chapter three

Evolutionary Computation Concepts and Paradigms

One of the component methodologies of computational intelligence, and the one we believe provides its foundation, is evolutionary computation. This chapter goes into some detail in reviewing the field of evolutionary computation, which consists of machine learning optimization and classification paradigms that are roughly based on evolution mechanisms such as biological genetics, natural selection, and emergent adaptive behavior. Evolutionary computation paradigms provide tools to build intelligent systems that model intelligent behavior.

This chapter also provides basic information needed to use evolutionary computation tools to solve practical problems. The terminology and key concepts are presented, followed by paradigms that are developed from and illustrate the key concepts. The chapter is written largely from the perspective of an engineer or computer scientist, emphasizing the application potential of evolutionary computation tools and drawing comparisons with other applied problem-solving techniques. ■

History of Evolutionary Computation

There are a number of ways to address the history of almost any subject, evolutionary computation included. We choose to focus on people rather than theory or technology for two main reasons. First, it seems a more interesting way to look at history. History is, after all, just a record of people doing things. Second, the evolutionary computation field, particularly in the early days, revolved around a few key individuals. These individuals and their followers seem to us to have sometimes resembled minicultures.

Having said that, the selection of individuals is somewhat arbitrary because the intent is to provide a broad sample of people, rather than an exhaustive list, who contributed to current technology. Some well-known researchers are mentioned only briefly, and others are omitted. The fact that someone is discussed only briefly, or even omitted altogether, is not meant to reflect the authors' opinion of that person's contribution. The selected people and their contributions are discussed roughly in chronological order. We organize our history according to the main evolutionary computation areas.

The evolutionary computation field considered in this book includes the following five areas[1]:

- Genetic algorithms
- Evolutionary programming
- Evolution strategies
- Genetic programming
- Particle swarm optimization

Of the five methodologies, more work has been done in genetic algorithms than in any other area, and so we focus on that field. (We realize that the emphasis on genetic algorithms is fading somewhat. In fact, hybrids of the five methodologies are becoming increasingly popular.) Contributors to the other four areas are also discussed but in somewhat less detail. Although it might be argued that work in the early twentieth century on Darwinian synthesis by Haldane (1990) and others is the place to start, what is now known as evolutionary computation really began to take shape about 50 years later. We begin our journey looking at the roots of genetic algorithms in the 1950s.

Genetic Algorithms

The development of genetic algorithms (GA) has its origins in work done in the 1950s by biologists using computers to simulate natural genetic systems. One of

[1] There are other ways to look at the field, such as considering genetic programming as a branch of genetic algorithms, but we choose this approach.

those doing work most closely related to our current concepts of genetic algorithms was A. S. Fraser, an Australian who began publishing in the field in the late 1950s (Fraser 1957). Our history of evolutionary computation thus (arbitrarily) begins with him.

Fraser was working in the area of epistasis (suppression of the effect of a gene) and represented each of three parameters of an epistatic function as 5 bits in a 15-bit string. He then based his selection of "parents" by choosing those strings whose variable values produced function values between −1 and +1. Fraser was working with natural systems, and although his work somewhat resembles function optimization as currently done by genetic algorithms, he apparently did not consider the possibilities of applying his methodology to artificial systems (Fraser 1960, 1962).

Also beginning to publish in the early 1960s was the man who, together with his students, has probably had more influence on the field of genetic algorithms than any others: John H. Holland of The University of Michigan. Holland attended MIT as an undergraduate, where he was influenced by such luminaries as Norbert Weiner and John McCarthy. He was part of a team that programmed the prototype of the IBM 701 to "learn" something about running a maze, prompting Holland to regard the computer as a sort of "simulated lab rat." After working at IBM, Holland went to the University of Michigan, where, under Arthur Burks, he obtained the first Ph.D. in the United States in computer science (Levy 1992).

Davis (1991) stated:

> John Holland . . . created the genetic algorithm field. The field would not exist if he had not decided to harness the power inherent in genetic processes in the early 1970s and functioned as the technical and political leader of the genetic algorithm field from its inception to the present time. Our understanding of the unique features of genetic algorithms has been shaped by the careful and insightful work of Holland and his students from the field's critical first years to the present time. (p. vi)

Holland's interest is in machine intelligence, and he and his students developed and applied the capabilities of genetic algorithms to artificial systems. He taught courses in adaptive systems in the early 1960s while laying the groundwork for applications to artificial systems with his publications on adaptive systems theory (Holland 1962). Holland's systems were adaptive because of their robustness in spite of changes and uncertainty in the environment. Further, they were self-adaptive in that they could make adjustments based on their interaction with the environment over time.

The GA metaphor is genetic inheritance at the level of the individual. A problem solution is considered as an individual's chromosome, or pattern of genetic alleles, and low-level operations such as those in the nuclei of cells are proposed for developing new solutions.

One of Holland's many contributions was his use of a population of individuals, conceptualized as chromosomes, in the search process, rather than single

individuals, as was common at the time. (Fraser used populations but, as stated previously, didn't apply his methodology to artificial systems.) He also derived the schema theorem, which shows that schema (fundamental building blocks of individual chromosomes) that are more "fit" with respect to a defined fitness function are more likely to reproduce in successive generations of the population of chromosomes. We go into more detail about the schema theorem later in this chapter.

Chromosomes in nature are formed of twisted strands of DNA, composed of the four proteins adenine, cytosine, guanine, and thymine. These strands are presently understood as a kind of computer program that gives instructions to the cells that comprise the organism; the DNA sequence contains instructions about how to develop and what to do. While our digital computers use the base-2, or binary, number system to encode program instructions and data, chromosomes use a base-4 method, encoded in the ordering of the four proteins. Genetic algorithms usually use base-2 chromosomes, though the methods developed by Holland and his followers can be applied to any base number system, including floating-point decimals.

Beginning in the 1960s Holland's students routinely used selection, crossover, and mutation in their applications. Several of Holland's students made significant contributions to the genetic algorithm field, often starting with their Ph.D. dissertations. We mention only a few.

The term *genetic algorithm* was used first by Bagley (1967) in his dissertation, which utilized genetic algorithms to find parameter sets in evaluation functions for playing the game of Hexapawn, which is played on a 3 × 3 chessboard on which each player starts with three pawns. Bagley's genetic algorithm resembled many used today, with selection, crossover, and mutation.

In 1975, Holland published one of the field's most important books, entitled *Adaptation in Natural and Artificial Systems.* In the first five years after it was published, the book sold 100 to 200 copies per year and seemed to be fading into oblivion. Instead, between 1985 and 1990, the number of people working on genetic algorithms—and interest in Holland's book—increased sufficiently to persuade Holland to update and reissue it (Holland 1992).

Also in 1975, K. A. De Jong, one of Holland's students, published his Ph.D. dissertation entitled, "An Analysis of the Behavior of a Class of Genetic Adaptive Systems." As part of his work, De Jong put forward a set of five test functions designed to measure the performance of any genetic algorithm. Two metrics were devised, one to measure the convergence of the algorithm, the other to measure the ongoing performance. De Jong examined the effects of varying four parameters (population size, crossover probability, mutation probability, and generation gap) on the performance of six main kinds of genetic algorithm paradigm (De Jong 1975). Although a number of other benchmark functions have emerged, De Jong's five-function test

bed and two performance metrics are still among frequently referenced criteria for genetic algorithm performance.

From Michigan De Jong went to the University of Pittsburgh, where he taught genetic algorithms to a number of students, among them Steve Smith and John Grefenstette. Smith published a significant dissertation on machine learning involving a classifier system that became known as "Smith's Poker Player" (Smith 1980). After graduation, Grefenstette began teaching yet another generation of students at Vanderbilt University, including J. David Schaffer, who was the first to develop a multiobjective algorithm (Schaffer 1984), work that has enjoyed a revival in popularity.

Grefenstette developed a genetic algorithm implementation called GENESIS that, in its various incarnations and reincarnations, became perhaps the most widely used genetic algorithm implementation in the late 1980s (Grefenstette 1984a, 1984b). He also was instrumental in founding and editing the proceedings of the first International Conference on Genetic Algorithms, a premier conference in the field (Grefenstette 1985).

David E. Goldberg, another of Holland's students, has concentrated on engineering applications of genetic algorithms. He is a former gas pipeline worker whose Ph.D. dissertation considered a 10-compressor, 10-pipe, steady-state, serial gas pipeline problem (Goldberg 1983). The goal was to provide a strategy that minimizes the power consumed in the pumping stations, subject to pressure-related constraints. He summarized the power the genetic algorithm brought to the pipeline problem when he wrote, "If we were, for example, to search for the best person among the world's 4.5 billion people as rapidly as the GA, we would only need to talk to four or five people before making our near optimal selection" (Goldberg 1987). Goldberg's 1989 volume is one of the most influential books on genetic algorithms: *Genetic Algorithms in Search, Optimization and Machine Learning* (Goldberg 1989). He continues to be an important contributor to the field.

The author of another significant genetic algorithm book is self-taught in genetic algorithms. Lawrence (Dave) Davis got interested in them while working at Texas Instruments, where he obtained support to evaluate genetic algorithms for 2D bin packing in a chip layout application. He published the *Handbook of Genetic Algorithms* after moving to the Boston area, where he worked for BBN. His book comprises two main parts. The first is a tutorial on genetic algorithms; the second is a collection of case studies contributed by a number of researchers (Davis 1991). In the mid-1990s, two of the most widely read books by people wanting to learn about genetic algorithms were those by Goldberg and Davis.

At approximately the same time that Holland and his students were developing genetic algorithms, two groups were working on opposite sides of the Atlantic on different approaches that do not use *crossover*, a main feature of genetic algorithm

implementations. These approaches are evolutionary programming and evolution strategies. We begin with evolutionary programming.

Evolutionary Programming

In the United States, Larry J. Fogel and his colleagues developed what they named *evolutionary programming*. Evolutionary programming uses the selection of the fittest, but the only structure-modifying operation allowed is mutation—there is no crossover. Fogel and his colleagues mainly worked with finite state machines and were interested in machine intelligence; they were able to solve some problems that were quite difficult for genetic algorithms.

Fogel (1994) described evolutionary programming as taking a fundamentally different approach from that of genetic algorithms:

> The procedure abstracts evolution as a top-down process of adaptive behavior, rather than a bottom-up process of adaptive genetics. It is argued that this approach is more appropriate because natural selection does not act on individual components in isolation, but rather on the complete set of expressed behaviors of an organism in light of its interaction with its environment.

Philosophically, then, evolutionary programming researchers consider each point in the population to represent an entire species, with species competing to fill environmental niches.

Fogel summarizes evolutionary programming as implementing "survival of the more skillful" rather than the "survival of the fittest" emphasized by genetic algorithm developers. In the mid-1960s a book documenting this approach proved to be quite controversial (Fogel et al. 1966). Misunderstandings and misinterpretations related to the book have been identified as a contributing factor to problems experienced by researchers in obtaining funding for evolutionary computation in the late 1960s and 1970s (Goldberg 1989). It is probable, however, that another significant factor was the well-known symbolics versus numerics controversy (temporarily won by Minsky and the symbolics researchers). One of the leading evolutionary programming researchers during the 1970s was at New Mexico State University. Don Dearholt and his students were responsible for a significant number of publications on evolutionary programming during this decade.

Evolution Strategies

At the same time that Fogel and his group were working on evolutionary programming, across the Atlantic Ocean Ingo Rechenberg and Hans-Paul Schwefel were experimenting with mutation in their attempts to find optimal physical configurations for a series of hinged plates in a wind tunnel and a tube that delivered liquid—the usual gradient-descent techniques were unable to solve the sets of

equations for reducing wind resistance. They began experimenting with mutation, slightly perturbing their best problem solutions to search randomly in the nearby regions of the problem space.

Rechenberg and Schwefel used the first computer available at the Technical University of Berlin to simulate various versions of the approach that became known as *evolution strategies* (Rechenberg 1965; Schwefel 1965). In the early 1970s, Rechenberg published a book that is considered the foundation for this approach (Rechenberg 1973), and evolution strategies continue to experience significant activity, especially in Europe. Research developments in Germany and the United States continued in parallel, with each group unaware of the other's findings until the 1980s (although they may have known about each other [Fogel 2000]).

Genetic Programming

The fourth major area of evolutionary computation is genetic programming. Some of the earliest related work (Friedberg 1958; Friedberg et al. 1959) dealt with fixed-length computer programs that were coded by another program designed to optimize their performance. Their programs, dubbed "Herman" and "Ramsey," each comprised a set of 64 instructions, with each instruction being 14 bits long. The programs were defined such that every arrangement of the 14 bits was a valid instruction, and each set of 64 instructions was a valid program. Unfortunately, the results of the efforts did not live up to expectations; and, in retrospect, there were probably three main reasons for this. First, the programs were limited in length to 64 instructions: A "failure" was tallied if the program did not terminate successfully by the end of the 64th instruction (even if there was a loop). Second, there was only one program; thus, there was a population of just one that evolved. Third, it is not clear that the fitness function used was appropriate.

These limitations were successfully dealt with by Stanford's John Koza (yet another former student of Holland), who developed genetic programming in its current form in the late 1980s. Whereas the other three evolutionary computation approaches use string-shaped chromosomes, Koza evolved computer programs in a population of tree-shaped ones. The units used for crossover were LISP symbolic expressions that are essentially subroutines. Koza has been a prolific producer of documentation, including books (Koza 1992) and videotapes related to genetic programming, which is one of the fastest-growing and most fascinating areas of evolutionary computation. The idea of evolving computer programs has been around for decades; it is now becoming a reality.

Particle Swarm Optimization

The fifth major area of evolutionary computation is the "new kid on the block," particle swarm optimization, which has roots in three main component areas.

Perhaps most obvious are its ties to artificial life (A-life) in general and to bird flocking, fish schooling, and swarming theory in particular. It is also related to evolutionary computation, with ties to both genetic algorithms and evolution strategies (Bäck 1995). The third component area is social psychology. This brief history focuses on three of the main contributing paradigms from social psychology. The A-life and evolutionary computation roots are reviewed in the introduction to the section on particle swarm optimization later in this chapter.[2]

The first social psychology paradigm is Latané's dynamic social impact theory (Latané 1981). Summarized, this theory states that the behaviors of individuals can be explained in terms of the self-organizing properties of their social system, that clusters of individuals develop similar beliefs, and that subpopulations diverge from one another (polarize). There are four major characteristics of social impact theory: consolidation, clustering, correlation, and continuing diversity. Consolidation means that opinion diversity is reduced as individuals are exposed to majority arguments. Clustering means that individuals become more like their neighbors in social space. Correlation means that attitudes that were originally independent tend to become associated. Finally, continuing diversity means that clustering prevents minority views from complete consolidation. In summary, individuals influence one another and, in doing so, become more similar, and patterns of belief held by individuals tend to correlate within regions of a population. This theory is consistent with findings in the fields of social psychology, economics, and anthropology.

