GENETIC ALGORITHMS

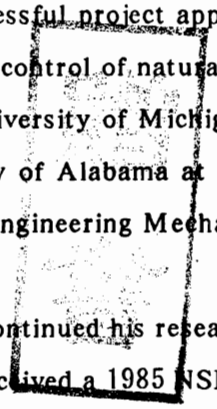
in Search,

Optimization &

Machine Learning

DAVID E. GOLDBERG

As a graduate student at the University of Michigan, **David E. Goldberg** spearheaded a successful project applying genetic algorithms and classifier systems to the control of natural gas pipelines. After receiving his Ph. D. at the University of Michigan, Dr. Goldberg joined the faculty of the University of Alabama at Tuscaloosa where he is now Associate Professor of Engineering Mechanics.



Dr. Goldberg has continued his research in genetic algorithms and classifier systems and received a 1985 NSF Presidential Young Investigator Award for his work. Dr. Goldberg has 12 years of consulting experience in industry and government and has published numerous articles and papers.

Genetic Algorithms in Search, Optimization, and Machine Learning

David E. Goldberg
The University of Alabama

ADDISON-WESLEY PUBLISHING COMPANY, INC.
Reading, Massachusetts • Menlo Park, California • Sydney
Don Mills, Ontario • Madrid • San Juan • New York • Singapore
Amsterdam • Wokingham, England • Tokyo • Bonn

Library of Congress Cataloging-in-Publication Data

Goldberg, David E. (David Edward), 1953–
Genetic algorithms in search, optimization, and machine learning.

Bibliography: p.
Includes index.
1. Combinatorial optimization. 2. Algorithms.
3. Machine learning. I. Title.
QA402.5.G635 1989 006.3'1 88-6276
ISBN 0-201-15767-5

Reprinted with corrections January, 1989

8 9 10 MA 959493

Foreword

I first encountered David Goldberg as a young, PhD-bound Civil Engineer inquiring about my course *Introduction to Adaptive Systems.* He was something of an anomaly becuase his severely practical field experience in the gas-pipeline industry, and his evident interest in that industry, did not seem to dampen his interest in what was, after all, an abstract course involving a lot of "biological stuff." After he enrolled in the course, I soon realized that his expressed interests in control, gas pipelines, and AI were the surface tell-tales of a wide-ranging curiosity and a talent for careful analysis. He was, and is, an engineer interested in building, but he was, and is, equally interested in ideas.

Not long thereafter, Dave asked if I would be willing to co-chair (with Ben Wylie, the chairman of our Civil Engineering Department) a dissertation investigating the use of genetic algorithms and classifier systems in the control of gas-pipeline transmission. My first reaction was that this was too difficult a problem for a dissertation—there are no closed analytic solutions to even simple versions of the problem, and actual operation involves long, craftsmanlike apprenticeships. Dave persisted, and in a surprisingly short time produced a dissertation that, in turn, produced for him a 1985 NSF Presidential Young Investigator Award. So much for my intuition as to what constitutes a reasonable dissertation.

In the past few years GAs have gone from an arcane subject known to a few of my students, and their students, to a subject engaging the curiosity of many different research communities including researchers in economics, political science, psychology, linguistics, immunology, biology, and computer science. A major reason for this interest is that GAs really work. GAs offer robust procedures that can exploit massively parallel architectures and, applied to classifier systems, they provide a new route toward an understanding of intelligence and adaptation. David Goldberg's book provides a turnpike into this territory.

One cannot be around David Goldberg for long without being infected by his enthusiasm and energy. That enthusiasm comes across in this book. It is also

an embodiment of his passion for clear explanations and carefully worked examples. His book does an exceptional job of making the methods of GAs and classifier systems available to a wide audience. Dave is deeply interested in the intellectual problems of GAs and classifier systems, but he is interested even more in seeing these systems used. This book, I think, will be instrumental in realizing that ambition.

John Holland
Ann Arbor, Michigan

Preface

This book is about genetic algorithms (GAs)—search procedures based on the mechanics of natural selection and natural genetics. In writing it, I have tried to bring together the computer techniques, mathematical tools, and research results that will enable you to apply genetic algorithms to problems in your field. If you choose to do so, you will join a growing group of researchers and practitioners who have come to appreciate the natural analogues, mathematical analyses, and computer techniques comprised by the genetic algorithm methodology.

The book is designed to be a textbook and a self-study guide. I have tested the draft text in a one semester, senior-level undergraduate/first-year graduate course devoted to genetic algorithms. Although the students came from different backgrounds (biochemistry, chemical engineering, computer science, electrical engineering, engineering mechanics, English, mathematics, mechanical engineering, and physics) and had wide differences in mathematical and computational maturity, they all acquired an understanding of the basic algorithm and its theory of operation. To reach such a diverse audience, the tone of the book is intentionally casual, and rigor has almost always been sacrificed in the interest of building intuition and understanding. Worked out examples illustrate major topics, and computer assignments are available at the end of each chapter.

I have minimized the mathematics, genetics, and computer background required to read this book. An understanding of introductory college-level mathematics (algebra and a little calculus) is assumed. Elementary notions of counting and finite probability are used, and Appendix A summarizes the important concepts briefly. I assume no particular knowledge of genetics and define all required genetic terminology and concepts within the text. Last, some computer programming ability is necessary. If you have programmed a computer in any language, you should be able to follow the computer examples I present. All computer code in this book is written in Pascal, and Appendix B presents a brief introduction to the essentials of that language.

Although I have not explicitly subdivided the book into separate parts, the chapters may be grouped in two major categories: those dealing with search and optimization and those dealing with machine learning.

