Genetic Algorithms: Principles of Natural Selection Applied to Computation

Stephanie Forrest

A genetic algorithm is a form of evolution that occurs on a computer. Genetic algorithms are a search method that can be used for both solving problems and modeling evolutionary systems. With various mapping techniques and an appropriate measure of fitness, a genetic algorithm can be tailored to evolve a solution for many types of problems, including optimization of a function or determination of the proper order of a sequence. Mathematical analysis has begun to explain how genetic algorithms work and how best to use them. Recently, genetic algorithms have been used to model several natural evolutionary systems, including immune systems.

Evolution by natural selection is one of the most compelling themes of modern science, and it has revolutionized the way we think about biological systems. There is a form of evolution, called a genetic algorithm, that takes place in a computer. In genetic algorithms, selection operates on strings of binary digits stored in the computer's memory, and over time, the functionality of these strings evolves in much the same way that natural populations of individuals evolve. Although the computational setting is highly simplified compared with the natural world, genetic algorithms are capable of evolving surprisingly complex and interesting structures. These structures, called individuals, may represent solutions to problems, strategies for playing games, visual images, or simple computer programs.

The Darwinian theory of evolution depicts biological systems as the product of the ongoing process of natural selection. Likewise, genetic algorithms allow engineers to use a computer to evolve solutions over time, instead of designing them by hand. Because almost any method, theory, or technique can be encoded on a computer, this implies an approach to problem-solving that can be, at least partially, automated by a computer. More specifically, computer science has long been interested in how the design, development, and debugging of computer programs could be automated, and genetic algorithms provide one avenue toward this goal.

The very name genetic algorithm is puzzling because the first word refers to a biological science and the second word is borrowed from computer science. An algorithm is a step-by-step procedure for accomplishing some specific task—sorting numbers, formatting text on a page, or diagnosing car problems. Many algorithms can be readily implemented as computer programs. Thus, an algorithm is the general description of a procedure, and a program is its realization as a sequence of instructions to a computer. Genetic algorithms are loosely based on ideas from population genetics; they feature populations of genotypes (an individual's genetic material) stored in memory, differential reproduction of these genotypes, and variations that are created by processes analogous to the biological processes of mutation and crossover.

Although genetic algorithms are known primarily as a problem-solving method, they can also be used to study evolution itself and to model dynamic systems. Many systems that evolve over time can be modeled with a genetic algorithm, including biological systems (such as ecologies, immune systems, and genetic systems) and social systems (such as economies and political systems). This second, and quite different, use of genetic algorithms suggests a computational view of evolution in which the mechanisms of natural selection, inheritance, and variation serve primarily to transmit and process information. One interesting question is whether data produced by a highly idealized model such as a genetic algorithm can provide useful insights about natural evolutionary systems.

Genetic Algorithm Overview

The basic idea of a genetic algorithm is very simple. First, a population of individuals is created in a computer (typically stored as binary strings in the computer's memory), and then the population is evolved with use of the principles of variation, selection, and inheritance. There are many ways of implementing this simple idea, and I will describe the one invented by Holland (*1*, *2*) (Fig. 1).

The idea of using selection and variation to evolve solutions to problems goes back at least to Box (*3*), although this work did not make use of a computer. In the late 1950s and early 1960s, there were several independent efforts to incorporate ideas from evolution in computation. Of these, the best known are those by Holland (*4*), Fogel *et al.* (*5*), and Rechenberg (*6*, *7*). Rechenberg emphasized the importance of selection and mutation as mechanisms for solving difficult real-valued optimization problems. Fogel *et al.* developed similar ideas for evolving intelligent agents in the form of finite state machines. Holland emphasized the adaptive properties of entire populations and the importance of recombination mechanisms such as crossover.

In its simplest form, each individual in the population consists of a string of binary digits (bits). Genetic algorithms often use more complex representations, including diploid (*8*, *9*) and multiple chromosomes (*9*) and higher cardinality alphabets (*10*–*12*). However, the binary case is both the simplest and the most general. By analogy with biological systems, the string of bits is referred to as the "genotype." Each individual consists only of its genetic material, which is organized into one (haploid) chromosome. Each bit position (set to 1 or 0) represents one gene. The term "bit string" refers to both genotypes and the individuals that they define. There are a variety of techniques for mapping bit strings to different problem domains.

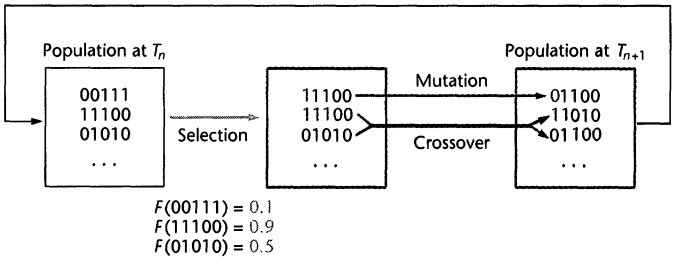

Fig. 1. Operation of the genetic algorithm. A population of three individuals is shown. Each is assigned a fitness value by the function *F*. On the basis of these fitnesses, the selection phase assigns the first individual (00111) zero copies, the second (111000) two, and the third (01010) one copy. After selection, the genetic operators are applied probabilistically; the first individual has its first bit mutated from a 1 to a 0, and crossover combines the second two individuals into two new ones. The resulting population is shown in the box labeled T_{n+1}.

The author is in the Department of Computer Science, University of New Mexico, Albuquerque, NM 87131–1386 (forrest@cs.unm.edu).