The second paradigm is Axelrod's culture model (Axelrod 1984). In this model, populations of individuals are represented as strings of symbols, or "features." The probability of interaction between two individuals is a function of their similarity, and individuals become more similar as a result of their interactions. The observed dynamic is polarization, that is, homogeneous subpopulations that differ from one another.

The third paradigm is Kennedy's adaptive culture model (Kennedy 1998). In this model, there is no effect of similarity of individuals on the probability of their interaction. In fact, the effect of similarity is negative in that it is *dis*similarity that creates boundaries between cultural regions. Interactions between individuals occur if their fitnesses are different. Kennedy's work in culture and cognition can be summarized as follows:

- Individuals searching for solutions learn from the experiences of others (individuals learn from their neighbors).
- An observer of the population perceives phenomena of which the individuals are the parts (individuals that interact frequently become similar).

[2] For a more detailed account of all three component areas, see Kennedy, Eberhart, and Shi (2001).

- Culture affects the performance of individuals that comprise it (individuals gain benefit by imitating their neighbors).

Jim Kennedy and Russ Eberhart both worked at Research Triangle Institute in North Carolina in the early 1990s. Kennedy was interested in exploring the possibility that an evolutionary computation paradigm might play a role in his modeling of social systems. The two continued to collaborate even after Kennedy moved to Washington, D.C., and Eberhart moved to Indianapolis (both moved in 1994). The first two papers were published in 1995 (Kennedy and Eberhart 1995, Eberhart and Kennedy 1995). One was delivered in Nagoya, Japan; the other, in Perth, Australia. The international flavor of the work in the field continues. As of the writing of this book, the authors are aware of work being done in over 30 countries on particle swarm optimization.

Toward Unification

As the 1980s came to a close, the first four areas of evolutionary computation continued to develop relatively independently, with little cooperation or communication among them. In 1994, however, an important meeting was held that brought together researchers from all four evolutionary computation areas: the IEEE World Congress on Computational Intelligence, held at Walt Disney World, Florida. The World Congress comprised a mini-symposium on computational intelligence and three conferences: The International Conference on Neural Networks; the fuzzy logic conference (FUZZ/IEEE 1994); and the First IEEE Conference on Evolutionary Computation (ICEC), chaired by Zbigniew Michalewicz of the University of North Carolina at Charlotte. A total of 96 papers were presented orally in ICEC and 63 in poster sessions, representing authors from 23 countries worldwide. The two volumes of proceedings from this evolutionary computation conference are a landmark in the field (Michalewicz et al. 1994).

At the second World Congress, held in Anchorage, Alaska, in 1998, particle swarm optimization joined the program. The third World Congress, held in Honolulu, Hawaii, featured a significant number of papers from each of the five main areas, as well as interesting and promising hybrids. Researchers in the five areas of evolutionary computation are now communicating and working significantly more with each other.

Now that we've looked at the history of evolutionary computation, let's look at what it is and how to use it.

Evolutionary Computation Overview

The five areas of evolutionary computation (EC) share attributes and implementation procedures, which we now discuss before moving on to separate overviews

of each area. EC paradigms generally differ from traditional search and optimization paradigms in three main ways:

1. EC paradigms utilize a population of points (potential solutions) in their search.
2. EC paradigms use direct "fitness" information instead of function derivatives or other related knowledge.
3. EC paradigms use stochastic, rather than deterministic, transition rules.

In addition, EC implementations sometimes encode the parameters in binary or other symbols, rather than working with the parameters themselves. We now examine these differences in more detail, beginning with the attributes of EC paradigms.

EC Paradigm Attributes

How do traditional optimization methods differ from EC paradigms? Most traditional optimization paradigms move from one point in the decision hyperspace to another, using some deterministic rule. One of the drawbacks of this approach is the likelihood of getting stuck at a local optimum. For example, if the fitness landscape resembles some hills surrounding a mountain that represents the optimum, it is likely that a traditional paradigm will get stuck at the top of a hill and never find the mountain (global optimum). EC paradigms, on the other hand, start with a population of points (hyperspace vectors). They typically generate a new population with the same number of members each epoch, or generation. Thus, many maxima or minima can be explored simultaneously, lowering the probability of getting stuck. Operators such as crossover and mutation effectively enhance this parallel search capability, allowing the search to directly "tunnel through" from one promising hyperspace region to another. (An operator is a rule for changing a proposed problem solution.)

Evolutionary computation paradigms do not require information that is auxiliary to the problem, such as function derivatives. Many hill-climbing search paradigms, for example, require the calculation of derivatives in order to explore the local maximum. In EC optimization paradigms the fitness of each member of the population is calculated from the value of the function being optimized, and it is common to use the function output as the measure of fitness. Fitness is a direct metric of the individual population member's performance on the function being optimized.

The fact that EC paradigms use probabilistic transition rules certainly does not mean that a strictly random search is being carried out. Rather, stochastic operators are applied to operations that direct the search toward regions of the hyperspace that are likely to have higher values of fitness. Thus, for example,

reproduction (selection) is often carried out with a probability that is proportional to the individual's fitness value.

Some EC paradigms, particularly genetic algorithms, use special encodings for the parameters of the problem being solved. In genetic algorithms, the parameters are often encoded as binary strings, but any finite alphabet can be used. These strings are almost always of fixed length, with a fixed total number of 1s and 0s, in the case of a binary string, being assigned to each parameter. By "fixed length" it is meant that the string length does not vary during the running of the EC paradigm. The string length (number of bits for a binary string) assigned to each parameter depends on its maximum range for the problem being solved and on the precision required.

Now that we've discussed the attributes of the paradigms, let's see how to implement them.

Implementation

Regardless of the paradigm implemented, evolutionary computation applications often follow a similar procedure:

1. Initialize the population.
2. Calculate the fitness for each individual in the population.
3. Reproduce selected individuals to form a new population.
4. Perform evolutionary operations, such as crossover and mutation, on the population.
5. Loop to step 2 until some condition is met.

Initialization is commonly done by seeding the population with random values. When the parameters are represented by binary strings, this simply means generating random strings of 1s and 0s (with a uniform probability for each value) of the fixed length described earlier. It is sometimes feasible to seed the population with "promising" values that are known to be in the hyperspace region relatively close to the optimum. (Based on our experience, however, we caution you against using this approach. Randomly generated populations tend to be more reliable.) The number of individuals chosen to make up the population is both problem and paradigm dependent, but it is often in the range of a few dozen to a few hundred.

The fitness value is often proportional to the output value of the function being optimized, though it may also be derived from some combination of a number of function outputs. The fitness function takes as its inputs the outputs of one or more functions, and then it outputs some probability of reproduction. Sometimes it is necessary to transform the function outputs to produce an appropriate fitness metric; sometimes it is not.

Selection of individuals for reproduction to constitute a new population (often called a new generation) is usually based on fitness values. The higher the fitness, the more likely it is that the individual will be selected for the new generation. Some paradigms that are considered evolutionary, however, such as particle swarm optimization, can retain all population members from epoch to epoch.

Now that we've discussed the step-by-step process, let's consider the process as a whole. In many, if not most, cases, a global optimum exists at one point in the decision hyperspace. (Sometimes multiple optima exist.) Furthermore, stochastic or chaotic noise might be present. Occasionally the global optimum changes dynamically because of external influences; frequently there are very good local optima as well. For these and other reasons, the bottom line is that it is often unreasonable to expect any optimization method to find a global optimum (even if it exists) within a finite time. The best that can be hoped for is to find near-optimum solutions and that the time it takes to find them increases less than exponentially with the number of variables. We agree with one leading EC researcher who suggests that the focus should be on "meliorization" (improvement) rather than on optimization (Schwefel 1994).

Put another way, evolutionary computation is often the second-best way to solve a problem. Classical methods such as linear programming should often be tried first, as should customized approaches that take full advantage of knowledge about the problem. (It is also possible that a hybrid approach that uses elements from classical methods with elements of evolutionary computation will work well.)

Why should we be satisfied with second best? For one thing, classical and customized approaches are frequently not feasible, while EC paradigms are feasible in a vast number of situations. Also, a real strength of EC paradigms is that they are generally quite robust. In this field, robustness means that an algorithm can be used to solve many problems, and even many kinds of problems, with a minimum amount of special adjustments to account for special qualities of a particular problem. Typically an evolutionary algorithm requires specification of the length of the problem solution vectors, some details of their encoding, and an evaluation function; the rest of the program does not need to be changed. Finally, robust methodologies are generally fast and easy to implement. This is especially true of EC paradigms, which are often one or more orders of magnitude faster than other approaches (if other approaches exist).

We've completed our overview of evolutionary computation. The next sections review five areas of evolutionary computation: genetic algorithms, evolutionary programming, evolution strategies, genetic programming, and particle swarm optimization. Genetic algorithms, discussed in the next section, receive a majority of the attention, as they currently account for most of the successful applications in the literature (although this is changing).

Genetic Algorithms

It seems that every technology has its jargon, and genetic algorithms are no exception. Therefore, we begin by reviewing some of the basic terminology that is needed to understand the genetic algorithm (GA) literature. A sample problem is then presented to illustrate how GAs work; a step-by-step analysis illustrates a GA application, with options discussed for some of the individual operations. The section concludes with a more detailed look at the fundamental Schema theorem and at approaches for improving GA performance in some situations.

In this book, unless otherwise specified, we deal with canonical genetic algorithms, a basic version of GAs that feature binary parameter encoding, one- or two-point crossover, and bit-by-bit mutation. (We discuss these attributes later in this section.)

Details of implementing GAs are discussed in Chapter 4, where a specific GA implementation is summarized. We begin here by looking at the general features of GAs.

Overview of Genetic Algorithms

One perspective of genetic algorithms is that they are search algorithms that reflect in a very primitive way some of the processes of natural evolution. (As such, they are analogous to artificial neural networks' status as primitive approximations of biological neural processing.) Engineers and computer scientists do not care as much about the biological foundations of GAs as about their utility as analysis tools (another parallel with neural networks). GAs often provide very effective search mechanisms that can be used in optimization or classification applications.

EC paradigms work with a population of points rather than a single point; each "point" is actually a vector in hyperspace representing one potential, or candidate, solution to the optimization problem. A population is thus just an ensemble, or set, of hyperspace vectors. Each vector is called an individual in the population; sometimes an individual in a GA is referred to as a chromosome because of the analogy to genetic evolution of organisms.

Because real numbers are often encoded in GAs using binary numbers, the dimensionality of the problem vector might be different from the dimensionality of the bitstring chromosome. The number of elements in each vector (individual) equals the number of real parameters in the optimization problem. A vector "element" generally corresponds to one parameter, or dimension, of the numeric vector. Each element can be encoded in any number of bits, depending on the representation of each parameter. The total number of bits defines the dimension of hyperspace being searched. If a GA is being used to find "optimum" weights for a neural network, for example, the number of vector elements equals the number

of weights in the network. If there are w weights, and it is desired to calculate each weight to a precision of b bits, then each individual will consist of $w \cdot b$ bits, and the dimension of the binary hyperspace being searched is 2^{wb}. Thus we can see that even for a fairly modest problem involving the optimization of three variables to a resolution of three decimal places each (10 bits), the search space is 2^{30}. The variables being optimized comprise what is called the phenotype space, and the behavior of the system given certain values of the variables is the *phenotype*. The binary strings on which operators such as crossover and mutation work comprise what is called the *genotype space*, and the strings themselves are the *genotypes*.

The series of operations carried out when implementing a canonical (basic) GA paradigm is:

1. Initialize the population.
2. Calculate fitness for each individual in the population.
3. Reproduce selected individuals to form a new population.
4. Perform crossover and mutation on the population.
5. Loop to step 2 until some condition is met.

In some GA implementations, operations other than crossover and mutation are carried out in step 4. We will further explore GAs by applying a basic GA to a simple problem.

A Sample GA Problem

Because implementing a canonical (basic) GA paradigm is so simple, a sample problem (also simple) seems to be the best way to introduce most of the basic GA concepts and methods. As will be seen, implementing a basic GA involves only copying strings, exchanging portions of strings, and flipping bits in strings.

Our sample problem is to find the value of x that maximizes the function $f(x) = \sin(\pi x / 256)$ over the range $0 \leq x \leq 255$, where values of x are restricted to integers. This is just the sine function from zero to π radians, as illustrated in Figure 3.1. Its maximum value of 1 occurs at $\pi / 2$, or $x = 128$. The function value and the fitness value are thus defined to be identical for the sample problem.

There is only one variable in our sample problem: x. We assume for the sample problem that the GA paradigm uses a binary alphabet. The first decision to be made is how to represent the variable. It is easy in this case because the variable can only take on integer values between 0 and 255. It is therefore logical to represent each individual in our population with an 8-bit binary string. Using standard binary encoding, the binary string 00000000 will evaluate to 0; 11111111, to 255.

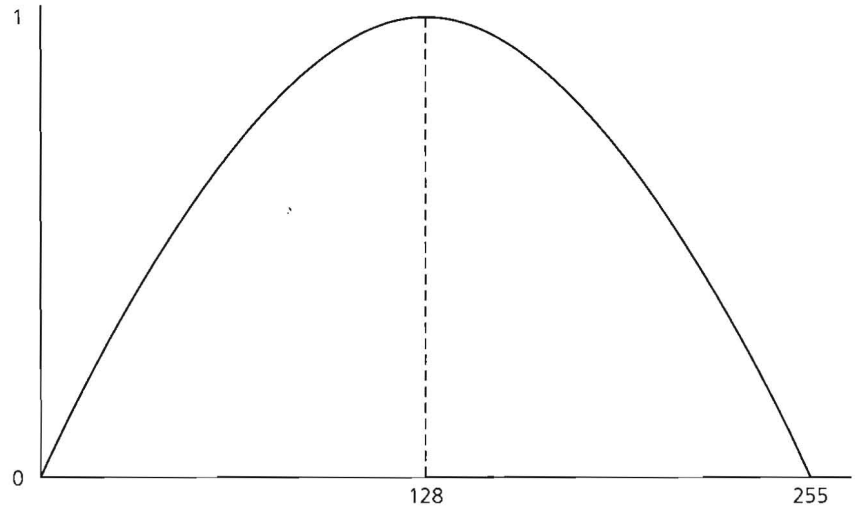

Figure 3.1 Function to be optimized in example problem.