The first five chapters are devoted to genetic algorithms in search and optimization. Chapter 1 introduces the topic of genetic search; it also describes a simple genetic algorithm and illustrates the GA's application through a hand calculation. Chapter 2 introduces the essential theoretical basis of GAs, covering topics including schemata, the fundamental theorem, and extended analysis. If you dislike theory, you can safely skip Chapter 2 without excessive loss of continuity; however, before doing so, I suggest you try reading it anyway. The mathematical underpinnings of GAs are not difficult to follow, but their ramifications are subtle; some attention to analysis early in the study of GAs promotes fuller understanding of algorithm power. Chapter 3 introduces computer implementation of genetic algorithms through example. Specifically, a Pascal code called the simple genetic algorithm (SGA) is presented along with a number of extensions. Chapter 4 presents a historical account of early genetic algorithms together with a potpourri of current applications. Chapter 5 examines more advanced genetic operators and presents a number of applications illustrating their use. These include applications of micro- and macro-level operators as well as hybrid techniques.

Chapters 6 and 7 present the application of genetic algorithms in machine learning systems. Chapter 6 gives a generic description of one type of genetics-based machine learning (GBML) system, a classifier system. The theory of operation of such a system is briefly reviewed, and one Pascal implementation called the simple classifier system (SCS) is presented and applied to the learning of a boolean function. Chapter 7 rounds out the picture of GBML by presenting a historical review of early GBML systems together with a selective survey of other current systems and topics.

ACKNOWLEDGMENTS

In writing acknowledgments for a book on genetic algorithms, there is no question who should get top billing. I thank John H. Holland from the University of Michigan for his encouragement of this project and for giving birth to the infant we now recognize as the genetic algorithms movement. It hasn't been easy nurturing such a child. At times she showed signs of stunted intellectual growth, and the other kids on the block haven't always treated her very nicely. Nonetheless, John stood by his baby with the quiet confidence only a father can possess, knowing that his daughter would one day take her rightful place in the community of ideas.

I also thank two men who have influenced me in more ways than they know: E. Benjamin Wylie and William D. Jordan. Ben Wylie was my dissertation adviser

in Civil Engineering at the University of Michigan. When I approached him with the idea for a dissertation about gas pipelines and genetic algorithms, he was appropriately skeptical, but he gave me the rope and taught me the research and organizational skills necessary not to hang myself. Bill Jordan was my Department Head in Engineering Mechanics at The University of Alabama (he retired in 1986). He was and continues to be a model of teaching quality and administrative fairness that I still strive to emulate.

I thank my colleagues in the Department of Engineering Mechanics at Alabama, A. E. Carden, C. H. Chang, C. R. Evces, S. C. Gambrell, J. L. Hill, S. E. Jones, D. C. Raney, and H. B. Wilson, for their encouragement and support. I also thank my many colleagues in the genetic algorithms community. Particular thanks are due Stewart Wilson at the Rowland Institute for Science for providing special encouragement and a sympathetic ear on numerous occasions.

I thank my students (the notorious Bama Gene Hackers), including C. L. Bridges, K. Deb, C. L. Karr, C. H. Kuo, R. Lingle, Jr., M. P. Samtani, P. Segrest, T. Sivapalan, R. E. Smith, and M. Valenzuela-Rendon, for lots of long hours and hard work. I also recognize the workmanlike assistance rendered by a string of right-hand persons: A. L. Thomas, S. Damsky, B. Korb, and K. Y. Lee.

I acknowledge the editorial assistance provided by Sarah Bane Wood at Alabama. I am also grateful to the team at Addison-Wesley, including Peter Gordon, Helen Goldstein, Helen Wythe, and Cynthia Benn, for providing expert advice and assistance during this project.

I thank the reviewers, Ken De Jong, John Holland, and Stewart Wilson, for their comments and suggestions.

A number of individuals and organizations have granted permission to reprint or adapt materials originally printed elsewhere. I gratefully acknowledge the permission granted by the following individuals: L. B. Booker, G. E. P. Box, K. A. De Jong, S. Forrest, J. J. Grefenstette, J. H. Holland, J. D. Schaffer, S. F. Smith, and S. W. Wilson. I also acknowledge the permission granted by the following organizations: Academic Press, Academic Press London Ltd. (*Journal of Theoretical Biology*), the American Society of Civil Engineers, the Association for Computing Machinery, the Conference Committee of the International Conference on Genetic Algorithms, Kluwer Academic Publishers (*Machine Learning*), North-Holland Physics Publishing, the Royal Statistical Society (*Journal of the Royal Statistical Society, C*), and John Wiley and Sons, Inc.

I thank my spouse and still best friend, Mary Ann, for her patience and assistance. There were more than a few evenings and weekends I didn't come home when I said I would, and she proofread the manuscript, judiciously separating my tolerable quips from my unacceptable quirks. Untold numbers of readers would thank you, Nary (sic), if they knew the fate they have been spared by your sound judgment.

This material is based upon work supported by the National Science Foundation under Grant MSM-8451610. I am also grateful for research support provided by the Alabama Research Institute, Digital Equipment Corporation, Intel

Corporation, Mr. Peter Prater, the Rowland Institute for Science, Texas Instruments Incorporated, and The University of Alabama.

Last, it has become a cliche in textbooks and monographs: after thanking one and all for their assistance, the author gallantly accepts blame for all remaining errors in the text. This is usually done with no small amount of pomp and circumstance—a ritualistic incantation to ward off the evil spirits of error. I will forgo this exercise and close these acknowledgments by paraphrasing a piece of graffiti that I first spotted on the third floor of the West Engineering Building at the University of Michigan:

> To err is human. To really foul up, use a computer.

Unfortunately, in writing this book, I find myself subject to both of these sources of error, and no doubt many mistakes remain. I can only take comfort in knowing that error is the one inevitable side effect of our human past and the probable destiny of our artificially intelligent future.

