goals when we say we want to optimize a function or a process. What are we trying to accomplish when we optimize? The conventional view is presented well by Beightler, Phillips, and Wilde (1979, p. 1):

> Man's longing for perfection finds expression in the theory of optimization. It studies how to describe and attain what is Best, once one knows how to measure and alter what is Good or Bad. . . . *Optimization theory* encompasses the quantitative study of optima and methods for finding them.

Thus optimization seeks to improve performance toward some optimal point or points. Note that this definition has two parts: (1) we seek improvement to approach some (2) optimal point. There is a clear distinction between the *process* of improvement and the *destination* or optimum itself. Yet, in judging optimization procedures we commonly focus solely upon convergence (does the method reach the optimum?) and forget entirely about interim performance. This emphasis stems from the origins of optimization in the calculus. It is not, however, a natural emphasis.

Consider a human decision maker, for example, a businessman. How do we judge his decisions? What criteria do we use to decide whether he has done a good or bad job? Usually we say he has done well when he makes adequate selections within the time and resources allotted. Goodness is judged relative to his competition. Does he produce a better widget? Does he get it to market more efficiently? With better promotion? We never judge a businessman by an attainment-of-the-best criterion; perfection is all too stern a taskmaster. As a result, we conclude that convergence to the best is not an issue in business or in most walks of life; we are only concerned with doing better relative to others. Thus, if we want more humanlike optimization tools, we are led to a reordering of the priorities of optimization. The most important goal of optimization is improvement. Can we get to some good, "satisficing" (Simon, 1969) level of performance quickly? Attainment of the optimum is much less important for complex systems. It would be nice to be perfect: meanwhile, we can only strive to improve. In the next chapter we watch the genetic algorithm for these qualities; here we outline some important differences between genetic algorithms and more traditional methods.

HOW ARE GENETIC ALGORITHMS DIFFERENT FROM TRADITIONAL METHODS?

In order for genetic algorithms to surpass their more traditional cousins in the quest for robustness, GAs must differ in some very fundamental ways. Genetic algorithms are different from more normal optimization and search procedures in four ways:

1. GAs work with a coding of the parameter set, not the parameters themselves.
2. GAs search from a population of points, not a single point.
3. GAs use payoff (objective function) information, not derivatives or other auxiliary knowledge.
4. GAs use probabilistic transition rules, not deterministic rules.

Genetic algorithms require the natural parameter set of the optimization problem to be coded as a finite-length string over some finite alphabet. As an example, consider the optimization problem posed in Fig. 1.5. We wish to maximize the function $f(x) = x^2$ on the integer interval $[0, 31]$. With more traditional methods we would be tempted to twiddle with the parameter x, turning it like the vertical hold knob on a television set, until we reached the highest objective function value. With GAs, the first step of our optimization process is to code the parameter x as a finite-length string. There are many ways to code the x parameter, and Chapter 3 examines some of these in detail. At the moment, let's consider an optimization problem where the coding comes a bit more naturally.

Consider the black box switching problem illustrated in Fig. 1.6. This problem concerns a black box device with a bank of five input switches. For every setting of the five switches, there is an output signal f, mathematically $f = f(s)$,

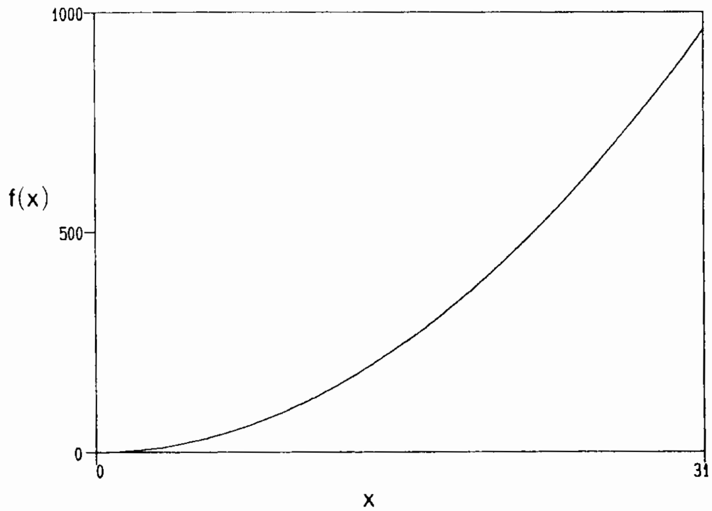

FIGURE 1.5 **A simple function optimization example, the function $f(x) = x^2$ on the integer interval [0, 31].**

where s is a particular setting of the five switches. The objective of the problem is to set the switches to obtain the maximum possible f value. With other methods of optimization we might work directly with the parameter set (the switch settings) and toggle switches from one setting to another using the transition rules of our particular method. With genetic algorithms, we first code the switches as a finite-length string. A simple code can be generated by considering a string of five 1's and 0's where each of the five switches is represented by a 1 if the switch is on and a 0 if the switch is off. With this coding, the string 11110 codes the setting where the first four switches are on and the fifth switch is off. Some of the codings introduced later will not be so obvious, but at this juncture we acknowledge that genetic algorithms use codings. Later it will be apparent