The determination of the number of bits needed is usually more complex than this case. There is generally more than one variable, and the number of bits for each variable must be chosen to yield the desired precision. For example, a real variable that varies between 0 and 1 and has a precision of three decimal places (one part in a thousand) can be represented by a string of 10 bits (one part in 1,024).

We must decide next how many individuals will make up the population. In an actual application, it is common to have between a few dozen and a few hundred individuals. For the purposes of this illustrative example, however, the population consists of eight individuals.

The next step is to initialize the population, which is usually done randomly. A random number generator is thus used to assign a 1 or 0 to each of the eight positions in each of the eight individuals, resulting in the initial population in Figure 3.2. Also shown in the figure are the values of x and $f(x)$ for each binary string.

After fitness calculation, the next step is reproduction. Reproduction consists of forming a new population with the same number of individuals by selecting from members of the current population with a stochastic process that is weighted by each of their fitness values. In the sample problem, the sum of all fitness values for the initial population is 5.083. Dividing each fitness value by 5.083, then, yields a *normalized fitness* value f_{norm} for each individual. The sum of the normalized values is, of course, 1. The normalized values are shown in an accumulated fashion in the *cumulative* f_{norm} column in Figure 3.2.

Individuals	x	$f(x)$	f_{norm}	cumulative f_{norm}
1 0 1 1 1 1 0 1	189	0.733	0.144	0.144
1 1 0 1 1 0 0 0	216	0.471	0.093	0.237
0 1 1 0 0 0 1 1	99	0.937	0.184	0.421
1 1 1 0 1 1 0 0	236	0.243	0.048	0.469
1 0 1 0 1 1 1 0	174	0.845	0.166	0.635
0 1 0 0 1 0 1 0	74	0.788	0.155	0.790
0 0 1 0 0 0 1 1	35	0.416	0.082	0.872
0 0 1 1 0 1 0 1	53	0.650	0.128	1.000
	$\Sigma f(x) = 5.083$			

Figure 3.2 Initial population and $f(x)$ values for GA example.

These normalized fitness values are used in a process called "roulette wheel" selection, where the size of the roulette wheel wedge for each population member, which reflects the probability of the individual being selected, is proportional to its normalized fitness value.

The roulette wheel is "spun" by generating eight random numbers between 0 and 1. If a random number is between 0 and 0.144, the first individual in the existing population is selected for the next population. If it is between 0.144 and $(0.144 + 0.093) = 0.237$, the second individual is selected, and so on. Finally, if the random number is between $(1 - 0.128) = 0.872$ and 1.0, the last individual is selected. The probability that an individual is selected is thus proportional to that individual's fitness value. It is possible, though highly improbable, that the individual with the lowest fitness value could be selected eight times in a row and make up the entire next population. It is more likely that individuals with high fitness values are picked more than once for the new population. (Note that roulette wheel selection works as described here only when all fitness values are positive. Modifications must be made to accommodate negative fitness values.)

The eight random numbers generated (presented in random order) are 0.293, 0.971, 0.160, 0.469, 0.664, 0.568, 0.371, and 0.109. As shown in Figure 3.3, this results in initial population member numbers 3, 8, 2, 5, 6, 5, 3, and 1 being chosen to make up the population after reproduction.

The next operation is crossover. To many evolutionary computation practitioners, crossover of binary encoded substrings is what makes a genetic algorithm a genetic algorithm. Crossover is the process of exchanging portions of the strings of two "parent" individuals. An overall probability is assigned to the crossover process, which is the probability that, given two parents, the crossover process will occur. This *crossover rate* is often in the range of 0.65 to 0.80; a value of 0.75 is selected for the sample problem.

First, the population is divided randomly into pairs of parents. Because the order of the population after reproduction in Figure 3.3 is already randomized,

```
0 1 1 0 0 0 1 1
0 0 1 1 0 1 0 1
1 1 0 1 1 0 0 0
1 0 1 0 1 1 1 0
0 1 0 0 1 0 1 0
1 0 1 0 1 1 1 0
0 1 1 0 0 0 1 1
1 0 1 1 1 1 0 1
```

Figure 3.3 Population after reproduction.

0 1 1|0 0 0|1 1	0 1 1 1 0 1 1 1	119	0.994
0 0 1|1 0 1|0 1	0 0 1 0 0 0 0 1	33	0.394
1|1 0 1 1|0 0 0	1 0 1 0 1 0 0 0	168	0.882
1|0 1 0 1|1 1 0	1 1 0 1 1 1 1 0	222	0.405
0 1|0 0 1 0 1|0	1 0 0 0 1 0 1 0	138	0.992
1 0|1 0 1 1 1|0	0 1 1 0 1 1 1 0	110	0.976
0 1 1 0 0 0 1 1	0 1 1 0 0 0 1 1	99	0.937
1 0 1 1 1 1 0 1	1 0 1 1 1 1 0 1	189	0.733
(a)	(b)	(c)	(d)

Figure 3.4 Population before crossover showing crossover points (a); after crossover (b); and values of x (c) and $f(x)$ (d) after crossover.

parents will be paired as they appear there. For each pair, a random number is generated to determine whether crossover will occur. It is thus determined that three of the four pairs will undergo crossover.

Next, for the pairs undergoing crossover, two crossover points are selected at random. (Other crossover techniques are discussed later in this chapter.) The portions of the strings between the first and second crossover points (moving from left to right in the string) will be exchanged. The paired population, with the first and second crossover points labeled for the three pairs of individuals undergoing crossover, is illustrated in Figure 3.4(a) before the crossover operation. The portions of the strings to be exchanged are in bold. Figure 3.4(b) illustrates the population after crossover is performed.

Note that, for the third pair from the top, the first crossover point is to the right of the second. The crossover operation thus "wraps around" the end of the string, exchanging the portion between the first and the second, moving from

left to right. For two-point crossover, then, it is as if the head (left end) of each individual string is joined to the tail (right end), thus forming a ring structure. The section exchanged starts at the first crossover point, moving to the right along the binary ring, and ends at the second crossover point. The values of x and $f(x)$ for the population following crossover appear in Figure 3.4(c) and (d), respectively.

The final operation in this plain vanilla genetic algorithm is mutation. Mutation consists of flipping bits at random, generally with a constant probability for each bit in the population. As is the case with the probability of crossover, the probability of mutation can vary widely according to the application and the preference of the researcher. Values between 0.001 and 0.01 are not unusual for the mutation probability. This means that the bit at each site on the bitstring is flipped, on average, between 0.1 and 1.0 percent of the time. One fixed value is used for each generation and is often maintained for an entire run.

As there are 64 bits in the sample problem's population (8 bits × 8 individuals), it is quite possible that none will be altered as a result of mutation, so the population of Figure 3.4(b) will be taken as the "final" population after one iteration of the GA procedure. Going through the entire GA procedure one time is said to produce a new generation. The population of Figure 3.4(b) therefore represents the first generation of the initial randomized population.

Note that the fitness values now total 6.313, up from 5.083 in the initial random population, and that there are now two members of the population with fitness values higher than 0.99. The average and maximum fitness values have thus both increased. It is important to note that in most GA applications the fitnesses don't monotonically increase. There are times when the children have lower fitnesses than their parents. If this situation continues, however, the individuals with lower fitness will probably be eliminated through the selection process.

The population of Figure 3.4(b) and the corresponding fitness values in Figure 3.4(d) are now ready for another round of reproduction, crossover, and mutation, producing yet another generation. More generations are produced until some stopping condition is met. The researcher may simply set a maximum number of generations for the algorithm to search, may let it run until a performance criterion has been met, or may stop it after some number of generations with no improvement.

This completes our simple application of the basic GA. It's time to back up and review the GA's operations.

Review of GA Operations in the Simple Example

Now that one iteration of the GA operations (one generation) for the sample problem has been completed, each operation is reviewed in more detail. Various approaches, and reasons for each, are examined.

The representation of the values for the variable x was made (perhaps unrealistically) straightforward by choosing a dynamic range of 256; an 8-bit binary number was thus an obvious approach. Standard binary coding, however, is only one approach; others may be more appropriate.

In this example, the nature of the sine function places the optimal value of x at 128, where $f(x)$ is 1. The binary representation of 128 is 10000000; the representation of 127 is 01111111. Thus, the smallest change in fitness value can require a change of every bit in the representation. This situation is an artifact of the encoding scheme and is not desirable—it only makes the GA's search more difficult. Often, a better representation is one in which adjacent integer values have a Hamming distance of 1; in other words, adjacent values differ by only a single bit.

Gray coding overcomes this impediment while retaining the advantages of binary operations (Gray 1953). The challenge is to devise a scheme, using 0s and 1s, to encode integers where the Hamming distance between adjacent numbers equals 1; this is called the "adjacency property." There are many ways to accomplish this for any length bitstring; the most commonly used version is called "binary-reflected Gray code." As shown Table 3.1, Gray coded integers that are one unit different in value are also one unit distant in Hamming distance.

Table 3.1 Gray Codes and Binary Codes for Integers 0–15

Integer	Binary code	Gray code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

The algorithm for generating Gray code from binary is quite simple. The length of the Gray bitstring will be the same length as the binary version. Further, the leftmost bit will be the same. Starting at the second position from the left, then, the formula is

$$G_i = XOR(B_i, B_{i-1})$$

where G_i is the bit in the ith position of the Gray code (G_1 is the leftmost bit); B_i is the bit in the ith position of the binary code; and the function $XOR()$ returns 1 if the adjacent bits are different from one another, 0 if they are the same. In other words, set the most significant bit on the Gray bitstring equal to the same bit on the binary bitstring, and move to the right. Where a bit matches the bit to the left of it on the binary bitstring, place a 0 in the Gray bitstring; otherwise, place a 1. Go down the line doing this at each position. With Gray coding, a movement of one unit on the number line is performed by flipping a single bit, allowing an optimizer to climb more gracefully toward optima.

Some GA software allows the user to specify the dynamic range and resolution for each variable. The program then assigns the correct number of bits and the coding. For example, if a variable has a range from 2.5 to 6.5 (a dynamic range of 4) and it is desired to have a resolution of three decimal places, the product of the dynamic range and the resolution requires a string 12 bits long, where the string of 0s represents the value 2.5. A major advantage of being able to represent variables in this way is that the user can think of the population individuals as real-valued vectors rather than as bit strings, thus simplifying the development of GA applications.

This kind of representation can present some challenges. If, for instance, the dynamic range is 5 and resolution is 3 decimal places, we need 13 bits (same as for dynamic range of 8) and some of the bitstrings resulting from crossover and mutation will not be within the dynamic range. Provisions have to be made to take care of such situations. One approach is to define "repair" functions that move population members that are outside of the dynamic range back in. Another approach is to assign particularly high penalties to locations outside the dynamic range.

The "alphabet" used in the representation can, in theory, be any finite alphabet. Thus, rather than use the binary alphabet of 1 and 0, we could use an alphabet containing more characters or numbers. Engineers frequently represent variables with real numbers. Many GA implementations, however, use the binary alphabet.

Turning our attention to the size of the population, De Jong's dissertation (1975) offers guidelines that are still usually observed: Start with a relatively high crossover rate, a relatively low mutation rate, and a moderately sized population (though just what constitutes a moderately sized population is unclear). The main trade-off is obvious: A large population will search the space more thoroughly

but at a higher computational cost. The authors have generally used populations of between 20 and 200 individuals, depending primarily, it seems, on the string length of the individuals. It also seems (in the authors' experience) that the sizes of populations tend to increase approximately linearly with individual string length rather than exponentially, but "optimal" population size (if an optimal size exists) depends on the problem as well.

The initialization of the population is usually done stochastically, though it is sometimes appropriate to start with one or more individuals that are selected heuristically. The GA is thereby initially aimed in promising directions, or given hints. It is not uncommon to seed the population with a few members selected heuristically and to complete it with randomly chosen members. Regardless of the process used, the population should represent a wide assortment of individuals. The urge to skew the population significantly should generally be avoided if the limited experience of the authors is generalizable.

The calculation of fitness values is conceptually simple, though it can be quite complex to implement in a way that optimizes the efficiency of the GA's search of the problem space. In the sample problem, the value of $f(x)$ varies (quite conveniently) from 0 to 1. Lurking within the problem, however, are two drawbacks to using the "raw" function output as a fitness function: one that is common to many implementations, the other arising from the nature of the sample problem.

The first drawback common to many implementations is that after the GA has been run for a number of generations it is not unusual for most (if not all) of the individuals' fitness values, after, say, a few dozen generations, to be quite high. In cases where the fitness value can range from 0 to 1, for example (as in the sample problem), most or all of the fitness values may be 0.9 or higher. This lowers the fitness differences among individuals that provide the impetus for effective roulette wheel selection; relatively higher fitness values should have a higher probability of reproduction.

One way around this problem is to space the fitness values equally. For example, in the sample problem the fitness values used for reproduction could be equally spaced from 0 to 1, assigning a fitness value of 1 to the most fit population member, 0.875 to the second, and 0.125 to the least fit of the eight. In this case the population members are ranked on the basis of fitness and then their ranks are divided by the number of individuals to provide a probability threshold for selection. Note that the value of 0 is often not assigned, since that would result in one population member being made ineligible for reproduction. Also note that $f(x)$, the function result, is now not equal to the fitness and that, in order to evaluate actual performance of the GA, the function value should be monitored as well as the spaced fitness.

Another way around the problem is to use what is called *scaling*. Scaling takes into account the recent history of the population and assigns fitness values on the basis of comparison of individuals' performance to the population's recent average performance. When the GA optimization is maximizing some function,

scaling involves keeping a record of the minimum fitness value obtained in the last w generations, where w is the size of the scaling window. If, for example, $w = 5$, the minimum fitness value in the last five generations is kept and used, instead of 0, as the "floor" of fitness values. Fitness values can be assigned a value based on their actual distance from the floor value, or they can be equally spaced, as described earlier.

The second drawback is that the sample problem exacerbates the compression of fitness values situation described earlier because near the global optimum fitness value of 1, $f(x)$ (which is also the fitness) is relatively flat. There is thus relatively little selection advantage for population members near the optimum value $x = 128$. If this situation is known to exist, a different representation scheme might be selected, such as defining a new fitness function, which is the function output raised to some power.

What we have been talking about with respect to both drawbacks is *selection pressure*, or how much reproduction advantage is given to population members with higher fitness values. Too much pressure (advantage) can result in premature convergence, and not enough may allow the population to wander aimlessly.

Note that the shape of some functions "assists" discrimination near the optimum value. For example, consider maximizing the function $f(x) = x^2$ over the range 0 to 10; there is a higher differential in values of $f(x)$ between adjacent values of x near 10 than near 0. Thus a slight change in the independent variable results in great improvement or deterioration of performance—which is equally informative—near the optimum.