Contents

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 differ-

ent 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,

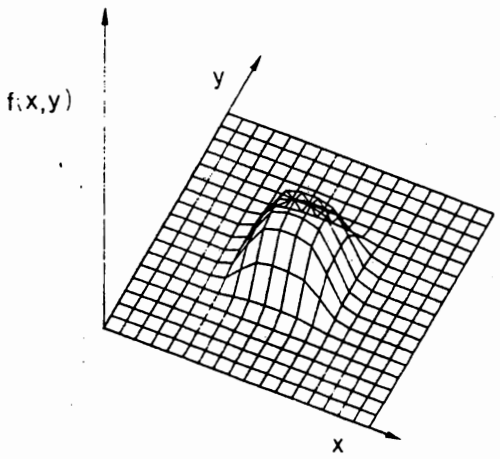


FIGURE 1.1 **The single-peak function is easy for calculus-based methods.**

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

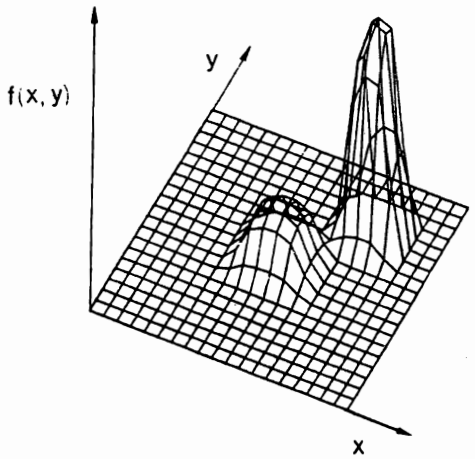


FIGURE 1.2 **The multiple-peak function causes a dilemma. Which hill should we climb?**

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

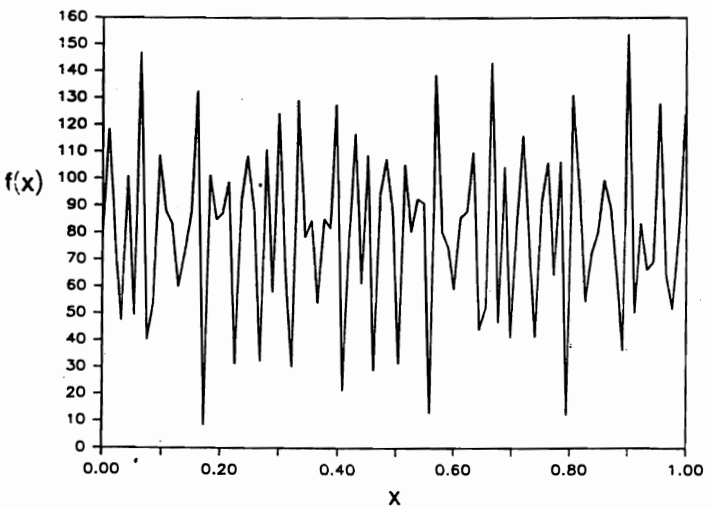


FIGURE 1.3 **Many functions are noisy and discontinuous and thus unsuitable for search by traditional methods.**

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.

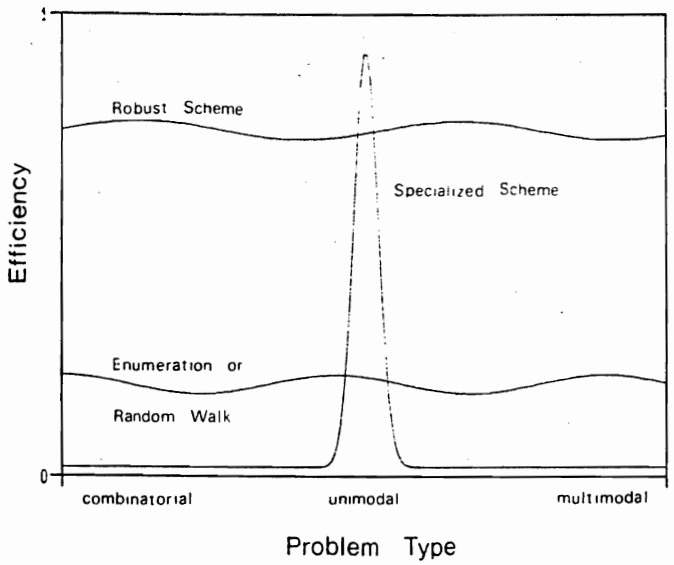


FIGURE 1.4 **Many traditional schemes work well in a narrow problem domain. Enumerative schemes and random walks work equally inefficiently across a broad spectrum. A robust method works well across a broad spectrum of problems.**

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 $f = f(s)$,

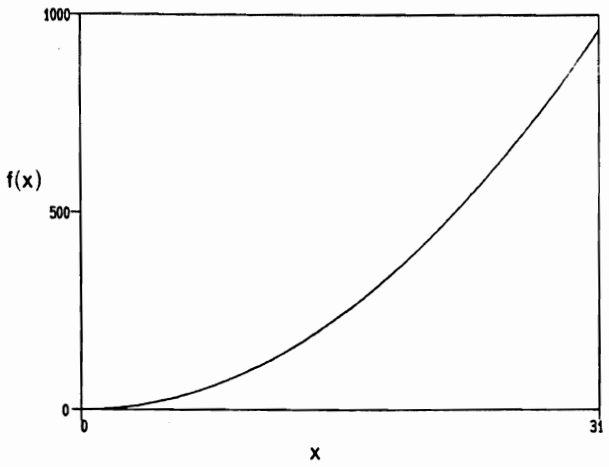


FIGURE 1.5 A simple function optimization example, the function $f(x) = x^2$ on the integer interval [0, 31].