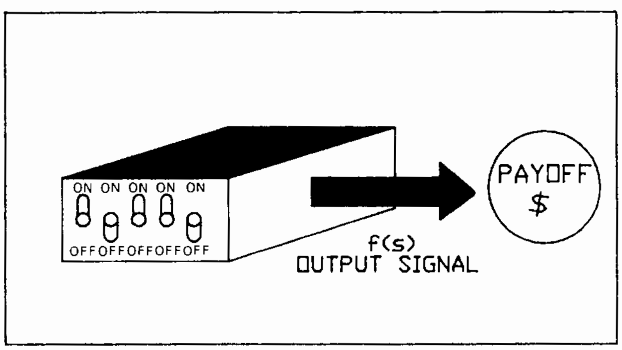

FIGURE 1.6 **A black box optimization problem with five on-off switches illustrates the idea of a coding and a payoff measure. Genetic algorithms only require these two things: they don't need to know the workings of the black box.**

that genetic algorithms exploit coding similarities in a very general way; as a result, they are largely unconstrained by the limitations of other methods (continuity, derivative existence, unimodality, and so on).

In many optimization methods, we move gingerly from a single point in the decision space to the next using some transition rule to determine the next point. This point-to-point method is dangerous because it is a perfect prescription for locating false peaks in multimodal (many-peaked) search spaces. By contrast, GAs work from a rich database of points simultaneously (a population of strings), climbing many peaks in parallel; thus, the probability of finding a false peak is reduced over methods that go point to point. As an example, let's consider our black box optimization problem (Fig. 1.6) again. Other techniques for solving this problem might start with one set of switch settings, apply some transition rules, and generate a new trial switch setting. A genetic algorithm starts with a population of strings and thereafter generates successive populations of strings. For example, in the five-switch problem, a random start using successive coin flips (head = 1, tail = 0) might generate the initial population of size $n = 4$ (small by genetic algorithm standards):

```
01101
11000
01000
10011
```

After this start, successive populations are generated using the genetic algorithm. By working from a population of well-adapted diversity instead of a single point, the genetic algorithm adheres to the old adage that there is safety in numbers; we will soon see how this parallel flavor contributes to a genetic algorithm's robustness.

Many search techniques require much auxiliary information in order to work properly. For example, gradient techniques need derivatives (calculated analytically or numerically) in order to be able to climb the current peak, and other local search procedures like the greedy techniques of combinatorial optimization (Lawler, 1976; Syslo, Deo, and Kowalik, 1983) require access to most if not all tabular parameters. By contrast, genetic algorithms have no need for all this auxiliary information: GAs are blind. To perform an effective search for better and better structures, they only require payoff values (objective function values) associated with individual strings. This characteristic makes a GA a more canonical method than many search schemes. After all, every search problem has a metric (or metrics) relevant to the search; however, different search problems have vastly different forms of auxiliary information. Only if we refuse to use this auxiliary information can we hope to develop the broadly based schemes we desire. On the other hand, the refusal to use specific knowledge when it does exist can place an upper bound on the performance of an algorithm when it goes head to head with methods designed for that problem. Chapter 5 examines ways to use nonpayoff information in so-called knowledge-directed genetic algorithms; however, at this juncture we stress the importance of the blindness assumption to pure genetic algorithm robustness.

Unlike many methods, GAs use probabilistic transition rules to guide their search. To persons familiar with deterministic methods this seems odd, but the use of probability does not suggest that the method is some simple random search; this is not decision making at the toss of a coin. Genetic algorithms use random choice as a tool to guide a search toward regions of the search space with likely improvement.

Taken together, these four differences—direct use of a coding, search from a population, blindness to auxiliary information, and randomized operators—contribute to a genetic algorithm's robustness and resulting advantage over other more commonly used techniques. The next section introduces a simple three-operator genetic algorithm.

A SIMPLE GENETIC ALGORITHM

The mechanics of a simple genetic algorithm are surprisingly simple, involving nothing more complex than copying strings and swapping partial strings. The explanation of why this simple process works is much more subtle and powerful. Simplicity of operation and power of effect are two of the main attractions of the genetic algorithm approach.