In the discussion thus far, we have assumed that optimization implies finding a maximum value. Sometimes, of course, optimization requires finding a minimum value. Many versions of GA implementations allow for this possibility. Often, it is required that the user specify the maximum value f_{max} of the function being optimized, $f(x)$, over the range of the search. The GA can then be programmed to maximize the fitness function $f_{max} - f(x)$. In this case, scaling, described previously, keeps track of f_{max} over the past w generations and uses it as a "roof" value from which to calculate fitness.

We now consider roulette wheel selection. In genetic algorithms, the expected number of times each individual in the current population is selected for the new population is proportional to the fitness of that individual relative to the average fitness of the entire population. Thus, in the initial population of the sample problem, where the average fitness was 5.083/8 = 0.635, the third population member had a fitness value of 0.937, so it could be expected to appear about 1.5 times in the next population; it appeared twice.

The conceptualization is that of a wheel whose surface is divided into wedges representing the probabilities for each individual (see Figure 3.5). For instance, one point on the edge is determined to be the zero point and each arc around the circle corresponds to an area on the number line between 0 and 1. A random number

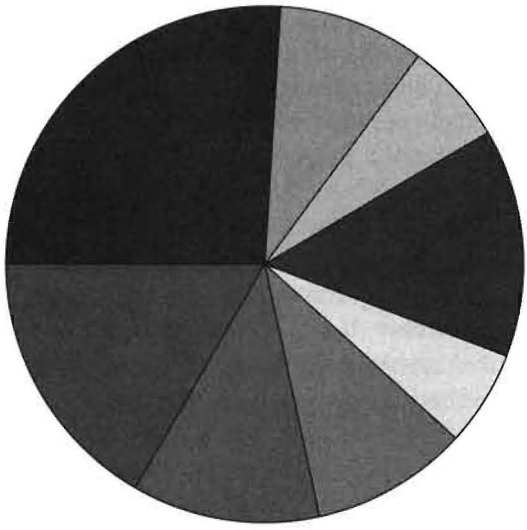

Figure 3.5 Roulette wheel selection, in which the probability of an individual being selected is proportional to its fitness.

is generated, between 0.0 and 1.0, and the individual whose wedge contains that number is chosen. In this way, individuals with greater fitness are more likely to be chosen. The selection algorithm can be repeated until the desired number of individuals has been selected. There are a number of variations to the roulette wheel procedure. A few of them are reviewed next.

One variation on the basic roulette wheel procedure is a process developed by Baker (1987) in which the portion of the roulette wheel is assigned based on each unique string's relative fitness. One spin of the roulette wheel then determines the number of times each string will appear in the next generation. To illustrate how this is done, assume that the fitness values are normalized (sum of all equals 1). Each string is assigned a portion of the roulette wheel proportional to its normalized fitness. Instead of one "pointer" on the roulette wheel spun n times, there are n pointers spaced $1/n$ apart; the n-pointer assembly is spun only once. Each of the n pointers now points to a string; each place one of the n pointers points determines one population member in the next generation. If a string has a normalized fitness greater than $1/n$ (corresponding to an expected value greater than 1), it is guaranteed at least one occurrence in the next generation.

In the discussion thus far, we have assumed that all of the population members are replaced each generation. Although this is usually the case, sometimes it is desirable to replace only a portion of the population—for example, the 80 percent with the worst fitness values. The percentage of the population replaced each generation is sometimes called the *generation gap*.

Unless some provision is made, with standard roulette wheel selection it is possible that the individual with the highest fitness value in a given generation may not survive reproduction, crossover, and mutation to appear unaltered in the new generation. It is frequently helpful to use what is called the *elitist strategy*, which ensures that the individual with the highest fitness is always copied into the next generation. Most GA applications with which the authors are familiar implement elitist strategy.

The most important operator in GAs is crossover, based on the metaphor of sexual combination. Its purpose is to pass on information from population member to population member. If a solution is encoded as a bitstring, then mutation may be implemented by setting a probability threshold and flipping bits when a random number is less than the threshold. As a matter of fact, mutation is not considered by most GA practitioners to be an especially important operator in GA; it is usually set at a very low rate and sometimes omitted. Crossover is generally considered more important because it is considered to play a more important role in guiding the population toward an acceptable solution.

Crossover is a term for the recombination of genetic information during sexual reproduction. In GAs, offspring have equal probabilities of receiving any gene from either parent because the parents' chromosomes are combined randomly. In nature, chromosomal combination leaves sections intact—that is, contiguous sections of chromosomes from one parent are combined with sections from the other, rather than simply shuffling randomly. In GAs there are many ways to implement crossover.

The two main attributes of crossover that can be varied are the type of crossover that is implemented and the probability that it occurs. The following paragraphs examine variations of each.

A crossover probability of 0.75 was used in the sample problem, and two-point crossover was implemented. Two-point crossover with a probability of 0.60 to 0.80 is a relatively common choice, especially when Gray coding is used.

The most basic crossover type is *one-point crossover*, as described by Holland (1992) and others, for example, Goldberg (1989), and Davis (1991). It is inspired by natural evolution processes. One-point crossover involves selecting a single crossover point at random and exchanging the portions of the individual strings to the right of the crossover point. Figure 3.6 illustrates one-point crossover; portions to be exchanged are in bold in Figure 3.6(a).

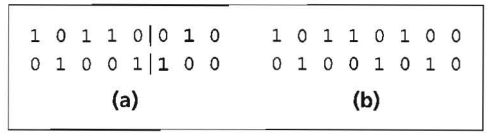

Figure 3.6 One-point crossover before (a) and after (b) crossover.

Another type of crossover that has been found useful is *uniform crossover*, described by Syswerda (1989). A random decision is made at each bit position in the string as to whether or not to exchange (cross over) bits between the parent strings. If a 0.50 probability at each bit position is implemented, an average of about 50 percent of the bits in the parent strings are exchanged. Note that a 50 percent rate will result in the maximum disruption due to uniform crossover. Higher rates just mirror rates lower than 50 percent. For example, a 0.60 probability uniform crossover rate produces results identical to a 0.40 probability rate. If the rate were 100 percent, the two strings would simply switch places, and if it were 0 percent neither would change.

Values for the probability of crossover vary with the problem. In general, values between 60 and 80 percent are common for one-point and two-point crossover. Uniform crossover sometimes works better with slightly lower crossover probability. It is also common to start out running the GA with a relatively higher value for crossover, then taper off the value linearly to the end of the run, ending with a value of, say, one-half the initial value.

Inversion is a GA operation that is not generally used today. It is functionally related to crossover, but involves a single parent producing a single child. Figure 3.7 illustrates the process, which consists of switching end for end a portion of the parent structure, shown between the cut points in bold in Figure 3.7(a), in the child. One reason it is not in general use is that it is perceived to destroy the basic building blocks, or schemata, by inverting them. The term *schemata* usually refers to substrings of an individual population member string; a more detailed description appears in the next section, Schemata and the schema theorem.

In GAs, mutation is the stochastic flipping of bits that occurs in each generation. Its purpose is to introduce diversity into the population and is generally done bit by bit on the entire population. It is often done with a probability of something like 0.001, but higher probabilities are not unusual. For example, Liepins and Potter (1991) used a mutation probability of 0.033 in a multiple-fault diagnosis application.

If the population comprises real-valued parameters, mutation can be implemented in different ways. For instance, in an image classification application, Montana (1991) used strings of real-valued parameters that represented thresholds of event detection rules as the individuals. Each parameter in the string was range-limited and quantized (i.e., could take on only a certain finite number of values). If chosen for mutation, a parameter was randomly assigned any allowed value in the range of values valid for that parameter.

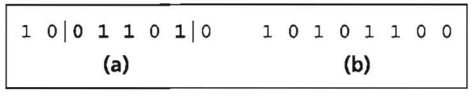

Figure 3.7 Example of string before (a) and after (b) inversion operation.

The probability of mutation is often held constant for the entire run of the GA, although this approach does not produce optimal results in many cases. It can be varied during the run and, if varied, usually is increased. For example, mutation rate may start at 0.001 and end at approximately 0.01 when the specified number of generations has been completed. In the software implementation described on this book's web site, a flag in the run file can be set that increases the mutation rate significantly when the variability in fitness values becomes low, as is often the case late in the run.

Selecting the number of generations for which the GA is run is often a trial-and-error process. In general, given enough computing time, the number of generations is adjusted until the desired response is obtained. Other factors, such as population diversity and fitness improvement of the best population member, can enter into the decision to end the GA run. For example, if the best fitness has not changed for, say, 100 generations, we may choose to terminate the run.

The optimum number of generations is often a function of the problem. For instance, if the GA is being used to train a neural network, the same caveats apply as would apply if any neural network paradigm such as back-propagation were being used. What is desired is optimum results with a test set, so conditions such as overtraining must be avoided.

Whatever the application, given the stochastic nature of a GA, multiple runs will probably be desirable. Then the best-performing individuals from each run can be tested.

This completes our review of basic GA operations. In the next section, we consider a theorem that provides some insight into how GAs work.

Schemata and the Schema Theorem

Exactly how do GAs do what they do? How is it possible to develop new population members that, on average, are fitter than the previous generation while searching new regions of the problem space? Since all that GAs have to work with are (often binary) strings, there must be features related to the fitness inherent in the strings that are used.

The string features that are relevant to the optimization process are called *schemata* (singular: *schema*). The *schema theorem* describes why the canonical GA paradigm is able to efficiently direct an optimization process. (This theorem also applies to other proportional selection methodologies.)

First described for the GA field by Holland (1975, 1992), schemata are similarity templates for strings. Each schema defines a subset of strings with identical values at specified string locations. As used here, the word *string* usually refers to substrings of an individual population member string, but it can refer to the entire string. Schemata provide a means by which relevant similarities among the individual population members can be described and exploited.

To define schemata, the alphabet of the strings is used to define values at specified locations, and an additional character is used as a "don't care" symbol in locations where the value doesn't matter. As is common in the GA literature, the pound symbol (#) is used in this book as the "don't care" symbol. Schemata can thus generally be thought of as comprising an alphabet of $a_o + 1$ characters, where a_o is the number of characters in the GA representation. In most cases, as in the example, the GA strings have a binary representation, so the schemata comprise the characters {0, 1, #}.

As an example, consider the schemata of length 4 that may appear in, say, the leftmost four positions of the population strings of the sample problem. One such schema is #000, which has two member strings. That is, two strings match the schema: 1000 and 0000. The schema 1##0 has four matching strings: 1000, 1010, 1100, and 1110.

Holland argues that adaptation can be thought of in terms of schemata. Genetic optimization increases the likelihood that the schemata that most improve the species' fitness will persist to the next generation. He also argues that crossover among the fittest members of a population will result in the discovery and survival of better schemata.

It should be noted that some researchers have recently found errors in Holland's argument, and the issue is currently controversial. Even if the proof is shaky, it can be observed empirically, simply by running GA programs, that crossover is quite effective, if not always fast, for finding good solutions to highly complex problems.

How many schemata are possible for a string length of l and an alphabet of a_o characters? In the previous example, for $a_o = 2$, there can be a 0, 1, or # at each string position, resulting in a total possible number of schemata of $3 \times 3 \times 3 \times 3 = 81$. Generalizing, there are $(a_o + 1)^l$ total possible schemata for any representation of length l.

Another informative measure is the total number of unique schemata possible in a population. Consider a specific string of length 8, taken from the example problem: 01110111. Since each string position can assume the value it has or the wild-card value, the string belongs to $2^8 = 256$ schemata. Any binary string of length l thus belongs to 2^l schemata. In a population of n individuals, then, there are between 2^l (if all members are identical) and $n2^l$ (if no two individuals are the same) schemata. Populations with higher diversity have a greater number of schemata.

Schemata that are part of an individual with high fitness have a higher than average probability of reproducing. Therefore, highly fit schemata benefit from differential reproduction relative to fitness. If selection were the only operator used, though, no new regions of the search hyperspace would ever be explored. Crossover and mutation provide new schemata to guide the search into new regions.

Crossover is a slightly more complicated matter than reproduction. Consider two schemata: ##1####0 and ###10###. If both are part of strings of equal

fitness, which is more likely to be passed on to the new population? Either one- or two-point crossover is more likely to disrupt the first, since it is quite likely that a crossover point will occur between the two string endpoints. The second is more compact and less likely to be disrupted by a one- or two-point crossover operation.

Mutation is not likely to disrupt either schema, since it typically occurs at a very low rate. And since it is considered on a bit-by-bit basis, if it does occur it is just as likely to disrupt one as the other.

Although crossover and mutation are potentially disruptive, they facilitate an efficient search by introducing innovations. Furthermore, compact (short) schemata that are part of highly fit individuals will, with high probability, appear in ever-increasing numbers in future generations. The schemata are the elements from which future generations are built; Holland (1992) named them "building blocks." The schema theorem sums up all of this and provides a quantitative estimation of one aspect of GA performance.

The schema theorem predicts the number of times a specific schema will appear in the next generation of a GA, given the fitness of the population member(s) containing the schema, the average fitness of the population, and other parameters. The GA can be thought of as effectively working with a large number of schemata simultaneously, ranging from very short schemata to schemata as long as the individual population members. This has been named "intrinsic parallelism" by Holland. The schema theorem provides a quantitative prediction for all schemata, regardless of length. It should be noted that the theorem applies only to "plain vanilla" GAs. As soon as you do anything special, including something as simple as implementing elitism, where the fittest population member is automatically copied into the next generation, the schema theorem no longer applies.

The schema theorem appears as equation 3.1.[3]

$$n_{t+1}(S) \geq n_t(S)\frac{f(S)}{f_{\text{avg}}}\left[1 - p_c\frac{\delta(S)}{l-1} - o(S)p_m\right] \tag{3.1}$$

In equation 3.1, n is the total number of examples of a particular schema S. The subscripts $t+1$ and t refer to time steps, or generations. The parameter $f(S)$ is the average fitness of the individual population members that contain the schema S, while f_{avg} is the average fitness of the entire population. The probabilities of crossover and mutation are p_c and p_m, respectively.

The parameter $\delta(S)$ is called the "defining length" of the schema; it is the distance between the first and last specific string positions. For example, for the schema #01#11#, the defining length is 4. The total length of the string is l, while $o(S)$ is the "order" of the schema, or the number of fixed positions (1s and 0s) in

[3] The derivation of the theorem is beyond the scope of this book. The reader is referred to the derivation in Goldberg (1989).

the schema. In the preceding example, the order of the schema is 4. The defining length of a schema is just the number of potential "cut" points within the schema that could be affected by crossover.