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

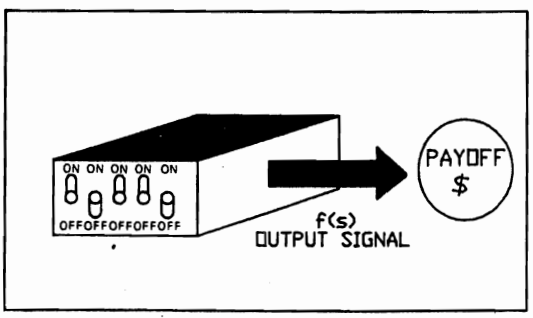


FIGURE 1.6 A black box optimization problem with five on-off switches illustrates the idea of a coding and a payoff measure. Genetic algorithms only require these two things: they don't need to know the workings of the black box.

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-

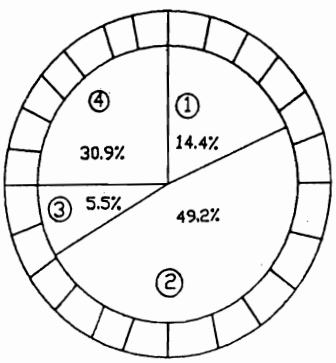


FIGURE 1.7 Simple reproduction allocates offspring strings using a roulette wheel with slots sized according to fitness. The sample wheel is sized for the problem of Tables 1.1 and 1.2.

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:

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

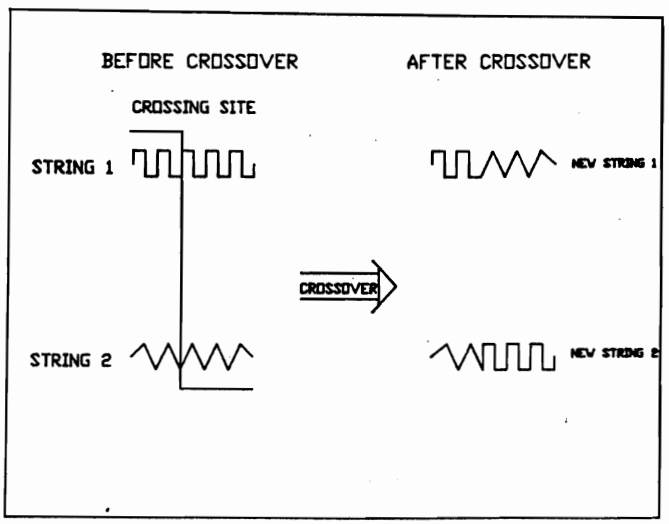


FIGURE 1.8 A schematic of simple crossover shows the alignment of two strings and the partial exchange of information, using a cross site chosen at random.

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.

TABLE 1.3 Comparison of Natural and GA Terminology

Natural	Genetic Algorithm
chromosome	string
gene	feature, character, or detector
allele	feature value
locus	string position
genotype	structure
phenotype	parameter set, alternative solution, a decoded structure
epistasis	nonlinearity

SUMMARY

This chapter has laid the foundation for understanding genetic algorithms, their mechanics and their power. We are led to these methods by our search for robustness; natural systems are robust—efficient and efficacious—as they adapt to a wide variety of environments. By abstracting nature's adaptation algorithm of choice in artificial form we hope to achieve similar breadth of performance. In fact, genetic algorithms have demonstrated their capability in a number of analytical and empirical studies.

The chapter has presented the detailed mechanics of a simple, three-operator genetic algorithm. Genetic algorithms operate on populations of strings, with the string coded to represent some underlying parameter set. Reproduction, crossover, and mutation are applied to successive string populations to create new string populations. These operators are simplicity itself, involving nothing more complex than random number generation, string copying, and partial string exchanging; yet, despite their simplicity, the resulting search performance is wide-ranging and impressive. Genetic algorithms realize an innovative notion exchange among strings and thus connect to our own ideas of human search or discovery. A simulation of one generation of the simple genetic algorithm has helped illustrate both the detail and the power of the method.

Four differences separate genetic algorithms from more conventional optimization techniques:

1. Direct manipulation of a coding
2. Search from a population, not a single point
3. Search via sampling, a blind search
4. Search using stochastic operators, not deterministic rules

Genetic algorithms manipulate decision or control variable representations at the string level to exploit similarities among high-performance strings. Other methods usually deal with functions and their control variables directly. Because

genetic algorithms operate at the coding level, they are difficult to fool even when the function may be difficult for traditional schemes.

Genetic algorithms work from a population; many other methods work from a single point. In this way, GAs find safety in numbers. By maintaining a population of well-adapted sample points, the probability of reaching a false peak is reduced.

Genetic algorithms achieve much of their breadth by ignoring information except that concerning payoff. Other methods rely heavily on such information, and in problems where the necessary information is not available or difficult to obtain, these other techniques break down. GAs remain general by exploiting information available in any search problem. Genetic algorithms process similarities in the underlying coding together with information ranking the structures according to their survival capability in the current environment. By exploiting such widely available information, GAs may be applied in virtually any problem.

The transition rules of genetic algorithms are stochastic; many other methods have deterministic transition rules. A distinction exists, however, between the randomized operators of genetic algorithms and other methods that are simple random walks. Genetic algorithms use random choice to guide a highly exploitative search. This may seem unusual, using chance to achieve directed results (the best points), but nature is full of precedent.

We have started a more rigorous appraisal of genetic algorithm performance through the concept of schemata or similarity templates. A schema is a string over an extended alphabet, {0,1,*} where the 0 and the 1 retain their normal meaning and the * is a wild card or don't care symbol. This notational device greatly simplifies the analysis of the genetic algorithm method because it explicitly recognizes all the possible similarities in a population of strings. We have discussed how building blocks—short, high-performance schemata—are combined to form strings with expected higher performance. This occurs because building blocks are sampled at near optimal rates and recombined via crossover. Mutation has little effect on these building blocks; like an insurance policy, it helps prevent the irrecoverable loss of potentially important genetic material.

The simple genetic algorithm studied in this chapter has much to recommend it. In the next chapter, we will analyze its operation more carefully. Following this, we will implement the simple GA in a short computer program and examine some applications in practical problems.