The previous section pointed out how genetic algorithms process populations of strings. Recalling the black box switching problem, remember that the initial population had four strings:

```
01101
11000
01000
10011
```

Also recall that this population was chosen at random through 20 successive flips of an unbiased coin. We now must define a set of simple operations that take this initial population and generate successive populations that (we hope) improve over time.

A simple genetic algorithm that yields good results in many practical problems is composed of three operators:

1. Reproduction
2. Crossover
3. Mutation

Reproduction is a process in which individual strings are copied according to their objective function values, f (biologists call this function the fitness function). Intuitively, we can think of the function f as some measure of profit, utility, or goodness that we want to maximize. Copying strings according to their fitness values means that strings with a higher value have a higher probability of contributing one or more offspring in the next generation. This operator, of course, is an artificial version of natural selection, a Darwinian survival of the fittest

TABLE 1.1 Sample Problem Strings and Fitness Values

No.	String	Fitness	% of Total
1	01101	169	14.4
2	11000	576	49.2
3	01000	64	5.5
4	10011	361	30.9
Total		1170	100.0

among string creatures. In natural populations fitness is determined by a creature's ability to survive predators, pestilence, and the other obstacles to adulthood and subsequent reproduction. In our unabashedly artificial setting, the objective function is the final arbiter of the string-creature's life or death.

The reproduction operator may be implemented in algorithmic form in a number of ways. Perhaps the easiest is to create a biased roulette wheel where each current string in the population has a roulette wheel slot sized in proportion to its fitness. Suppose the sample population of four strings in the black box problem has objective or fitness function values f as shown in Table 1.1 (for now we accept these values as the output of some unknown and arbitrary black box—later we will examine a function and coding that generate these same values).

Summing the fitness over all four strings, we obtain a total of 1170. The percentage of population total fitness is also shown in the table. The corresponding weighted roulette wheel for this generation's reproduction is shown in Fig. 1.7. To reproduce, we simply spin the weighted roulette wheel thus defined four times. For the example problem, string number 1 has a fitness value of 169, which represents 14.4 percent of the total fitness. As a result, string 1 is given 14.4 percent of the biased roulette wheel, and each spin turns up string 1 with prob-

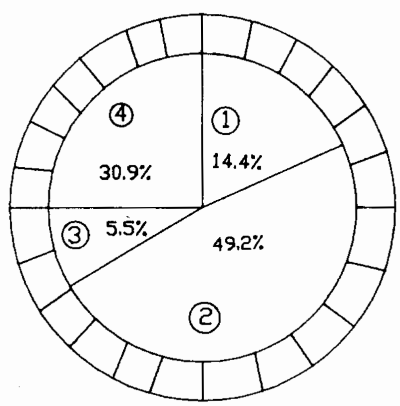

FIGURE 1.7 Simple reproduction allocates offspring strings using a roulette wheel with slots sized according to fitness. The sample wheel is sized for the problem of Tables 1.1 and 1.2.

ability 0.144. Each time we require another offspring, a simple spin of the weighted roulette wheel yields the reproduction candidate. In this way, more highly fit strings have a higher number of offspring in the succeeding generation. Once a string has been selected for reproduction, an exact replica of the string is made. This string is then entered into a mating pool, a tentative new population, for further genetic operator action.

After reproduction, simple crossover (Fig. 1.8) may proceed in two steps. First, members of the newly reproduced strings in the mating pool are mated at random. Second, each pair of strings undergoes crossing over as follows: an integer position k along the string is selected uniformly at random between 1 and the string length less one $[1, l - 1]$. Two new strings are created by swapping all characters between positions $k + 1$ and l inclusively. For example, consider strings A_1 and A_2 from our example initial population:

$$A_1 = 0\ 1\ 1\ 0 \mid 1$$
$$A_2 = 1\ 1\ 0\ 0 \mid 0$$

Suppose in choosing a random number between 1 and 4, we obtain a $k = 4$ (as indicated by the separator symbol $\mid$). The resulting crossover yields two new strings where the prime (′) means the strings are part of the new generation:

$$A'_1 = 0\ 1\ 1\ 0\ 0$$
$$A'_2 = 1\ 1\ 0\ 0\ 1$$

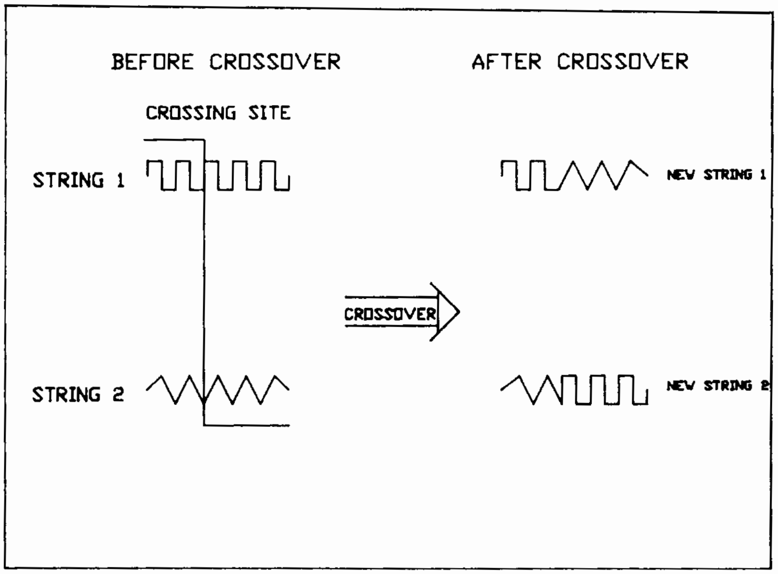

FIGURE 1.8 **A schematic of simple crossover shows the alignment of two strings and the partial exchange of information, using a cross site chosen at random.**

The mechanics of reproduction and crossover are surprisingly simple, involving random number generation, string copies, and some partial string exchanges. Nonetheless, the combined emphasis of reproduction and the structured, though randomized, information exchange of crossover give genetic algorithms much of their power. At first this seems surprising. How can two such simple (and computationally trivial) operators result in anything useful, let alone a rapid and robust search mechanism? Furthermore, doesn't it seem a little strange that chance should play such a fundamental role in a directed search process? We will examine a partial answer to the first of these two questions in a moment; the answer to the second question was well recognized by the mathematician J. Hadamard (1949, p. 29):

> We shall see a little later that the possibility of imputing discovery to pure chance is already excluded.... On the contrary, that there is an intervention of chance but also a necessary work of unconsciousness, the latter implying and not contradicting the former.... Indeed, it is obvious that invention or discovery, be it in mathematics or anywhere else, takes place by combining ideas.

Hadamard suggests that even though discovery is not a result—cannot be a result—of pure chance, it is almost certainly guided by directed serendipity. Furthermore, Hadamard hints that a proper role for chance in a more humanlike discovery mechanism is to cause the juxtaposition of different notions. It is interesting that genetic algorithms adopt Hadamard's mix of direction and chance in a manner that efficiently builds new solutions from the best partial solutions of previous trials.

To see this, consider a population of n strings (perhaps the four-string population for the black box problem) over some appropriate alphabet, coded so that each is a complete *idea* or prescription for performing a particular task (in this case, each string is one complete switch-setting idea). Substrings within each string (idea) contain various ***notions*** of what is important or relevant to the task. Viewed in this way, the population contains not just a sample of n ideas; rather, it contains a multitude of notions and rankings of those notions for task performance. Genetic algorithms ruthlessly exploit this wealth of information by (1) reproducing high-quality notions according to their performance and (2) crossing these notions with many other high-performance notions from other strings. Thus, the action of crossover with previous reproduction speculates on new ideas constructed from the high-performance building blocks (notions) of past trials. In passing, we note that despite the somewhat fuzzy definition of a notion, we have not limited a notion to simple linear combinations of single features or pairs of features. Biologists have long recognized that evolution must efficiently process the epistasis (positionwise nonlinearity) that arises in nature. In a similar manner, the notion processing of genetic algorithms must effectively process notions even when they depend upon their component features in highly nonlinear and complex ways.

Exchanging of notions to form new ideas is appealing intuitively, if we think in terms of the process of *innovation.* What is an innovative idea? As Hadamard suggests, most often it is a juxtaposition of things that have worked well in the past. In much the same way, reproduction and crossover combine to search potentially pregnant new ideas. This experience of emphasis and crossing is analogous to the human interaction many of us have observed at a trade show or scientific conference. At a widget conference, for example, various widget experts from around the world gather to discuss the latest in widget technology. After the lecture sessions, they all pair off around the bar to exchange widget stories. Well-known widget experts, of course, are in greater demand and exchange more ideas, thoughts, and notions with their lesser known widget colleagues. When the show ends, the widget people return to their widget laboratories to try out a surfeit of widget innovations. The process of reproduction and crossover in a genetic algorithm is this kind of exchange. High-performance notions are repeatedly tested and exchanged in the search for better and better performance.