Summarized, equation 3.1 states that the expected number of occurrences of schema S in generation $t + 1$ is the number in the current generation multiplied by the average schema fitness divided by the average population fitness, less the disruptive effects caused by crossover and mutation. Schemata with above-average fitness values will be represented an increasing number of times as generations proceed. Those with below-average values will be represented less and less; they will "die out," just as happens in nature.

The schemata with small values for defining length are disrupted least by crossover, so the most rapidly increasing representation in any population will be of highly fit, short schemata, called building blocks, which will experience exponential growth. Building blocks illustrate that it is often beneficial to keep some parts of a solution intact. This is the most important consequence of the schema theorem.

Note that the schema theorem, by itself, does not specify how well a GA will solve a particular problem. It should also be noted that there is controversy in the EC community with respect to the usefulness and validity of the theorem. We include it, as have other recent books dealing with GAs such as (Mitchell 1996), (Pedrycz 1998), and (Haupt and Haupt 1998), because we believe it provides useful insights into GA processes.

We've now told you what we think you need to know about GAs, how they work, and how to apply them to practical problems. All that is left are a few final observations.

Comments on Genetic Algorithms

In sum, a genetic algorithm operates by evaluating a population of bitstrings (there are real-numbered GAs, but binary implementations are more common) and selecting survivors stochastically based on their fitness; thus, fitter members of the population are more likely to survive. Survivors are paired for crossover, and often some mutation is performed on chromosomes. Other operations might be performed as well, but crossover and mutation are the most important ones. Sexual recombination of genetic material is a powerful method for adaptation.

In Chapter 2, we discussed three spaces of adaptation: the parameter space, the function space, and the fitness space. Much of the literature in evolutionary computation treats the function space as if it were identical to the fitness space; that is, the function output provides a number that indicates how close to the global optimum the search algorithm is. There are, however, dangerous ambiguities in the confusion of these two quantities. The fitness landscape can be very different depending on the fitness function utilized. The fitness measure should probably

be scaled between 0 and 1 when possible, making it easy to understand as well as an indication of the probability of a population member's survival.

The material on genetic algorithms in this chapter provides only an introduction to the subject. We suggest that you explore GAs further by sampling the references cited in this section. With further study and application, it will become apparent why GAs have such a devoted following. In the words of Davis (1991):

> [T]here is something profoundly moving about linking a genetic algorithm to a difficult problem and returning later to find that the algorithm has evolved a solution that is better than the one a human found. With genetic algorithms we are not optimizing; we are creating conditions in which optimization occurs, as it may have occurred in the natural world. One feels a kind of resonance at such times that is uncommon and profound.

This feeling, of course, is not unique to experiences with GAs; using other evolutionary algorithms can result in similar feelings. An implementation of a genetic algorithm is presented in Chapter 4. The software for the GA implementation is on the book's web site.

That's it for genetic algorithms. Let's now turn our attention to an evolutionary computation paradigm that eschews crossover—evolutionary programming.

Evolutionary Programming

Evolutionary programming (EP) is similar to genetic algorithms in its use of a population of candidate solutions to evolve an answer to a specific problem; it differs in its concentration on top-down processes of adaptive behavior. The emphasis in evolutionary programming is on developing behavioral models, that is, models of observable system interactions with the environment. Theories of natural evolution heavily influence the development of evolutionary programming concepts and paradigms.

Evolutionary programming is derived from the simulation of adaptive behavior in evolution: GAs are derived from the simulation of genetics. The difference is perhaps subtle but important. Genetic algorithms work in the genotype space of the information codings, while evolutionary programming (EP) emphasizes the phenotype space of observable behaviors (Fogel 1990). EP is therefore directed at evolving "behavior" that solves the problem at hand; it mimics "phenotypic evolution."

Evolutionary programming is a more flexible approach to evolution than some of the other paradigms. Operators are freely adapted to fit the problem at hand. Generally, the paradigm relies on mutation—not sexual recombination—to produce offspring. Whereas evolution strategies systems usually generate many more

offspring than parents (a ratio of seven to one is common, as we will see in the next section), EP usually generates the same number of children as parents. Parents are selected to reproduce using a tournament method; their features are mutated to produce children who are added to the population. When the population has doubled, the members—parents and offspring together—are ranked, and the best half are kept for the next generation.

Now that we have a rough idea of what EP entails, let's see how to implement it in an application. After that, we'll look at examples of specific application areas.

Evolutionary Programming Procedure

The process for implementing EP will look familiar to you; the process itself is similar to the one we used for GAs. The procedure generally followed when implementing EP appears in the following list:

1. Initialize the population.
2. Expose the population to the environment.
3. Calculate fitness for each member.
4. Randomly mutate each "parent" population member.
5. Evaluate parents and children.
6. Select members of the new population.
7. Go to step 2 until some condition is met.

The population is randomly initialized. For problems in real (computable) space, each component variable of each individual's vector is generally a real value that is constrained to some dynamic range. In the two EP examples that follow, the variables (vector elements) represent finite state machine parameters and function variables, respectively. The number of population members is problem dependent, but is often a few dozen to a few hundred, as in to GA populations.

To better understand the remaining steps in the EP procedure, two examples are examined. These two examples are representative of two main types of problem to which EP paradigms are often applied. The first involves time series prediction using a finite state machine. The second is the optimization of a mathematical function.

Finite State Machine Evolution for Prediction

Remember that prediction is one of the attributes of computational intelligence systems we discussed in Chapter 2. Evolutionary programming paradigms are sometimes used for problems involving prediction. One way to represent prediction of

the environment is with a sequence of symbols. As with GAs, the symbols must be members of a finite alphabet. A system comprising a finite state machine, for example, can be used to analyze a symbol sequence and to generate an output that optimizes a fitness function, which often involves predicting the next symbol in the sequence. In other words, a prediction is used to calculate a system response that seeks to achieve some specified goal.

A *finite state machine* is defined as "a transducer that can be stimulated by a finite alphabet of input symbols, can respond in a finite alphabet of output signals, and possesses some finite number of different internal states" (Fogel 1991). The input and output symbol alphabets need not be identical. The initial state of the machine must be specified. It is also necessary to specify, for each state and input symbol combination, the output symbol and next state. Table 3.2 specifies a three-state finite state machine with an input alphabet of two characters and three possible output symbols.

Finite state machines are essentially a subset of Turing machines, developed by the English mathematician and computer science pioneer Alan Turing (1937). Turing machines are capable, in principle, of solving all mathematical problems (of a defined general class) in sequence. Finite state machines, as used in EP, can model, or represent, an organism or system.

Unlike GAs, where crossover is an important component of producing a new generation, mutation is the only operator used in EP systems. Each member of the current population typically undergoes mutation to produce a "child." Given the specification of the finite state machine, and its operation, five main types of mutation can occur: As long as more than one state exists, the initial state can be changed and/or a state can be deleted. A state can be added. A state transition can be changed. Finally, an output symbol for a given state-input symbol can be changed.

Although the number of children produced by each parent is a system parameter, each "parent" typically produces one "child," and the population becomes twice its original size after mutation. After measuring the fitness of each structure, the best half are kept, maintaining the population size at a constant value from

Table 3.2 Specification Table for a Three-State Finite State Machine

Existing state	A	A	B	B	C	C
Input symbol	1	0	1	0	1	0
Output symbol	Y	Y	X	Z	Z	Y
Next state	A	B	C	B	A	B

Source: Fogel (1991).

generation to generation. At some point in some applications, it is necessary to predict the next symbol in a sequence. The structure with the highest fitness is chosen to generate this new symbol, which is then added to the sequence. (It is also possible to specify the problem so that the symbol predicted is farther in the future than one time step.)

Unlike other evolutionary paradigms, in EP systems mutation can change the size of structures (states can be added and deleted). This fact and the potential for changing state transitions lead to another consideration: The specification table for a finite state machine can have unfilled blanks. There can be mutations that add states that are never utilized in a given problem; Fogel (1991) calls these "neutral mutations." It is also possible to create the situation via mutation where a specified state transition is not possible because the new state has been deleted. These mutations and others, such as changing output symbols, tend to have less effect the more states the machine has, but can still cause fatal errors in the finite state machine if they are not handled properly.

Although Fogel (1995) usually allows a variable-length structure, it is also possible to evolve a finite state machine with EP using a fixed structure. First, the maximum number of states must be determined. For purposes of illustration, using the three-state machine defined earlier as an example, we will assume that no more than four states are allowed.

Each state can then be represented by a fixed 5-bit binary element as follows. The first bit could represent the "activation" of the state: if it is 1, the state is active; if 0, the state is inactive (that is, it does not exist). The next two bits can represent the output symbol (X, Y, or Z) for an input of 0, and the final two bits can represent the output symbol for an input of 1. (Note that our example above has only three output symbols. With binary representation, we have to allow for four and handle a nonexistent symbol the way nonexistent states are handled.) We thus require a total element length of $(1 + n_i * b_o)$ bits, where n_i is the number of possible inputs and b_o is the number of bits needed to represent the output symbols.

The population in our example is thus initialized with individuals 20 bits long. For the example it may be a good idea to specify that only individuals with at least two active states can be allowed in the initial population.

A child is now generated for each parent. Given the five possible kinds of mutation outlined earlier, one possible mutation procedure is:

1. For each individual, generate a random number from 0 to 1.
2. If the number is between 0.0 and 0.2, change the initial state; if between 0.2 and 0.4, delete a state, etc.
3. The mutation selected in step 2 is done with a flat probability across all possibilities. For example, if the initial state is to be changed and there

are *a* active states, then one active state is selected to be the initial state; each active state has the probability of $1/a$ of being selected.

4. Infeasible state transitions are modified to be feasible. If a state transition to an inactive state has been specified, one of the active states is selected to be the object of the transition. As above, each active state has the probability of $1/a$ of being selected.
5. Evaluate fitnesses and keep the best 50 percent, resulting in a new population of the same size.

This scenario is only one of many possibilities. For example, it might be desirable to lower the probability ranges (the ranges between 0 and 1 in step 2) for adding and deleting states and correspondingly increase the mutation probability ranges for changing input symbols and/or output symbols. It is also possible to *evolve* the ranges, number of states, and so on.

One example of finite state machines is the development by Fogel (1995) using evolutionary programming of finite state machines that do very well at playing Axelrod's prisoner's dilemma game. As described in Kennedy, Eberhart, and Shi (2001):

> The prisoner's dilemma is a situation where two interacting players have opposite, symmetrical motives. Each player has the choice to cooperate or compete with the opponent: if both cooperate, their payoffs are high, and if both compete payoffs are low. If one competes (the technical term is defecting) while the other cooperates, the defector receives a very high reward while the cooperator's payoff is very low—the lowest in the game, called the "sucker's payoff." When the game is played just one time, the most reasonable thing to do is to defect, as there is no basis for trusting the other player, and there is nothing to gain by being a sucker.
>
> Usually though, the game is iterated, a series of games is played. A player would score the highest if he always defected while his partner always cooperated—but of course no sensible player would continue to cooperate while being hammered repeatedly by a competitive opponent. Repeated trials require some consideration of strategy, for instance, a player might end up with the highest score if he lulled his opponent into cooperating, then struck with a defection, then lulled and defected, and so on. It might be that the best approach would be just to cooperate from the start—except that nothing then prevents the opponent from taking advantage. The simple game then produces opportunities for many kinds of strategies. Axelrod roughly grouped these into two kinds: "nice" strategies, which rely on cooperation to keep the level of payoffs high for both parties, and strategies he refers to as "mean" (specifically that includes only the all-defect strategy) or "not nice." Strategies that are not nice include ones that might try to use cooperation as a way to make the opponent vulnerable, then defect for the higher payoff.

The payoff function is that used by Axelrod (1980): If both cooperate, each player gets 3 points; if both defect, each player gets 1 point; if one defects and one

cooperates, the cooperating player gets no points while the defecting player gets 5 points.

Fogel allowed the finite state machines to have up to eight states. This doesn't represent all possible behaviors *à la* Axelrod, but it does allow a dependence on sequences of greater than third-order. Fogel was able to evolve finite state machines that had average scores slightly greater than 3.0, which is the score that is achieved through mutual cooperation alone.

Figure 3.8 is the diagram for a seven-state finite state machine (one of many evolved by Fogel) to play prisoner's dilemma. The start state is state 6, and play is begun by cooperating. In the table, "C" denotes cooperate and "D" denotes defect.

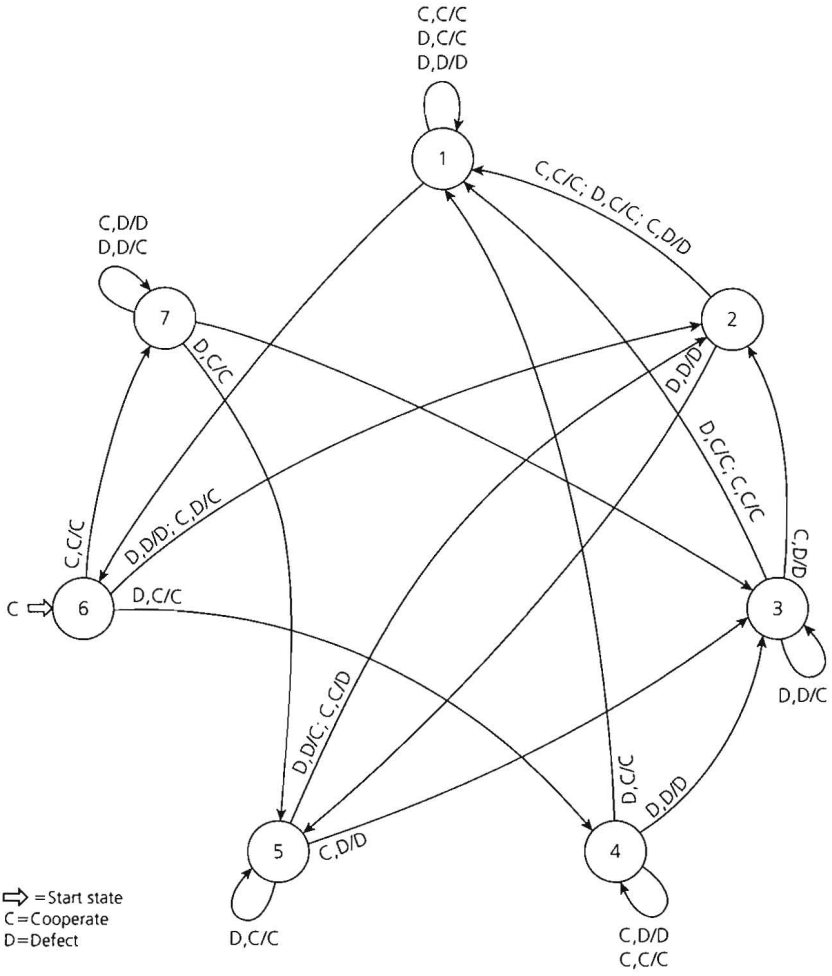

Figure 3.8 A seven-state finite state machine to play prisoner's dilemma. *Source*: Fogel 1995; © IEEE. Used with permission.