■ PROBLEMS

1.1. Consider a black box containing eight multiple-position switches. Switches 1 and 2 may be set in any of 16 positions. Switches 3, 4, and 5 are four-position switches, and switches 6–8 have only two positions. Calculate the number of unique switch settings possible for this black box device.

1.2. For the black box device of Problem 1.1, design a natural string coding that uses eight positions, one position for each switch. Count the number of switch

settings represented by your coding and count the number of schemata or similarity templates inherent in your coding.

1.3. For the black box device of Problem 1.1, design a minimal binary coding for the eight switches and compare the number of schemata in this coding to a coding for Problem 1.2.

1.4. Consider a binary string of length 11, and consider a schema, 1*********1. Under crossover with uniform crossover site selection, calculate a lower limit on the probability of this schema surviving crossover. Calculate survival probabilities under the same assumptions for the following schemata: ****10*****, 11*********, ***111*****, ****1*0****, **1***1**0*.

1.5. If the distance between the outermost alleles of a particular schema is called its ***defining length*** δ, derive an approximate expression for the survival probability of a particular schema of total length l and defining length δ under the operation of simple crossover.

1.6. Six strings have the following fitness function values: 5, 10, 15, 25, 50, 100. Under roulette wheel selection, calculate the expected number of copies of each string in the mating pool if a constant population size, $n = 6$, is maintained.

1.7. Instead of using roulette wheel selection during reproduction, suppose we define a copy count for each string, $ncount_i$ as follows: $ncount_i = f_i/\bar{f}$ where f_i is the fitness of the ith string and $\bar{f}$ is the average fitness of the population. The copy count is then used to generate the number of members of the mating pool by giving the integer part of $ncount_i$ copies to the ith string and an additional copy with probability equal to the fractional part of $ncount_i$. For example, with $f_i = 100$ and $\bar{f} = 80$, string i would receive an $ncount_i$ of 1.25, and thus would receive one copy with probability 1.0 and another copy with probability 0.25. Using the string fitness values in Problem 1.6, calculate the expected number of copies for each of the six strings. Calculate the total number of strings expected in the gene pool under this form of reproduction.

1.8. The form of reproduction discussed in Problem 1.7 is sometimes called reproduction with expected number control. In a short essay, explain why this is so. In what ways are roulette wheel selection and expected number control similar? In what ways are they different?

1.9. Suppose the probability of a mutation at a single bit position is 0.1. Calculate the probability of a 10-bit string surviving mutation without change. Calculate the probability of a 20-bit string surviving mutation without change. Recalculate the survival probabilities for both 10- and 20-bit strings when the mutation probability is 0.01.

1.10. Consider the strings and schemata of length 11. For the following schemata, calculate the probability of surviving mutation if the probability of mutation is 0.1 at a single bit position: ***1**0****, 1*********0, ***111*****, *1000010*11. Recalculate the survival probabilities for a mutation probability $p_m = 0.01$.

■ COMPUTER ASSIGNMENTS

A. One of the primitive functions required in doing genetic algorithms on a computer is the ability to generate pseudorandom numbers. The numbers are pseudorandom because as von Neumann once said, "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin." As part of this assignment, go forth and sin some more. Use the random number generator given in Appendix B to create a program where you generate 1000 random numbers between 0 and 1. Keep track of how many numbers are generated in each of the four quartiles, 0–0.25, 0.25–0.5, 0.5–0.75, 0.75–1.0, and compare the actual counts with the expected number. Is the difference within reasonable limits? How can you quantify whether the difference is reasonable?

B. Suppose you have 10 strings with the following probabilities of selection in the next generation: 0.1, 0.2, 0.05, 0.15, 0.0, 0.11, 0.07, 0.04, 0.00, 0.12, 0.16. Given that these are the only possible alternatives, calculate whether the probabilities are consistent. Write a computer program that simulates roulette wheel selection for these 10 strings. Spin the wheel 1000 times and keep track of the number of selections for each string, comparing this number to the expected number of selections.

C. Write a function that generates a pseudorandom integer between some specified lower limit and some specified upper limit. Test the program by generating 1000 numbers between 3 and and 12. Keep track of the quantity of each number selected and compare these figures to the expected quantities.

D. Create a procedure that receives two binary strings and a crossing site value, performs simple crossover, and returns two offspring strings. Test the program by crossing the following strings of length 10: 1011101011, 0000110100. Try crossing site values of -3, 1, 6, and 20.

E. Create a function *mutation* that complements a particular bit value with specified mutation probability p_m. Test the function by performing 1000 calls to mutation using mutation probabilities $p_m = 0.001, 0.01, 0.1$. Compare the realized number of mutations to the expected number.

F. Using the simple crossover operator of Assignment D, repeatedly apply the crossover operator to strings contained within the following population of size $n = 200$ and $l = 5$:

100 copies of 11100
100 copies of 00011

Perform crossover ($p_c = 1.0$) for 50 generations without replacement under no selection. Compare the initial and final distributions of strings. Also compare the expected quantity of each string to the realized quantity in generation 50.

2 Genetic Algorithms Revisited: Mathematical Foundations

The broad brush of Chapter 1 painted an accurate, if somewhat crude, picture of genetic algorithms and their mechanics and power. Perhaps these brush strokes appeal to your own sense of human discovery and search. That somehow a regular though randomized procedure can achieve some of the breadth and intuitive flair of human search seems almost too good to be true. That this discovery procedure should mirror the natural processes that created the species possessing the procedure is a recursion of which Gödel, Escher, or Bach (Hofstadter, 1979) could each have been proud. Despite their intuitive appeal, and despite their symmetry, it is crucial that we back these fuzzy feelings and speculations about genetic algorithms using cold, mathematical facts.