If reproduction according to fitness combined with crossover gives genetic algorithms the bulk of their processing power, what then is the purpose of the mutation operator? Not surprisingly, there is much confusion about the role of mutation in genetics (both natural and artificial). Perhaps it is the result of too many B movies detailing the exploits of mutant eggplants that consume mass quantities of Tokyo or Chicago, but whatever the cause for the confusion, we find that mutation plays a decidedly secondary role in the operation of genetic algorithms. Mutation is needed because, even though reproduction and crossover effectively search and recombine extant notions, occasionally they may become overzealous and lose some potentially useful genetic material (1's or 0's at particular locations). In artificial genetic systems, the mutation operator protects against such an irrecoverable loss. In the simple GA, mutation is the occasional (with small probability) random alteration of the value of a string position. In the binary coding of the black box problem, this simply means changing a 1 to a 0 and vice versa. By itself, mutation is a random walk through the string space. When used sparingly with reproduction and crossover, it is an insurance policy against premature loss of important notions.

That the mutation operator plays a secondary role in the simple GA, we simply note that the frequency of mutation to obtain good results in empirical genetic algorithm studies is on the order of one mutation per thousand bit (position) transfers. Mutation rates are similarly small (or smaller) in natural populations, leading us to conclude that mutation is appropriately considered as a secondary mechanism of genetic algorithm adaptation.

Other genetic operators and reproductive plans have been abstracted from the study of biological example. However, the three examined in this section, reproduction, simple crossover, and mutation, have proved to be both computationally simple and effective in attacking a number of important optimization problems. In the next section, we perform a hand simulation of the simple genetic algorithm to demonstrate both its mechanics and its power.

GENETIC ALGORITHMS AT WORK—A SIMULATION BY HAND

Let's apply our simple genetic algorithm to a particular optimization problem step by step. Consider the problem of maximizing the function $f(x) = x^2$, where x is permitted to vary between 0 and 31, a function displayed earlier as Fig. 1.5. To use a genetic algorithm we must first code the decision variables of our problem as some finite-length string. For this problem, we will code the variable x simply as a binary unsigned integer of length 5. Before we proceed with the simulation, let's briefly review the notion of a binary integer. As decadigited creatures, we have little problem handling base 10 integers and arithmetic. For example, the five-digit number 53,095 may be thought of as

$$5 \cdot 10^4 + 3 \cdot 10^3 + 0 \cdot 10^2 + 9 \cdot 10^1 + 5 \cdot 1 = 53{,}095.$$

In base 2 arithmetic, we of course only have two digits to work with, 0 and 1, and as an example the number 10,011 decodes to the base 10 number

$$1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 2 + 1 = 19.$$

With a five-bit (*b*inary dig*it*) unsigned integer we can obtain numbers between 0 (00000) and 31 (11111). With a well-defined objective function and coding, we now simulate a single generation of a genetic algorithm with reproduction, crossover, and mutation.

To start off, we select an initial population at random. We select a population of size 4 by tossing a fair coin 20 times. We can skip this step by using the initial population created in this way earlier for the black box switching problem. Looking at this population, shown on the left-hand side of Table 1.2, we observe that the decoded x values are presented along with the fitness or objective function values $f(x)$. To make sure we know how the fitness values $f(x)$ are calculated from the string representation, let's take a look at the third string of the initial population, string 01000. Decoding this string as an unsigned binary integer, we note that there is a single one in the $2^3 = 8$'s position. Hence for string 01000 we obtain $x = 8$. To calculate the fitness or objective function we simply square the x value and obtain the resulting fitness value $f(x) = 64$. Other x and $f(x)$ values may be obtained similarly.

You may notice that the fitness or objective function values are the same as the black box values (compare Tables 1.1 and 1.2). This is no coincidence, and the black box optimization problem was well represented by the particular function, $f(x)$, and coding we are now using. Of course, the genetic algorithm need not know any of this; it is just as happy to optimize some arbitrary switching function (or any other finite coding and function for that matter) as some polynomial function with straightforward binary coding. This discussion simply reinforces one of the strengths of the genetic algorithm: by exploiting similarities in codings, genetic algorithms can deal effectively with a broader class of functions than can many other procedures.