The input alphabet comprises [(C,C), (C,D), (D,C), (D,D)], where the first letter represents the finite state machine's previous move and the second the opponent's. So, for example, a label of C,D/C on the arrow leading from state *X* to state *Y* means that if the system is in state *X* and on the previous move the finite state machine cooperated and the opponent defected, then cooperate and transition to state *Y*. Sometimes, more than one situation can result in the same state transition. For example, in Figure 3.8, assume the machine is in state 6, in which case if the machine and opponent both defected on the previous move, the machine defects (D,D/D) and transitions to state 2. Likewise, a transition from state 6 to state 2 occurs if the machine cooperated and the opponent defected on the previous move; the machine cooperates in this case (C,D/C) as it moves into state 2.

Now that we've seen how to apply evolutionary programming to finite state machines used for prediction, let's look at another main area of application, function optimization.

Function Optimization

The second example of a type of problem to which EP paradigms are applied is function optimization. (Remember what we said previously about optimization: Usually we really don't find the optimum and often we don't know much about where it is or if it even exists. What we usually find is sufficiently good solutions to problems.) The following example features the modification of each component of the evolving individual structures with a Gaussian random function.

Consider, for the example, optimizing a function with two variables such as $F(x, y) = x^2 + y^2$. The extremum in this case is a minimum at $x = y = 0$. The first step is to establish a random initial population and then to specify the dynamic range of the two variables. One plausible approach might be to start with an initial population of 50 individuals, each variable of which is initialized randomly over the range $[-5, 5]$. The fitness value of each individual is then calculated. The inverse of the Euclidean distance from the origin is one reasonable fitness measure.

Each "parent" individual is mutated to create one "child." The mutation method used by Fogel (1991) is to add a Gaussian random variable with zero mean and variance equal to the parent's error value (the Euclidean distance from the origin in this example) to each parent vector component. The fitness of each child is then evaluated the same way as the parents'.

The process of mutation is illustrated in equation 3.2:

$$p_{i+k,j} = p_i + N(0, \beta_j \phi_{p_i} + z_j), \forall_j = 1..., n, \tag{3.2}$$

where

$p_{i,j}$ is the j^{th} element of the i^{th} organism

$N(\mu, \sigma^2)$ is a Gaussian random variable with mean μ and variance σ^2

ϕ_{p_i} is the fitness score for p_i
β_j is a constant of proportinality to scale ϕ_{p_i}
z_j represents an offset

For the function used in the example, it has been shown that the optimum rate of convergence is represented by $\sigma = \frac{1.224\sqrt{f(x)}}{n}$, where n is the number of dimensions (Bäck and Schwefel 1993).

Another way to perform mutation involves a process known as self-adaptation. In this variation, the standard deviations (and rotation angles, if used) are modified based on their current values. As a result, the search adapts to the error surface contours (Fogel 1995).

Fitness, however, is sometimes not used directly by itself to decide which half of the augmented population will survive to the next generation. Tournament selection is used, with each individual competing with a number, say 10, of other individuals in the following way.

For each of the 10 competitions with other individuals, a probability of "scoring a point" is set equal to the error score of the opponent divided by the sum of the individual and opponent errors. For instance, if the error of the individual is 2 and that of the opponent (one of 10 opponents) is 3, the probability of scoring a point is 3/5, or 60 percent. The total score is tallied over the 10 competitions for each individual, and the one-half of the population with the highest total scores is selected for the next generation.

This concludes our discussion of using evolutionary programming for optimization. (Keep in mind that, as we discussed previously, we believe that it really isn't optimization most of the time.)

Comments on Evolutionary Programming

The implementation of evolutionary programming concepts seems to vary more from application to application than GA implementations. A number of factors contribute to the differences in approach, but the most important factor seems to be the top-down emphasis of EP. Another is the fact that selection is a probabilistic function of fitness rather than being tied directly to it. One developer of EP (Fogel 1991) stated that EP is at its best when it is used to optimize overall system behavior.

Evolution Strategies

We begin our look at evolution strategies (ES) with the concept of the *evolution of evolution*. As a biological analogy, evolution strategies model problem solutions as species rather as they have been described earlier, as populations of normally distributed multivariate points scattered around a fitness landscape. The aspect of

these populations that permits them to adapt to their environment (in research this is often simulated by a test function or hard optimization problem) is their ability to evolve their own evolvability.

If evolutionary programming is based on evolution, then, reasons Rechenberg (1994), the field of evolution strategies is based on the evolution of evolution. Since biological processes have been optimized by evolution, and evolution is a biological process, then evolution must have optimized itself. Evolution strategies, although utilizing forms of both mutation and crossover (usually called "recombination" in the evolution strategies literature), have a slightly different view of both operations than either evolutionary programming or genetic algorithms.

There are many similarities between evolution strategies and evolutionary programming, and in fact the two paradigms are moving closer together as researchers exchange techniques across the Atlantic. Evolution strategies, like evolutionary programming, take a top-down view. They also stress the phenotypic behavior as opposed to the genotypic. This means, for example, that the phenotypic behavior ramifications of recombination are of importance, rather than what happens to the genotypes. ES paradigms also usually use real values for the variables rather than the binary coding favored in genetic algorithm implementations.

In evolution strategies the goal is to move the mass of the population toward the best region of the landscape. Through application of the simple rule, "survival of the fittest," the best individuals in any generation are allowed to reproduce; their offspring resemble them but with some differences introduced through mutation. An individual is a potential problem solution characterized by a vector of numbers representing phenotypic features. Mutation is performed by adding normally distributed random numbers to the parents' phenotypic coordinates, their position in the search space, so that the next generation of children explores around the area in the landscape that has proven good for their parents.

The amount of mutation—the evolvability of the population—is controlled in an interesting way in ES. An individual is typified by a set of features and by a corresponding set of strategy parameters. These are usually variances or standard deviations (the square root of the variance), though other statistics are sometimes used. The strategy parameters are used to mutate the feature vectors for the individual's offspring; for instance, standard deviations can be used to define the variability of the normal distribution used to perturb the parent's features. Random numbers can be generated from a probability distribution with a mean of zero and a standard deviation defined by the strategy parameters; adding these random numbers to the values in the parent's feature vector simulates mutation in the offspring. They resemble the parents but differ from them to some controlled extent. Since the evolutionary process is applied to the strategy parameters themselves, the range of mutation, or the variability of the changes introduced in the next generation, evolves along with the features that are being optimized.

Intuitively it can be seen that increasing the variance is like increasing the step-size taken by population members on the landscape. High variance equals exploration and wide-ranging search for good regions of the landscape, and it corresponds to a high rate of mutation; low variance is exploitation, focused search within regions. The strategy parameters stochastically determine the size of the steps taken when generating offspring of the individual; a large variance means that large steps are likely to be taken, that the children are likely to differ greatly from their parents. As the children are randomly generated from a normal distribution, though, a large variance *can* produce a small step size, and vice versa. It is known that 68.26 percent of random normal numbers generated fall within one standard deviation, 95 percent will fall within 1.96 standard deviations of the mean, and so on. So widening the standard deviation widens the dispersion of randomly generated points.

Evolution strategies' unique view of mutation includes the concept of an *evolution window*. The theory behind the concept is that mutation operations result in fitness improvement only if they land within a defined step-size band, or window (Rechenberg 1994). Crossover and mutation operations that land outside the evolution window are not helpful. A theoretical derivation of Rechenberg states that if mutations are carried out with an optimal standard deviation, the probability of a "successful" (helpful) mutation is about one-fifth. Evolution strategies carry the idea of the evolution window still further. They assert that dynamic adjustment of the mutation size to a dynamic evolution window can provide benefits called "meta-evolution," or evolution of the second kind (Rechenberg 1994).

Like evolutionary programming, ES employs Gaussian noise functions with zero mean to determine mutation magnitudes for the variables. For the strategic parameters, log normal distributions are sometimes used as mutation standard deviations.

Evolution strategies theory states that mutation rates should be inversely proportional to the number of variables in the individual population member and should be proportional to the distance from the function optimum. In real-world applications, of course, the exact value of the optimum is usually unknown. However, some knowledge often exists about the optimum. It is often known within an order of magnitude, sometimes to within a factor of two or three. Even limited knowledge such as this can be helpful in guiding the evolution strategy search.

In ES, *recombination* manipulates entire variable values. This is usually done using one of two methods. The first and more common method (the local method) involves forming one new individual using components (variables) from two randomly selected parents. The second method, the global method, uses the entire population of individuals as potential sources from which individual components for the new individual can be obtained.

Each of the two methods, local and global, is generally implemented in one of two ways. The first is called *discrete recombination*, which consists of selecting the

parameter value from either parent. In other words, the parameter value in the child equals the value of one parent. The second way, called *intermediate recombination*, involves setting each parameter value for a child at a point between the values for the two parents; typically, the value is set midway between those values. If the parents are denoted by A and B, and the ith parameter is being determined, then the value established using intermediate recombination is $x_i^{new} = x_{A,i} + C(x_{B,i} - x_{A,i})$, where C is a constant, usually set to 0.5 to yield the midpoint between the two parent values.

Thus we see that evolution strategies contain a component representing sexual combination of features. In intermediate recombination, for instance, the children's features are computed as a kind of average of the two parents' features; in discrete recombination, individual features may come intact or mutated from one parent or the other.

In the experience of ES practitioners, the best results often seem to be obtained by using the local version of discrete recombination for the parameter values and the local version of intermediate recombination for the strategy parameter(s). In fact, Bäck and Schwefel (1993) report that implementation of strategy parameter recombination is mandatory for the success of any ES paradigm.

All of this is well and good; we know now how to transform individual population members using recombination and mutation. How, then, do we select the members of the next generation? How do we accomplish selection?

Selection

In evolution strategies, as in all Darwinian models, an individual's fitness determines the probability that it will reproduce in the next generation. There can be many ways to decide this; for instance, we could rank all the individuals from best to worst, chop off the bottom of the list, and save only the proportion that we want to survive. This proportion depends on how many offspring they will have, assuming the population size remains constant from one generation to the next.

In nature, of course, there is no ranking of individuals; the survival of each depends on the environment and that individual's chance encounters. Imagine a snowshoe hare that has a mutation that makes its fur turn black in the winter. In the snow this hare is more visible than its camouflaged cousins. It might just happen, though, that no predators come into the area where this hare lives, so they don't see it and it subsequently reproduces, passing on the mutation. It can happen; it is just that the likelihood is reduced relative to the alternative, which is that a predator that comes into the area immediately notices this contrastive morsel and eats him rather than his harder-to-see littermates. In nature, the measure of fitness has a great amount of error in it; possible improvements are commonly lost.

This suggests that selection needs to be probabilistic—you can't just propagate the best so-many individuals to the next generation. A lesson learned from

simulated annealing is that sometimes a step backward is productive in the long run. In the same way, natural evolution lets some less-fit individuals reproduce, and it is quite likely that eventual improvement is transmitted through the less obvious route. Evolutionary computation researchers have come up with a number of techniques for stochastically selecting survivors for the next generation. In order to better model the stochastic aspect of natural selection—what could be called survival of the luckiest—several computational methods of selection have been devised. Common methods include ranking, roulette wheel selection, and tournament selection.

Ranking is the simplest procedure, though it does not have the advantage of allowing selection of less-fit individuals. The population is sorted from best to worst, and individuals above the cutoff in the list are chosen. One salient objection to this method is that it requires global information. Knowledge of all fitness values is needed in order to determine the rank of any individual. Obviously, nature does not work this way; only local information is used in natural selection, and errors in ranking—occasions where more-fit members fail to reproduce or less-fit members succeed—contribute to the adaptation of the population. This might be a weaker argument than it seems, though; there are plenty of times when a computer needs to use global information in order to accomplish things that nature does without it. For instance, to detect collisions in virtual worlds requires computation of the relative positions of all objects in the world, but in the physical world things behave appropriately without any such computations. Running into a brick wall stops you, period. So evolution in a computer program might be acceptable even if it required global information as a way to accomplish an end.

Roulette wheel selection was discussed in the section on genetic algorithms. Recall that, in roulette wheel selection, each individual is given a probability of selection proportional to its fitness. Tournament selection was discussed in the section on evolutionary programming.

Tournament selection uses local competitions to determine survivors. In its simplest form, individuals are paired at random and the better member of each pair is selected to reproduce. This can be repeated until the next generation is sufficiently populated. Other tournament methods pair up individuals in some number of competitions, adding a point to their score each time they win, and then keep individuals with more than a critical number of points; other methods select subgroups at random from the population and allow the one with the highest fitness to survive to the next generation.

The results of tournament selection correlate with the results of ranking—that is, fitter individuals survive in general. One-on-one, winner-take-all tournaments allow the most error in terms of less-fit individuals being selected; while the very best individual is guaranteed to survive and the very worst is guaranteed not to, it is entirely possible that the next-to-worse individual is paired with the worst one and thus is selected. Repetitive and subgroup tournaments decrease the amount of error while increasing the correlation with ranking results, until an algorithm where

each individual engages in n-1 unique tournaments, where n is the population size, is exactly equivalent to ranking.

Differences exist between evolution strategies and other paradigms of evolutionary computation with respect to selection. ESs generally operate with a surplus of descendants. Schwefel (1994) describes the most common versions of ES selection, known as the (μ, λ) and $(\mu + \lambda)$ ES. In both versions, the number of children generated from μ parents is $\lambda > \mu$. Commonly used is a λ/μ ratio of 7. In the original (1 + 1) ES, one parent produces one offspring, with only the fitter of the two surviving. This version is seldom used now.

The difference between the "plus" and "comma" versions comes in the next step. In the (μ, λ) version, the μ individuals with the highest fitness values out of the λ children are selected. Note that the μ parents are not eligible for selection in this scheme, only the children. In the $(\mu + \lambda)$ version, the best μ individuals are selected from a pool of candidates that includes both the μ parents and the λ children—that is, the union of the two groups of individuals. Whichever method is used, the μ individuals that are left have thus been selected completely deterministically and have equal probabilities to mate and have descendants in the next generation.