Actually, we have already begun a more rigorous appraisal of GAs. Toward the end of the last chapter, the fundamental concept of a schema or similarity template was introduced. Quantitatively, we found that there are indeed a large number of similarities to exploit in a population of strings. Intuitively, we saw how genetic algorithms exploit in parallel the many similarities contained in building blocks or short, high-performance schemata. In this chapter, we make these observations more rigorous by doing several things. First, we count the schemata represented within a population of strings and consider which grow and which decay during any given generation. To do this, we consider the effect of reproduction, crossover, and mutation on a particular schema. This analysis leads to the fundamental theorem of genetic algorithms that quantifies these growth and decay rates more precisely; it also points to the mathematical form

of this growth. This form is connected to an important and classical problem of decision theory, the two-armed bandit problem (and its extension, the *k*-armed bandit). The mathematical similarity between the optimal (minimal loss) solution to the two-armed and *k*-armed bandit and the equation describing the number of trials given to successive generations of schemata in the simple genetic algorithm is striking. Counting the number of schemata that are usefully processed by the simple genetic algorithm reveals tremendous leverage in the building block processing. Finally, we consider an important question: How do we know that combining building blocks leads to high performance in arbitrary problems? The question sparks our consideration of some relatively new tools of genetic algorithm analysis: schema transforms and the minimal deceptive problem.

WHO SHALL LIVE AND WHO SHALL DIE? THE FUNDAMENTAL THEOREM

The operation of genetic algorithms is remarkably straightforward. After all, we start with a random population of n strings, copy strings with some bias toward the best, mate and partially swap substrings, and mutate an occasional bit value for good measure. Even though genetic algorithms directly manipulate a population of strings in this straightforward manner, in Chapter 1 we started to recognize that this explicit processing of strings really causes the implicit processing of many schemata during each generation. To analyze the growth and decay of the many schemata contained in a population, we need some simple notation to add rigor to the discussion. We consider the operation of reproduction, crossover, and mutation on the schemata contained in the population.

We consider strings, without loss of generality, to be constructed over the binary alphabet $V = \{0, 1\}$. As a notational convenience, we refer to strings by capital letters and individual characters by lowercase letters subscripted by their position. For example, the seven-bit string $A = 0111000$ may be represented symbolically as follows:

$$A = a_1a_2a_3a_4a_5a_6a_7.$$

Here each of the a_i represents a single binary feature or detector (in accordance with natural analogy, we sometimes call the a_i's *genes*), where each feature may take on a value 1 or 0 (we sometimes call the a_i values *alleles*). In the particular string 0111000, a_1 is 0, a_2 is 1, a_3 is 1, etc. It is also possible to have strings where detectors are not ordered sequentially as in string A. For example a string A' could have the following ordering:

$$A' = a_2a_6a_4a_3a_7a_1a_5.$$

A later chapter explores the effect of extending the representation to allow features to be located in a manner independent of their function. For now, assume that a feature's function may be determined by its position.

Meaningful genetic search requires a population of strings, and we consider

a population of individual strings A_j, $j = 1, 2, \ldots, n$, contained in the population $\mathbf{A}(t)$ at time (or generation) t where the boldface is used to denote a population.

Besides notation to describe populations, strings, bit positions, and alleles, we need convenient notation to describe the schemata contained in individual strings and populations. Let us consider a schema H taken from the three-letter alphabet $V+ = \{0, 1, *\}$. As discussed in the previous chapter, the additional symbol, the asterisk or star *, is a don't care or wild card symbol which matches either a 0 or a 1 at a particular position. For example, consider the length 7 schema $H = *11*0**$. Note that the string $A = 0111000$ discussed above is an example of the schema H, because the string alleles a_i match schema positions b_i at the fixed positions 2, 3, and 5.

From the results of the last chapter, recall that there are 3^l schemata or similarity defined over a binary string of length l. In general, for alphabets of cardinality k, there are $(k + 1)^l$ schemata. Furthermore, recall that in a string population with n members there are at most $n \cdot 2^l$ schemata contained in a population because each string is itself a representative of 2^l schemata. These counting arguments give us some feel for the magnitude of information being processed by genetic algorithms; however, to really understand the important building blocks of future solutions, we need to distinguish between different types of schemata.

All schemata are not created equal. Some are more specific than others. For example, the schema 011*1** is a more definite statement about important similarity than the schema 0******. Furthermore, certain schema span more of the total string length than others. For example, the schema 1****1* spans a larger portion of the string than the schema 1*1****. To quantify these ideas, we introduce two schema properties: schema *order* and *defining length*.

The order of a schema H, denoted by $o(H)$, is simply the number of fixed positions (in a binary alphabet, the number of 1's and 0's) present in the template. In the examples above, the order of the schema 011*1** is 4 (symbolically, $o(011*1**) = 4$), whereas the order of the schema 0****** is 1.

The defining length of a schema H, denoted by $\delta(H)$, is the distance between the first and last specific string position. For example, the schema 011*1** has defining length $\delta = 4$ because the last specific position is 5 and the first specific position is 1, and the distance between them is $\delta(H) = 5 - 1 = 4$. In the other example (the schema 0******), the defining length is particularly easy to calculate. Since there is only a single fixed position, the first and last specific positions are the same, and the defining length $\delta = 0$.

Schemata and their properties are interesting notational devices for rigorously discussing and classifying string similarities. More than this, they provide the basic means for analyzing the net effect of reproduction and genetic operators on building blocks contained within the population. Let us consider the individual and combined effect of reproduction, crossover, and mutation on schemata contained within a population of strings.