A generation of the genetic algorithm begins with reproduction. We select the mating pool of the next generation by spinning the weighted roulette wheel

TABLE 1.2 **A Genetic Algorithm by Hand**

String No.	Initial Population (Randomly Generated)	x Value (Unsigned Integer)	$f(x)$ x^2	pselect$_i$ $\frac{f_i}{\Sigma f}$	Expected count $\frac{f_i}{\bar{f}}$	Actual Count from Roulette Wheel
1	0 1 1 0 1	13	169	0.14	0.58	1
2	1 1 0 0 0	24	576	0.49	1.97	2
3	0 1 0 0 0	8	64	0.06	0.22	0
4	1 0 0 1 1	19	361	0.31	1.23	1
Sum			1170	1.00	4.00	4.0
Average			293	0.25	1.00	1.0
Max			576	0.49	1.97	2.0

(shown in Fig. 1.7) four times. Actual simulation of this process using coin tosses has resulted in string 1 and string 4 receiving one copy in the mating pool, string 2 receiving two copies, and string 3 receiving no copies, as shown in the center of Table 1.2. Comparing this with the expected number of copies (n·pselect$_i$) we have obtained what we should expect: the best get more copies, the average stay even, and the worst die off.

With an active pool of strings looking for mates, simple crossover proceeds in two steps: (1) strings are mated randomly, using coin tosses to pair off the happy couples, and (2) mated string couples cross over, using coin tosses to select the crossing sites. Referring again to Table 1.2, random choice of mates has selected the second string in the mating pool to be mated with the first. With a crossing site of 4, the two strings 01101 and 11000 cross and yield two new strings 01100 and 11001. The remaining two strings in the mating pool are crossed at site 2; the resulting strings may be checked in the table.

The last operator, mutation, is performed on a bit-by-bit basis. We assume that the probability of mutation in this test is 0.001. With 20 transferred bit positions we should expect 20·0.001 = 0.02 bits to undergo mutation during a given generation. Simulation of this process indicates that no bits undergo mutation for this probability value. As a result, no bit positions are changed from 0 to 1 or vice versa during this generation.

Following reproduction, crossover, and mutation, the new population is ready to be tested. To do this, we simply decode the new strings created by the simple genetic algorithm and calculate the fitness function values from the x values thus decoded. The results of a single generation of the simulation are shown at the right of Table 1.2. While drawing concrete conclusions from a single trial of a stochastic process is, at best, a risky business, we start to see how genetic algorithms combine high-performance notions to achieve better performance. In the table, note how both the maximal and average performance have improved in the new population. The population average fitness has improved from 293 to

TABLE 1.2 *(Continued)*

Mating Pool after Reproduction (Cross Site Shown)	Mate (Randomly Selected)	Crossover Site (Randomly Selected)	New Population	x Value	$f(x)$ x^2
0 1 1 0 | 1	2	4	0 1 1 0 0	12	144
1 1 0 0 | 0	1	4	1 1 0 0 1	25	625
1 1 | 0 0 0	4	2	1 1 0 1 1	27	729
1 0 | 0 1 1	3	2	1 0 0 0 0	16	256
					1754
					439
					729

NOTES:

1) Initial population chosen by four repetitions of five coin tosses where heads = 1, tails = 0.
2) Reproduction performed through 1 part in 8 simulation of roulette wheel selection (three coin tosses).
3) Crossover performed through binary decoding of 2 coin tosses (TT = 00_2 = 0 = cross site 1, HH = 11_2 = 3 = cross site 4).
4) Crossover probability assumed to be unity p_c = 1.0.
5) Mutation probability assumed to be 0.001, p_m = 0.001, Expected mutations = 5·4·0.001 = 0.02. No mutations expected during a single generation. None simulated.

439 in one generation. The maximum fitness has increased from 576 to 729 during that same period. Although random processes help cause these happy circumstances, we start to see that this improvement is no fluke. The best string of the first generation (11000) receives two copies because of its high, above-average performance. When this combines at random with the next highest string (10011) and is crossed at location 2 (again at random), one of the resulting strings (11011) proves to be a very good choice indeed.

This event is an excellent illustration of the ideas and notions analogy developed in the previous section. In this case, the resulting good idea is the combination of two above-average notions, namely the substrings 11– – – and – – –11. Although the argument is still somewhat heuristic, we start to see how genetic algorithms effect a robust search. In the next section, we expand our understanding of these concepts by analyzing genetic algorithms in terms of schemata or similarity templates.

The intuitive viewpoint developed thus far has much appeal. We have compared the genetic algorithm with certain human search processes commonly called innovative or creative. Furthermore, hand simulation of the simple genetic algorithm has given us some confidence that indeed something interesting is going on here. Yet, something is missing. What is being processed by genetic algorithms and how do we know whether processing it (whatever it is) will lead to optimal or near optimal results in a particular problem? Clearly, as scientists, engineers, and business managers we need to understand the what and the how of genetic algorithm performance.