The discussion of genetic algorithms mentioned the elitist strategy, in which the individual in each generation with the highest fitness is guaranteed to survive to the next generation. This individual may be carried over from the previous generation or may appear as a result of operations in the current one. As can be seen from the preceding discussion, the $(\mu + \lambda)$ version implements elitism, as the most-fit parent will be retained, while the (μ, λ) version does not. Elitism is generally considered helpful in GA applications. With evolution strategies, however, the (μ, λ) version is generally observed to yield better performance (Bäck and Schwefel 1993).

The following list summarizes the procedure used in most evolution strategies.

1. Initialize population.
2. Perform recombination using the μ parents to form λ children.
3. Perform mutation on all children.
4. Evaluate λ or $\mu + \lambda$ population members.
5. Select μ individuals for the new population.
6. If the termination criterion is not met, go to step 2; otherwise, terminate.

Key Issues in Evolution Strategies

In sum, in evolution strategies mutation is applied to the parent's features to generate children that resemble the parent but differ stochastically from it. Each survivor's positional coordinates are entered as the mean of a normal distribution,

and the corresponding strategy parameter is entered as the variance or standard deviation, and a child vector of numbers is generated for both positions and strategy parameters. These are evaluated, selection is applied, and the cycle repeats. The evolution of strategy parameters suggests the evolution of evolvability, adaptation of the mutability of a species as it searches for, then settles into, a niche.

This completes our review of evolution strategies. Recall that evolutionary programming, the area we discussed just prior to evolution strategies, does not use crossover, only mutation. The area we discuss next, genetic programming, emphasizes crossover, relegating mutation to a minor supporting role. Genetic programming also uses a somewhat different structure than we've seen up to now.

Genetic Programming

The three areas of evolutionary computation discussed thus far have involved individual structures that are defined as strings. Some are strings of binary values and some include real-valued variables, but all are strings, or vectors. The genetic programming (GP) paradigm deals with evolving hierarchical computer programs that are generally represented as tree structures. Furthermore, while individual structures used up to this point have generally been of fixed length, programs being evolved by genetic programming generally vary in size, shape, and complexity.

One perspective is that GPs are a subset of GAs that evolve executable programs. Differences between GPs and generic GAs include:

- Population members are executable structures (generally computer programs) rather than strings of bits and/or variables.
- The fitness of an individual population member in a GP is measured by executing it. (Generic GAs' measure of fitness depends on the problem being solved.)

The goal of a genetic programming implementation is to "discover" a computer program within the space of potential computer programs being searched that gives a desired output for a given set of inputs. In other words, a computer is figuring out how to write its own code.

Each program is represented as a parse tree, where the functions defined for the problem appear at the internal tree points and the variables and constants are located at the external points (leaves). The nature of the computer programs generated makes genetic programming inherently hierarchical.

In preparation for running a genetic programming implementation, five steps are carried out.

1. Specify the terminal set.
2. Specify the function set.
3. Specify the fitness measure.
4. Select the system control parameters.
5. Specify termination conditions.

The terminal set comprises the variables (the system state variables) and constants associated with the problem being solved. For example, consider a "cart centering" problem, where the goal is to center a cart in the least amount of time on a one-dimensional frictionless track by imparting fixed-magnitude forces that accelerate the cart left or right. The variables are the cart's position x and velocity v. A constant such as -1 is also an appropriate terminal for this problem (see Koza 1992, Chapter 6).

The functions selected for the function set are limited only by the programming language implementation used to run the programs evolved by the GP implementation. They can thus include mathematical functions (cos, exp, etc.), arithmetic operations (+, * , etc.), Boolean operators (AND, NOT; etc.), conditional operators such as *if–then–else*, and iterative and recursive functions. Each function in the function set requires a certain (fixed) number of arguments, known as the function's *arity*. (Terminals are functions with arity 0.) One task of specifying the function set is to select a minimal set that is capable of accomplishing the task.

This leads to two properties that are desirable in any GP application: *closure* and *sufficiency*. For the closure property to be satisfied, each function must be able to successfully operate on any function in the function set and on any value of any data type assumable by a member of the terminal set.

This occasionally requires definition of special cases for functions. For example, in arithmetic functions division by 0 can be defined for the purposes of a problem as being equal to some constant value such as 1. If Boolean values returned by conditional operators are not acceptable, the conditional operator can be redefined in one of two ways: (1) Numerical values (such as 0 and 1) can be returned rather than Boolean values (such as F and T), or (2) conditional branching and conditional comparative operators can be defined to execute one of their arguments depending on the evaluation of the test involving an external state or condition or on the comparison test outcome. Functions that are redefined so as to return acceptable values are called protected functions. If the closure property is not satisfied, some method must be specified for dealing with infeasible population members and with members whose fitness is not acceptable.

For the sufficiency property to be satisfied, the set of functions and set of terminals must be sufficiently extensive to allow a solution to be evolved. In other words, some combination of functions and terminals must be capable of producing

a solution. Some knowledge of the problem is generally required to be able to judge when the sufficiency property is met. In some problem domains, sufficiency is relatively easy to determine. For example, if Boolean functions are being used, it is well known that the function set comprising AND, OR, NOT is sufficient for any problem. For other problems, it can be relatively difficult to establish sufficiency.

Having more than the minimally sufficient number of functions has been found to degrade performance somewhat in some cases and to significantly improve it in others. Having too many terminals, however, usually degrades performance (Koza 1992).

The fitness measure often is selected to be inversely proportional to the error produced by program output. Other fitness measures are also common, such as the score a program achieves in the game.

The two main control parameters are the population size and the maximum number of generations that will be run. Other parameters used include reproduction probability, crossover probability, and the maximum size allowed (as measured by the depth, or number of hierarchical levels) in the initial and final program populations.

The termination condition is usually determined by the maximum number of generations specified. The winning program is usually the best program (in terms of the fitness measure) created thus far in any generation.

After the five preparatory steps for running a GP are completed, the GP process can be implemented as follows:

1. Initialize the population of computer programs.
2. Determine the fitness of each individual program.
3. Carry out reproduction according to fitness values and reproduction probability.
4. Perform crossover of subexpressions.
5. Go to step 2 unless termination condition is met.

The population is initialized with randomly generated computer programs comprising functions and terminals from the selected sets. In other words, each program in the initial population is created by building a rooted tree structure with randomly selected functions and terminals from the defined sets. No restrictions are placed on the size or shape (configuration) of acceptable programs, other than the maximum depth, or number of hierarchical levels, allowed. Each structure created is a hierarchically structured executable program. A population of 500 has been reported to be sufficient for most problems solved with GP implementations (Koza 1992).

Figure 3.9 Example of root of randomly created program in initial population. Other functions continue down from the two branches.

The root of each program tree is a function randomly selected from the function set. The root of a randomly created program appears at the top of Figure 3.9. The number of lines, or branches, emanating from the function is equal to its arity. In the figure, the multiplication function "*" takes two arguments.

Once the root function is selected, program population can be created in a number of ways. Following is a description of what Koza (1992) calls the *ramped half-and-half* method. It makes use of two approaches to building program trees: the "grow" method and the "full" method.

In the grow approach, a random selection is made from the combined set of functions *and* terminals for placement at the end of each line emanating from the root function. If a function is selected, program creation continues recursively with selections from the combined set. Whenever a terminal is selected, a leaf, or endpoint, of the tree is established. Program creation along that line is thus terminated. Except for the root function, therefore, all functions are at internal tree locations. The leaves of the tree are all terminals. Any time the maximum depth (number of hierarchical levels) is reached, the random selection is limited to the terminal set. When the grow method is used, the program tree configuration is guided by the ratio of the number of functions to the number of terminals. When the ratio is higher, the average depth of each limb is higher.

In the full approach, each limb of the program tree extends for the full depth. Only functions are selected for placement at the end of each line until the maximum depth is reached, at which time only terminals are selected. All programs created using the full approach thus have identical fully developed structures.

The ramped half-and-half approach produces a population of diverse sizes and shapes. Koza (1992) reports using this method for almost all problems except those involving Boolean functions. The method consists of creating programs with evenly distributed depth parameters ranging from 2 to the maximum depth. For example, if the maximum depth is 5, 25 percent of the population will have depth 2; 25 percent, depth 3, and so on. Within each subpopulation of a given depth, one-half of the programs are created using the grow approach, one-half using the full approach.

The fitness of each program is generally calculated for a number of cases, with the average fitness value over the cases being defined as a program's fitness. For example, if a program were being evolved to calculate y as some function of x, each program might be tested over 50 or 100 cases, each representing a value of x

in the domain. It is important to use a sufficient number of cases to represent this domain. Although it is possible to use different cases in different generations, the same fitness cases are usually used across all generations.

Fitness can be calculated in a number of ways. Koza (1992) defines four fitness metrics: raw, standardized, adjusted, and normalized. Raw fitness can be calculated in one of several ways, according to the problem being solved. For example, if the objective is to maximize the score of a game, or a profit margin, the raw fitness can be the score or the profit margin, respectively. Likewise, if the objective is to minimize costs or miles traveled, raw fitness could be the cost or number of miles, respectively. Another, more common, raw fitness metric is the sum over all cases of the absolute value of error. The error can be calculated as the sum of the linear differences between the correct values and the program values, or as the sum of the squares of the differences. For programs that output Boolean or symbolic values, the error can be calculated as the number of incorrect outputs for the test cases. Note that desirable raw fitness values can be either larger or smaller, depending on how the fitness calculation is formulated.

Standardized fitness is configured so that lower values are more desirable. In fact, the fitness value is often mathematically adjusted such that the optimum standardized fitness value is 0. In some problems, such as when cost or error values are being minimized, raw fitness and standardized fitness are identical. If raw fitness is calculated such that better values are greater, then standardized fitness is calculated by subtracting the raw fitness from the maximum possible value of raw fitness.

Adjusted fitness is calculated using standardized fitness: adjusted fitness $f_a = 1/(1 - f_s)$, where f_s is standardized fitness. Values of adjusted fitness thus range between 0 and 1, where 1 is the optimum value. Koza prefers adjusted fitness for most of his applications (Koza 1992). One reason for this is its behavior as its value approaches 1. Near the optimum, small changes in standardized fitnesses have relatively more effect on adjusted fitness than similar changes that are distant from the optimum. For example, consider a problem where standardized fitness values can vary between 0 (optimum) and 20. A change in standardized fitness from 20 to 19 only moves the adjusted fitness from 0.0476 to 0.0500, while changing standardized fitness from 3 to 2 results in an adjusted fitness increment from 0.25 to 0.33. The calculation of adjusted fitness is somewhat analogous to spacing and scaling, discussed in the Genetic Algorithm subsection on fitness calculation.

Normalized fitness is the same as the normalized fitness used in GA applications. It is the adjusted fitness value (for an individual program) divided by the sum of adjusted fitness values for all programs that make up the population. As in GAs, normalized fitness is used in roulette wheel selection.

Steps 3 and 4 of the GP process are often carried out in parallel. A probability is assigned to reproduction, and another to crossover, so that the two sum to 1. If, for example, the probability of reproduction is 10 percent (a typical value in

Koza's problems), then the probability of crossover is 90 percent. This means that once fitness calculations have been made, and it is time to build the new program population, a decision is made based on these probabilities whether to perform reproduction or crossover.

If reproduction is selected, it is often carried out in a similar fashion to the roulette wheel selection used in GAs. A candidate program is selected for reproduction with a probability proportional to its fitness divided by the sum of all of the programs' fitnesses (its normalized fitness). For very large populations of 1,000 or more, highly fit individuals are sometimes given an even greater probability of selection than their normalized fitness. This is called *overselection.*

If crossover is selected, it is accomplished by first selecting two parents using a method based on normalized fitness similar to that used for reproduction. Then, one point is randomly selected in each parent as the crossover point. The point can be anywhere in each program, including the root and internal functions, or the terminals. The entire substructure consisting of the crossover point root and everything below it is exchanged between the two programs.

Note that the parent programs, as well as the exchanged substructures, are usually of different sizes and configurations. Note also that the results of some operations may not be what is usually expected of crossover. An example is when the roots of the two programs are selected as crossover points, in which case the results are identical to the two programs being selected for reproduction into the new population.

When a crossover operation results in a program that exceeds the maximum defined depth, the program that would exceed the depth limit as a result of crossover is copied unaltered into the new population, while the crossover operation is carried out for the other program. In other words, the subtree at and below the crossover point in the unaltered program replaces the program portion at and below the crossover point in the other program.

Preprocessing and postprocessing as typically done when working with other computational intelligence tools, such as artificial neural networks and genetic algorithms, play a relatively minor role in GP implementations. The selection of the function and terminal sets significantly depends on the problem domain, however, so this selection could be thought of as preprocessing.

Formulating the approach to solving a problem with a GP implementation can be difficult. Discovering what other people have done in similar circumstances is often helpful. Chapter 26 of Koza's 1992 book presents tables to guide a user in selection of terminal sets, function sets, population size, and so on. Koza's videotapes are also useful sources of information.

Now that we've explored genetic programming, we turn to the youngest of the evolutionary computation areas, particle swarm optimization. It has a number of attributes in common with the areas discussed previously but is also different in several ways.

Particle Swarm Optimization

Particle swarm optimization (PSO) is an evolutionary computation technique developed by Kennedy and Eberhart in 1995 (Kennedy and Eberhart 1995; Eberhart and Kennedy, 1995; Eberhart, Simpson, and Dobbins 1996). Thus, at the time of the writing of this book PSO has been around for just over 10 years. Already, it is being researched and used in more than 30 countries. This section reviews developments related to PSO since its origin in 1995, along with resources available to help you learn more about it. It is written from an engineering and computer science perspective, and it is not meant to be comprehensive in areas such as the social sciences.

Following the introduction, major developments in the particle swarm algorithm since its origin in 1995 are reviewed. The original algorithm is presented first. Following are brief discussions of constriction factors, inertia weights, and tracking dynamic systems. (Applications, both those already developed and promising future application areas, are presented in Chapter 12. Those already developed include human tremor analysis, power system load stabilization, and product mix optimization.) Finally, particle swarm optimization resources are listed. Most of them can be accessed via the book's web site.

Developments

The story of particle swarm optimization is still unfolding. We can report on only the developments that have occurred as of the publication of this book. For now, let's start at the beginning. The particle swarm concept originated as a simulation of a simplified social system. The original intent was to graphically simulate the graceful but unpredictable choreography of a bird flock. Initial simulations were modified to incorporate nearest-neighbor velocity matching, eliminate ancillary variables, and incorporate multidimensional search and acceleration by distance (Eberhart and Kennedy 1995; Kennedy and Eberhart 1995). At some point in the evolution of the algorithm, it was realized that the conceptual model was, in fact, an optimizer. Through a process of trial and error, a number of parameters extraneous to optimization were eliminated from the algorithm, resulting in the very simple original implementation (Eberhart, Simpson, and Dobbins 1996).