The effect of reproduction on the expected number of schemata in the population is particularly easy to determine. Suppose at a given time step t there are

m examples of a particular schema H contained within the population $A(t)$ where we write $m = m(H,t)$ (there are possibly different quantities of different schemata H at different times t). During reproduction, a string is copied according to its fitness, or more precisely a string A_i gets selected with probability $p_i = f_i/\Sigma f_j$. After picking a nonoverlapping population of size n with replacement from the population $A(t)$, we expect to have $m(H, t+1)$ representatives of the schema H in the population at time $t-1$ as given by the equation $m(H, t+1) = m(H,t)\cdot n\cdot f(H)/\Sigma f_j$, where $f(H)$ is the average fitness of the strings representing schema H at time t. If we recognize that the average fitness of the entire population may be written as $\bar{f} = \Sigma f_j/n$ then we may rewrite the reproductive schema growth equation as follows:

$$m(H, t+1) = m(H, t)\frac{f(H)}{\bar{f}}.$$

In words, a particular schema grows as the ratio of the average fitness of the schema to the average fitness of the population. Put another way, schemata with fitness values above the population average will receive an increasing number of samples in the next generation, while schemata with fitness values below the population average will receive a decreasing number of samples. It is interesting to observe that this expected behavior is carried out with every schema H contained in a particular population A in parallel. In other words, all the schemata in a population grow or decay according to their schema averages under the operation of reproduction alone. In a moment, we examine why this might be a good thing to do. For the time being, simply note that many things go on in parallel with simple operations on the n strings in the population.

The effect of reproduction on the number of schemata is qualitatively clear; above-average schemata grow and below-average schemata die off. Can we learn anything else about the mathematical form of this growth (decay) from the schema difference equation? Suppose we assume that a particular schema H remains above average an amount $c\bar{f}$ with c a constant. Under this assumption we can rewrite the schema difference equation as follows:

$$m(H, t+1) = m(H,t)\frac{(\bar{f} + c\bar{f})}{\bar{f}} = (1 + c)\cdot m(H, t).$$

Starting at $t = 0$ and assuming a stationary value of c, we obtain the equation

$$m(H, t) = m(H, 0)\cdot(1 + c)^t.$$

Business-oriented readers will recognize this equation as the compound interest equation, and mathematically oriented readers will recognize a geometric progression or the discrete analog of an exponential form. The effect of reproduction is now quantitatively clear; reproduction allocates exponentially increasing (decreasing) numbers of trials to above- (below-) average schemata. We will connect this rate of schemata allocation to the multiarmed bandit problem, but for right now we will investigate how crossover and mutation affect this allocation of trials.

To some extent it is curious that reproduction can allocate exponentially increasing and decreasing numbers of schemata to future generations in parallel; many, many different schemata are sampled in parallel according to the same rule through the use of n simple reproduction operations. On the other hand, reproduction alone does nothing to promote exploration of new regions of the search space, since no new points are searched: if we only copy old structures without change, then how will we ever try anything new? This is where crossover steps in. Crossover is a structured yet randomized information exchange between strings. Crossover creates new structures with a minimum of disruption to the allocation strategy dictated by reproduction alone. This results in exponentially increasing (or decreasing) proportions of schemata in a population on many of the schemata contained in the population.

To see which schemata are affected by crossover and which are not, consider a particular string of length $l = 7$ and two representative schemata within that string:

$$A = 0\ 1\ 1\ 1\ 0\ 0\ 0$$
$$H_1 = *\ 1\ *\ *\ *\ *\ 0$$
$$H_2 = *\ *\ *\ 1\ 0\ *\ *$$

Clearly the two schemata H_1 and H_2 are represented in the string A, but to see the effect of crossover on the schemata, we first recall that simple crossover proceeds with the random selection of a mate, the random selection of a crossover site, and the exchange of substrings from the beginning of the string to the crossover site inclusively with the corresponding substring of the chosen mate. Suppose string A has been chosen for mating and crossover. In this string of length 7, suppose we roll a single die to choose the crossing site (there are six sites in a string of length 7). Further suppose that the die turns up a 3, meaning that the cross cut will take place between positions 3 and 4. The effect of this cross on our two schemata H_1 and H_2 can be seen easily in the following example, where the crossing site has been marked with the separator symbol | :

$$A = 0\ 1\ 1\ |\ 1\ 0\ 0\ 0$$
$$H_1 = *\ 1\ *\ |\ *\ *\ *\ 0$$
$$H_2 = *\ *\ *\ |\ 1\ 0\ *\ *$$

Unless string A's mate is identical to A at the fixed positions of the schema (a possibility that we conservatively ignore), the schema H_1 will be destroyed because the 1 at position 2 and the 0 at position 7 will be placed in different offspring (they are on opposite sides of the separator symbol marking the cross point, or cut point). It is equally clear that with the same cut point (between bits 3 and 4), schema H_2 will survive because the 1 at position 4 and the 0 at position 5 will be carried intact to a single offspring. Although we have used a specific cut point for illustration, it is clear that schema H_1 is less likely to survive crossover than schema H_2 because on average the cut point is more likely to fall between the extreme fixed positions. To quantify this observation, we note that schema H_1 has a defining length of 5. If the crossover site is selected uniformly at random

among the $l - 1 = 7 - 1 = 6$ possible sites, then clearly schema H_1 is destroyed with probability $p_d = \delta(H_1)/(l - 1) = 5/6$ (it survives with probability $p_s = 1 - p_d = 1/6$). Similarly, the schema H_2 has defining length $\delta(H_2) = 1$, and it is destroyed during that one event in six where the cut site is selected to occur between positions 4 and 5 such that $p_d = 1/6$ or the survival probability is $p_s = 1 - p_d = 5/6$.

More generally, we see that a lower bound on crossover survival probability p_s can be calculated for any schema. Because a schema survives when the cross site falls outside the defining length, the survival probability under simple crossover is $p_s = 1 - \delta(H)/(l - 1)$, since the schema is likely to be disrupted whenever a site within the defining length is selected from the $l - 1$ possible sites. If crossover is itself performed by random choice, say with probability p_c at a particular mating, the survival probability may be given by the expression

$$p_s \geq 1 - p_c \cdot \frac{\delta(H)}{l - 1},$$

which reduces to the earlier expression when $p_c = 1.0$.