To obtain this understanding, we examine the raw data available for any search procedure and discover that we can search more effectively if we exploit important similarities in the coding we use. This leads us to develop the important notion of a ***similarity template,*** or ***schema.*** This in turn leads us to a keystone of the genetic algorithm approach, the ***building block hypothesis.***

GRIST FOR THE SEARCH MILL—IMPORTANT SIMILARITIES

For much too long we have ignored a fundamental question. In a search process given only payoff data (fitness values), what information is contained in a population of strings and their objective function values to help guide a directed search for improvement? To ask this question more clearly, consider the strings and fitness values originally displayed in Table 1.1 from the simulation of the previous section (the black box problem) and gathered below for convenience:

String	Fitness
01101	169
11000	576
01000	64
10011	361

What information is contained in this population to guide a directed search for improvement? On the face of it, there is not very much: four independent samples of different strings with their fitness values. As we stare at the page, however, quite naturally we start scanning up and down the string column, and we notice certain similarities among the strings. Exploring these similarities in more depth, we notice that certain string patterns seem highly associated with good performance. The longer we stare at the strings and their fitness values, the greater is the temptation to experiment with these high fitness associations. It seems perfectly reasonable to play mix and match with some of the substrings that are highly correlated with past success. For example, in the sample population, the strings starting with a 1 seem to be among the best. Might this be an important ingredient in optimizing this function? Certainly with our function ($f(x) = x^2$) and our coding (a five-bit unsigned integer) we know it is (why is this true?). But, what are we doing here? Really, two separate things. First, we are seeking similarities among strings in the population. Second, we are looking for causal relationships between these similarities and high fitness. In so doing, we admit a wealth of new information to help guide a search. To see how much and precisely what information we admit, let us consider the important concept of a schema (plural, *schemata*), or similarity template.

SIMILARITY TEMPLATES (SCHEMATA)

In some sense we are no longer interested in strings as strings alone. Since important similarities among highly fit strings can help guide a search, we question how one string can be similar to its fellow strings. Specifically we ask, in what ways is a string a representative of other string classes with similarities at certain string positions? The framework of schemata provides the tool to answer these questions.

A schema (Holland, 1968, 1975) is a similarity template describing a subset of strings with similarities at certain string positions. For this discussion, let us once again limit ourselves without loss of generality to the binary alphabet {0,1}. We motivate a schema most easily by appending a special symbol to this alphabet; we add the * or *don't care* symbol. With this extended alphabet we can now create strings (schemata) over the ternary alphabet {0, 1, *}, and the meaning of the schema is clear if we think of it as a pattern matching device: a schema matches a particular string if at every location in the schema a 1 matches a 1 in the string, a 0 matches a 0, or a * matches either. As an example, consider the strings and schemata of length 5. The schema *0000 matches two strings, namely {10000, 00000}. As another example, the schema *111* describes a subset with four members {01110, 01111, 11110, 11111}. As one last example, the schema 0*1** matches any of the eight strings of length 5 that begin with a 0 and have a 1 in the third position. As you can start to see, the idea of a schema gives us a powerful and compact way to talk about all the well-defined similarities among finite-length strings over a finite alphabet. We should emphasize that the * is only a metasymbol (a symbol about other symbols); it is never explicitly processed by the genetic algorithm. It is simply a notational device that allows description of all possible similarities among strings of a particular length and alphabet.

Counting the total number of possible schemata is an enlightening exercise. In the previous example, with $l = 5$, we note there are $3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 = 3^5 = 243$ different similarity templates because each of the five positions may be a 0, 1, or *. In general, for alphabets of cardinality (number of alphabet characters) k, there are $(k + 1)^l$ schemata. At first blush, it appears that schemata are making the search more difficult. For an alphabet with k elements there are only (only?) k^l different strings of length l. Why consider the $(k + 1)^l$ schemata and enlarge the space of concern? Put another way, the length 5 example now has only $2^5 = 32$ different alternative strings. Why make matters more difficult by considering $3^5 = 243$ schemata? In fact, the reasoning discussed in the previous section makes things easier. Do you recall glancing up and down the list of four strings and fitness values and trying to figure out what to do next? We recognized that if we considered the strings separately, then we only had four pieces of information;

however, when we considered the strings, their fitness values, and the similarities among the strings in the population, we admitted a wealth of new information to help direct our search. How much information do we admit by considering the similarities? The answer to this question is related to the number of unique schemata contained in the population. To count this quantity exactly requires knowledge of the strings in a particular population. To get a bound on the number of schemata in a particular population, we first count the number of schemata contained in an individual string, and then we get an upper bound on the total number of schemata in the population.