Partical swarm optimization is similar to a genetic algorithm in that the system is initialized with a population of random solutions. It is unlike a GA, however, in that each potential solution is also assigned a randomized velocity and the potential solutions, called *particles*, are then "flown" through the problem space.

Each particle keeps track of its coordinates in the problem space that are associated with the best solution (fitness) it has achieved so far. (The fitness value is also stored.) This value is called "pbest." Another "best" value that is tracked by the global version of the particle swarm optimizer is the overall best value, and its

location, obtained so far by any particle in the population. This location is called "gbest."

The PSO concept consists of, at each time step, changing the velocity (accelerating) each particle toward its pbest and gbest locations (in the global version of PSO). Acceleration is weighted by a random term, with separate random numbers being generated for acceleration toward pbest and gbest locations.

There is also a local version of PSO in which, in addition to pbest, each particle keeps track of the best solution, called "lbest," attained within a local topological neighborhood of particles.

The (original) process for implementing the global version of PSO is as follows:

1. Initialize a population (array) of particles with random positions and velocities on d dimensions in the problem space.
2. For each particle, evaluate the desired optimization fitness function in d variables.
3. Compare each particle's fitness evaluation with its pbest. If current value is better than pbest, set the pbest value equal to the current value and the pbest location equal to the current location in d-dimensional space.
4. Compare fitness evaluation with the population's overall previous best. If the current value is better than gbest, reset gbest to the current particle's array index and value.
5. Change the velocity and position of the particle according to equations 3.3 and 3.4, respectively:

$$v_{id} = v_{id} + c_1 * \text{rand}() * (p_{id} - x_{id}) + c_2 * \text{Rand}() * (p_{gd} - x_{id}) \tag{3.3}$$

$$x_{id} = x_{id} + v_{id} \tag{3.4}$$

6. Loop to step 2 until a criterion is met, usually a sufficiently good fitness or a maximum number of iteration generations.

Note that in equation 3.4 we appear to be adding a velocity to a position. However, we are really adding a velocity occurring over a single time increment (iteration), so the equation is valid.

Particles' velocities on each dimension are clamped to a maximum velocity Vmax. If the sum of accelerations causes the velocity on that dimension to exceed Vmax, which is a parameter specified by the user, then the velocity on that dimension is limited to Vmax.

Vmax is therefore an important parameter. It determines the resolution, or fineness, with which regions between the present position and the target (best so far)

position are searched. If Vmax is too high, particles might fly past good solutions. If Vmax is too small, on the other hand, particles may not explore sufficiently beyond locally good regions. In fact, they could become trapped in local optima, unable to move far enough to reach a better position in the problem space.

The acceleration constants c_1 and c_2 in equation 3.3 represent the weighting of the stochastic acceleration terms that pull each particle toward pbest and gbest positions. Thus, adjustment of these constants changes the amount of "tension" in the system. Low values allow particles to roam far from target regions before being tugged back, while high values result in abrupt movement toward, or past, target regions.

Early experience with particle swarm optimization (trial and error mostly) led us to set each the acceleration constant c_1 and c_2 equal to 2.0 for almost all applications. Vmax was thus the only parameter we routinely adjusted, and we often set it at about 10 to 20 percent of the dynamic range of the variable on each dimension.

Based on, among other things, findings from social simulations, it was decided to design a "local" version of the particle swarm. In this version, particles have information only of their own and their neighbors' bests, rather than that of the entire group. Instead of moving toward a kind of stochastic average of pbest and gbest (the best location of the entire group), particles move toward points defined by pbest and lbest, which is the index of the particle with the best evaluation in the particle's *neighborhood.*

If the neighborhood size is defined as two, for instance, particle(i) compares its fitness value with particle($i - 1$) and particle($i + 1$). Neighbors are defined as topological neighbors; neighbors and neighborhoods do not change during a run. For the neighborhood version, the only change to the process defined in the six steps given earlier is the substitution of p_{ld}, the location of the *neighborhood best,* for p_{gd}, the *global best,* in equation 3.4. Early experience (again, mainly trial and error) led to neighborhood sizes of about 15 percent of the population being used for many applications. So, for a population of 40 particles, a neighborhood of six, or three topological neighbors on each side, was not unusual.

The population size selected is problem-dependent. Population sizes of 20 to 50 are probably most common. It was learned early on that smaller populations than were common for other evolutionary algorithms (such as GAs and evolutionary programming) were optimal for PSO in terms of minimizing the total number of evaluations (population size times the number of generations) needed to obtain a sufficient solution.

We now look at the development of the *inertia weight.* The maximum velocity, Vmax, serves as a constraint to control the global exploration ability of a particle swarm. As stated earlier, a larger Vmax facilitates global exploration, while a smaller Vmax encourages local exploitation. The concept of an inertia weight was developed to better control exploration and exploitation. The motivation was to

be able to eliminate the need for Vmax. The inclusion of an inertia weight in the particle swarm optimization algorithm was first reported in the literature in 1998 (Shi and Eberhart 1998a, 1998b).

Equations 3.5 and 3.6 describe the velocity and position update equations with an inertia weight included. It can be seen that these equations are identical to equations 3.3 and 3.4 with the addition of the inertia weight w as a multiplying factor of v_{id} in equation 3.3.

$$v_{id} = w * v_{id} + c_1 * \text{rand}() * (p_{id} - x_{id}) + c_2 * \text{Rand}( ) * (p_{gd} - x_{id}) \tag{3.5}$$

$$x_{id} = x_{id} + v_{id} \tag{3.6}$$

The use of the inertia weight w has provided improved performance in a number of applications. As originally developed, w often is decreased linearly from about 0.9 to 0.4 during a run. Suitable selection of the inertia weight provides a balance between global and local exploration and exploitation and results in fewer iterations on average to find a sufficiently optimal solution. (A different form of w, explained later, is currently being used by one of the authors, RE.)

After some experience with the inertia weight, it was found that although the maximum velocity factor, Vmax, couldn't always be eliminated, the particle swarm algorithm works well if Vmax is set to the value of the dynamic range of each variable (on each dimension). Thus, you don't need to think about how to set Vmax each time the particle swarm algorithm is used.

Another approach to using an inertia weight is to adapt it using a fuzzy system. The first paper published reporting this approach used the Rosenbrock function with asymmetric initialization as the benchmark function (Shi and Eberhart 2000). The fuzzy system comprised nine rules, with two inputs and one output. Each input and the output had three fuzzy sets defined. One input was the global best fitness for the current generation; the other was the current inertia weight. The output was the change in intertia weight. The results reported show that by using a fuzzy adaptive inertia weight, the performance of particle swarm optimization can be significantly improved in terms of the mean best fitness achieved in a given number of iterations. We discuss fuzzy systems in Chapter 7.

The next major development we consider is the constriction factor. Because particle swarm optimization originated from efforts to model social systems, a thorough mathematical foundation for the methodology was not developed at the same time as the algorithm. Within the last few years, a few attempts have been made to begin to build this foundation.

Recent work done by Clerc (1999) indicates that use of a constriction factor may be necessary to ensure convergence of the particle swarm algorithm. A detailed discussion of the constriction factor is beyond the scope of this book,

but a simplified method of incorporating it appears in equation 3.7, where K is a function of c_1 and c_2 as reflected in equation 3.8.

$$v_{id} = K^* [v_{id} + c_1 \text{ *rand( ) } *(p_{id} - x_{id}) + c_2 \text{ *Rand( ) } *(p_{gd} - x_{id})] \tag{3.7}$$

$$K = \frac{2}{\left|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}\right|}, \text{ where } \varphi = c_1 + c_2,\ \varphi > 4 \tag{3.8}$$

Typically, when Clerc's constriction method is used, φ is set to 4.1 and the constant multiplier K is thus 0.729. This results in the previous velocity being multiplied by 0.729 and each of the two $(p - x)$ terms being multiplied by 0.729 * 2.05 = 1.49445 (times a random number between 0 and 1).

In initial experiments and applications, Vmax was set to 100,000, because it was believed that Vmax isn't necessary when Clerc's constriction approach is used. However, from subsequent experiments and applications (Eberhart and Shi 2000), it has been concluded that a better approach is to limit Vmax to Xmax, the dynamic range of each variable on each dimension, while selecting w, c_1, and c_2 according to equations 3.7 and 3.8.

What we've discussed so far is fine as long as we're dealing with static systems. Most applications of evolutionary algorithms are to the solution of static problems. Many real-world systems, however, change state frequently (or continuously). These system state changes result in a requirement for frequent, sometimes almost continuous, reoptimization.

It has been demonstrated that particle swarm optimization can be successfully applied to tracking and optimizing dynamic systems (Eberhart and Shi 2001). A slight adjustment was made to the inertia weight for this purpose. The inertia weight w in equation 3.5 was set equal to [0.5 + (Rand()/2.0)]. This produces a number randomly varying between 0.5 and 1.0, with a mean of 0.75. This was selected in the spirit of Clerc's constriction factor described above, which sets w to 0.729. Constants c_1 and c_2 in equation 3.5 were set to 1.494, also according to Clerc's constriction factor.

The random component of the inertia weight is important because when tracking a dynamic system, it cannot be predicted whether exploration (a larger inertia weight) or exploitation (a smaller inertia weight) will be better at any given time. An inertia weight that varies roughly within our previous range addresses this.

For the limited testing done (Eberhart and Shi 2001) using the parabolic function, the performance of particle swarm optimization was shown to compare favorably (faster to converge, higher fitness) with other evolutionary algorithms

for all conditions tested. The ability to track a 10-dimensional function was demonstrated.

Now that we've seen how particle swarm optimization works and some of the exciting developments that have occurred recently, let's look at how to get more information about it.

Resources

Three main categories of resources are available with respect to particle swarm optimization: books, web sites, and technical papers. The first book to include a section on particle swarm optimization was Eberhart, Simpson and Dobbins (1996). See Kennedy and Eberhart (1999) for a book chapter on PSO. An entire book is now available, however, on the subject of swarms: *Swarm Intelligence* (Kennedy, Eberhart, and Shi 2001) discusses both the social and psychological as well as the engineering and computer science aspects of swarm intelligence. The web site for the book, *www.Computelligence.org*, is a guide to a variety of resources related to particle swarm optimization. Included are Java applets that can be run online illustrating the optimization of a variety of benchmark functions. The user can select a variety of parameters. Also on the web site is PSO software written in C++, Visual BASIC, and Java that can be downloaded. A variety of links to other web sites are also provided. The web site for this book is, obviously, another major source of PSO information and pointers to other sites. With respect to conferences, those related to evolutionary computation (such as the Congress on Evolutionary Computation) sponsored or cosponsored by the IEEE provide the richest source of publications on PSO. A special issue of the *IEEE Transactions on Evolutionary Computation* devoted to particle swarm optimization was published in June 2004.

Summary

In this chapter, we first present a brief history of evolutionary computation, followed by an overview of the evolutionary computation field. Five main evolutionary algorithms are then discussed in detail in their own sections, respectively. The five areas are genetic algorithms, evolutionary programming, evolution strategies, genetic programming, and particle swarm optimization. Among the five, the genetic algorithm is emphasized, with more detailed discussion on subjects such as schemata and the schema theorem.

The five evolutionary algorithms share many features. First, all are population-based search algorithms. The cooperation and/or competition among the population move the potential solutions toward the better search areas. Second, all are motivated by nature. Particle swarm optimization is motivated by social behavior, and the other four main evolutionary algorithms are motivated by the

survival of the fittest and/or evolution. Third, the five evolutionary algorithms employ direct "fitness" information instead of function derivatives or other related knowledge. Therefore, evolutionary algorithms can solve problems that are not continuous, not differentiable, and multimodal. Fourth, randomness plays roles in all of the algorithms. The search process is not deterministic. It is this randomness and the "fitness" information that gives evolutionary algorithms the ability to enable individuals to move to anywhere and escape from local optima.

Finally, they all generate the next generation from the previous generation. In particle swarm optimization, the individuals (particles) "fly" through the search space with dynamically changing velocities. That is, the individuals "fly" to the next generation from the current generation. In the other four evolutionary algorithms, the next generation is obtained by applying so-called evolution operators to the current generation: In genetic algorithms and evolution strategies, the selection, mutation, and crossover (recombination) operators are applied; in genetic programming, selection and crossover operators are used; and in evolutionary programming, selection and mutation operators are utilized.

Comparisons of evolutionary computation tools (in these five areas) and other processing methods are also discussed in each section, respectively. Evolutionary algorithms are recommended to solve nonlinear problems for which the traditional approaches are hard, if not impossible, to apply. It is usual and reasonable to expect evolutionary algorithms to find near optimal solutions within a limited period of time—a solution that is good enough to be acceptable.

Exercises

1. Convert the following binary coded strings to Gray coding: 10101010, 10011100, 01100110.
2. How many schemata are possible for a 6-bit binary string?
3. According to the schema theorem, what happens to highly fit schemata in successive generations? What are the effects of crossover and mutation according to the theorem? Why use crossover and mutation?
4. Assume standard binary encoding of parameters is used for a genetic algorithm implementation. Briefly discuss how the effects of uniform crossover and two-point crossover change as the number of bits representing a parameter is increased.
5. After running a genetic algorithm for a fairly long time, the fitness values tend to cluster at the high end of the scale. For example, on a scale of 0 to 1, they might cluster from 0.90 to 0.98. What is the main problem with this? How can it be alleviated?

6. Assume that the average fitness of strings containing a particular schema S is 20 percent less than the average fitness of all schemata, and the schema appears in 50 percent of the initial population. Assume that the probability of disruption of this schema by crossover or mutation is negligible. Calculate when S will disappear from a population with 50 members. Repeat for a 100-member population.
7. Assume each population member in a GA consists of 8 binary coded bits (as in the GA example in the chapter), representing the integers 0 to 255. Briefly describe or sketch the portion of the problem space covered by the following schemata: 0*******, *******1, 10******, ******10, ***11***.
8. What is the main difference between evolutionary programming and evolution strategies?
9. Assume you are going to use genetic programming to evolve a program to classify the Iris dataset (pp. 197–198). Specify a function set and a terminal set that are appropriate to solve the problem.
10. Sketch out a genetic programming representation of the best possible *approximate* solution to $v = \pi r^2 h$, (v is the volume of a right cylinder, r is its radius, and h is its height) given that the maximum depth of the program is five layers and you may only use the constant values 0, 1, and 10. If you were going to evolve programs to do this calculation using genetic programming, what would you propose to use as a function set?
11. How is a particle swarm optimizer similar to a genetic algorithm? How is it different? How does it resemble an evolution strategies implementation?