The combined effect of reproduction and crossover may now be considered. As when we considered reproduction alone, we are interested in calculating the number of a particular schema H expected in the next generation. Assuming independence of the reproduction and crossover operations, we obtain the estimate:

$$m(H, t + 1) \geq m(H, t) \cdot \frac{f(H)}{\bar{f}} \left[1 - p_c \cdot \frac{\delta(H)}{l - 1} \right].$$

Comparing this to the previous expression for reproduction alone, the combined effect of crossover and reproduction is obtained by multiplying the expected number of schemata for reproduction alone by the survival probability under crossover p_s. Once again the effect of the operations is clear. Schema H grows or decays depending upon a multiplication factor. With both crossover and reproduction, that factor depends on two things: whether the schema is above or below the population average and whether the schema has relatively short or long defining length. Clearly, those schemata with both above-average observed performance and short defining lengths are going to be sampled at exponentially increasing rates.

The last operator to consider is mutation. Using our previous definition, mutation is the random alteration of a single position with probability p_m. In order for a schema H to survive, all of the specified positions must themselves survive. Therefore, since a single allele survives with probability $(1 - p_m)$, and since each of the mutations is statistically independent, a particular schema survives when each of the $o(H)$ fixed positions within the schema survives. Multiplying the survival probability $(1 - p_m)$ by itself $o(H)$ times, we obtain the probability of surviving mutation, $(1 - p_m)^{o(H)}$. For small values of p_m ($p_m << 1$), the schema survival probability may be approximated by the expression $1 - o(H) \cdot p_m$. We

therefore conclude that a particular schema H receives an expected number of copies in the next generation under reproduction, crossover, and mutation as given by the following equation (ignoring small cross-product terms):

$$m(H, t+1) \geq m(H, t) \cdot \frac{f(H)}{\bar{f}}\left[1 - p_c\frac{\delta(H)}{l-1} - o(H)p_m\right].$$

The addition of mutation changes our previous conclusions little. Short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations. This conclusion is important, so important that we give it a special name: the *Schema Theorem,* or the Fundamental Theorem of Genetic Algorithms. Although the calculations that led us to prove the schema theorem were not too demanding, the theorem's implications are far reaching and subtle. To see this, we examine the effect of the three-operator genetic algorithm on schemata in a population through another visit to the hand-calculated GA of Chapter 1.

SCHEMA PROCESSING AT WORK: AN EXAMPLE BY HAND REVISITED

Chapter 1 demonstrated the mechanics of the simple GA through a hand calculation of a single generation. Let us return to that example, this time observing how the GA processes schemata—not individual strings—within the population. The hand calculation of Chapter 1 is reproduced in Table 2.1. In addition to the information presented earlier, we also keep a running count of three particular schemata, which we call H_1, H_2, and H_3, where H_1 = 1****, H_2 = *10**, and H_3 = 1***0.

Observe the effect of reproduction, crossover, and mutation on the first schema, H_1. During the reproduction phase, the strings are copied probabilistically according to their fitness values. Looking at the first column of the table, we notice that strings 2 and 4 are both representatives of the schema 1****. After reproduction, we note that three copies of the schema have been produced (strings 2, 3, 4 in the mating pool column). Does this number correspond with the value predicted by the schema theorem? From the schema theorem we expect to have $m \cdot f(H)/\bar{f}$ copies. Calculating the schema average $f(H_1)$, we obtain (576 + 361)/2 = 468.5. Dividing this by the population average $\bar{f}$ = 293 and multiplying by the number of H_1 schemata at time t, $m(H_1, t) = 2$, we obtain the expected number of H_1 schemata at time $t + 1$, $m(H, t+1) = 2 \cdot 468.5/293 = 3.20$. Comparing this to the actual number of schemata (three), we see that we have the correct number of copies. Taking this one step further, we realize that crossover cannot have any further effect on this schema because a defining length $\delta(H_1) = 0$ prevents disruption of the single bit. Furthermore, with the mutation rate set at $p_m = 0.001$ we expect to have $m \cdot p_m = 3 \cdot 0.001 = 0.003$ or no bits changed within the three schema copies in the three strings. As a result, we ob-

TABLE 2.1 GA Processing of Schemata—Hand Calculations

String Processing

String No.	Initial Population (Randomly Generated)	x Value (Unsigned Integer)	$f(x)$ x^2	pselect, $\frac{f_i}{\sum f}$	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

Schema Processing

		Before Reproduction	
		String Representatives	Schema Average Fitness $f(H)$
H_1	1 * * * *	2,4	469
H_2	* 1 0 * *	2,3	320
H_3	1 * * * 0	2	576

serve that for schema H_1, we do obtain the expected exponentially increasing number of schemata as predicted by the schema theorem.

So far, so good; but schema H_1 with its single fixed bit seems like something of a special case. What about the propagation of important similarities with longer defining lengths? For example consider the propagation of the schema H_2 = *10** and the schema H_3 = 1***0. Following reproduction and prior to crossover the replication of schemata is correct. The case of H_2 starts with two examples in the initial population and ends with two copies following reproduction. This agrees with the expected number of copies, $m(H_2) = 2 \cdot 320/293 = 2.18$, where 320 is the schema average and 293 is the population average fitness. The case of H_3 starts with a single example (string 2) and ends with two copies following reproduction (strings 2 and 3 in the string copies column). This agrees with the expected number of copies $m(H_3) = 1 \cdot 576/293 = 1.97$, where 576 is the schema's average fitness and 293 is the population's average fitness. The circumstances following crossover are a good bit different. Notice that for the short schema, schema H_2, the two copies are maintained even though crossover has