To see this, consider a single string of length 5: 11111, for example. This string is a member of 2^5 schemata because each position may take on its actual value or a don't care symbol. In general, a particular string contains 2^l schemata. As a result, a population of size n contains somewhere between 2^l and $n \cdot 2^l$ schemata, depending upon the population diversity. This fact verifies our earlier intuition. The original motivation for considering important similarities was to get more information to help guide our search. The counting argument shows that a wealth of information about important similarities is indeed contained in even moderately sized populations. We will examine how genetic algorithms effectively exploit this information. At this juncture, some parallel processing appears to be needed if we are to make use of all this information in a timely fashion.

These counting arguments are well and good, but where does this all lead? More pointedly, of the 2^l to $n \cdot 2^l$ schemata contained in a population, how many are actually processed in a useful manner by the genetic algorithm? To obtain the answer to this question, we consider the effect of reproduction, crossover, and mutation on the growth or decay of important schemata from generation to generation. The effect of reproduction on a particular schema is easy to determine; since more highly fit strings have higher probabilities of selection, on average we give an ever increasing number of samples to the observed best similarity patterns (this is a good thing to do, as is shown in the next chapter); however, reproduction alone samples no new points in the space. What then happens to a particular schema when crossover is introduced? Crossover leaves a schema unscathed if it does not cut the schema, but it may disrupt a schema when it does. For example, consider the two schemata 1***0 and **11*. The first is likely to be disrupted by crossover, whereas the second is relatively unlikely to be destroyed. As a result, schemata of short defining length are left alone by crossover and reproduced at a good sampling rate by reproduction operator. Mutation at normal, low rates does not disrupt a particular schema very frequently and we are left with a startling conclusion. Highly fit, short-defining-length schemata (we call them ***building blocks***) are propagated generation to generation by giving exponentially increasing samples to the observed best; all this goes in parallel with no special bookkeeping or special memory other than our population of n strings. In the next chapter we will count how many schemata are processed usefully in each generation. It turns out that the number is something like n^3. This compares favorably with the number of function evaluations (n). Because this processing leverage is so important (and apparently unique to genetic algorithms), we give it a special name, ***implicit parallelism.***

LEARNING THE LINGO

The power behind the simple operations of our genetic algorithm is at least intuitively clearer if we think of building blocks. Some questions remain: How do we know that building blocks lead to improvement? Why is it a near optimal strategy to give exponentially increasing samples to the best? How can we calculate the number of schemata usefully processed by the genetic algorithm? These questions are answered fully in the next chapter, but first we need to master the terminology used by researchers who work with genetic algorithms. Because genetic algorithms are rooted in both natural genetics and computer science, the terminology used in the GA literature is an unholy mix of the natural and the artificial. Until now we have focused on the artificial side of the genetic algorithm's ancestry and talked about strings, alphabets, string positions, and the like. We review the correspondence between these terms and their natural counterparts to connect with the growing GA literature and also to permit our own occasional slip of a natural utterance or two.

Roughly speaking, the *strings* of artificial genetic systems are analogous to *chromosomes* in biological systems. In natural systems, one or more chromosomes combine to form the total genetic prescription for the construction and operation of some organism. In natural systems the total genetic package is called the *genotype*. In artificial genetic systems the total package of strings is called a *structure* (in the early chapters of this book, the structure will consist of a single string, so the text refers to strings and structures interchangeably until it is necessary to differentiate between them). In natural systems, the organism formed by the interaction of the total genetic package with its environment is called the *phenotype*. In artificial genetic systems, the structures decode to form a particular *parameter set, solution alternative,* or *point* (in the solution space). The designer of an artificial genetic system has a variety of alternatives for coding both numeric and nonnumeric parameters. We will confront codings and coding principles in later chapters; for now, we stick to our consideration of GA and natural terminology.

In natural terminology, we say that chromosomes are composed of *genes,* which may take on some number of values called *alleles*. In genetics, the position of a gene (its *locus*) is identified separately from the gene's function. Thus, we can talk of a particular gene, for example an animal's eye color gene, its locus, position 10, and its allele value, blue eyes. In artificial genetic search we say that strings are composed of *features* or *detectors,* which take on different *values*. Features may be located at different *positions* on the string. The correspondence between natural and artificial terminology is summarized in Table 1.3.

Thus far, we have not distinguished between a gene (a particular character) and its locus (its position); the position of a bit in a string has determined its meaning (how it decodes) uniformly throughout a population and throughout time. For example, the string 10000 is decoded as a binary unsigned integer 16 (base 10) because implicitly the 1 is in the 16's place. It is not necessary to limit codings like this, however. A later chapter presents more advanced structures that treat locus and gene separately.