The initial population of individuals is usually generated randomly, although it need not be. Each individual is tested empirically in an "environment" and is assigned a numerical evaluation of its merit by a fitness function F. The environment can be almost anything—another computer simulation, interactions with other individuals in the population, actions in the physical world (by a robot, for example), or a human's subjective judgment (*13, 14*). The fitness function returns a single number (usually, higher numbers are assigned to fitter individuals). This constraint is sometimes relaxed so that the fitness function returns a vector of numbers (*15*). The fitness function determines how each gene (bit) of an individual will be interpreted and, thus, what specific problem the population will evolve to solve. The fitness function is the primary place in which the traditional genetic algorithm is tailored to a specific problem.

Once all individuals in the population have been evaluated, their fitnesses are used as the basis for selection. Selection is implemented by eliminating low-fitness individuals from the population, and inheritance is implemented by making multiple copies of high-fitness individuals. Genetic operators such as mutation (flipping individual bits) and crossover (exchanging substrings of two individuals to obtain two offspring) are applied probabilistically to the selected individuals to produce a new population (or generation) of individuals. The term "crossover" is used here to refer to the exchange of homologous substrings between individuals, although the biological term "crossing over" generally implies exchange within an individual. New generations can be produced either synchronously, so that the old generation is completely replaced, or asynchronously, so that generations overlap.

By transforming the previous set of good individuals to a new one, the operators generate a new set of individuals that have a better than average chance of also being good. When this cycle of evaluation, selection, and genetic operations is iterated for many generations, the overall fitness of the population generally improves, and the individuals in the population represent improved "solutions" to whatever problem was posed in the fitness function.

There are many details left unspecified by this description. For example, selection can be performed in any of several ways: It could arbitrarily eliminate the least fit 50% of the population and make one copy of all the remaining individuals, it could replicate individuals in direct proportion to their fitness, or it could scale the fitnesses in any of several ways and replicate individuals in direct proportion to their scaled values (a more typical method). Likewise, the crossover operator can pass on both offspring to the new generation, or it can arbitrarily choose one to be passed on; the number of crossovers can be restricted to one per pair, two per pair, or N per pair. These and other variations of the basic algorithm have been discussed extensively (*2, 16–20*).

This computational definition of evolution has the advantage that mathematics and simulation can both be used to ask various questions: Does the population evolve a stable set of individuals? Does it always evolve the most fit individuals possible? If so, why? If not, why not? Does the population ever converge on one genotype? Does it always converge? How long does it take to converge? How do various parameters (such as population size, crossover rates, mutation rates, and so on) affect the rate of evolution? How do various changes to the details of the algorithm affect the rate of evolution? Research in genetic algorithms seeks to answer these questions, both mathematically and through simulation.

Genetic Algorithms for Solving Problems

The simple computation procedure described above can be applied in many different ways to solve a wide range of problems. In the design of a genetic algorithm to solve a specific problem, there are always two major decisions: (i) specifying the mapping between binary strings and candidate solutions (commonly referred to as the representation problem), and (ii) defining a concrete measure of fitness. In some cases, the best representation and fitness function are obvious, but in many cases, they are not, and in all cases, the particular representation and fitness function that are selected will determine the ultimate success of the genetic algorithm on the chosen problem.

Possibly the simplest representation is a feature list in which each bit, or gene, represents the presence or absence of a single feature. This representation is useful for learning pattern classes defined by a critical set of features. For example, in an operations research problem, Packard used a genetic algorithm to search for correlations between certain office decision procedures and the efficiency of that office (*21*). He described 19 different decision procedures (the independent variables) associated with different offices of an organization and defined a quantitative measure for the output (efficiency) of each office (the dependent variable). The genetic algorithm was then used to search for combinations of procedures that correlated strongly with either high-efficiency or low-efficiency offices. The feature list approach to this problem assigns one bit to represent the presence or absence of each different decision procedure, and fitness is assigned to those individuals whose feature settings correspond to high- (or low-) efficiency offices (Packard's system used a more complicated representation than that described here, but the principle was the same).

Of the many other representations that have been investigated, three will be discussed in detail: numerical encodings for function optimization, random-key encodings for ordering problems, and computer programs for automated programming. The first is by far the most frequently used in real applications. It is well understood and noncontroversial. The second two representations illustrate the possibilities of genetic algorithms, but they were developed more recently and are still largely unproven. For an overview of other representation techniques, see (*22*).

Function optimization. Perhaps the most obvious application of genetic algorithms, pioneered by DeJong (*23*), is multiparameter function optimization. Many problems can be formulated as a search for some optimal value, where the value is a complicated function of its input parameters. In some cases, the parameter settings that lead to the greatest (or least) value of the function are of interest. In other cases, the exact optimum is not required, just a near optimum, or even a value that represents a slight improvement over the previously best known value.

As a simple example, consider the function $f(x, y) = yx^2 - x^4$. This function is solvable analytically, but if it were not, a genetic algorithm could be used to find the values x and y that produce the maximum $f(x, y)$ in a particular region of $\Re^2$. The most straightforward method of representation (Fig. 2) is to assign regions of the bit string to represent each parameter (vari-

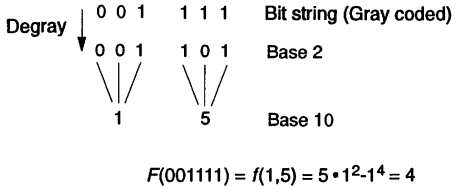

Fig. 2. Bit-string encoding of multiple real-valued parameters. An arbitrary string of six bits is interpreted in the following steps: (i) segment the string into two regions with the first three bits reserved for x and the second three bits for y; (ii) interpret each three-bit substring as a Gray code and map back to the corresponding binary code (see Table 1); (iii) map each three-bit substring from its binary code to its decimal equivalent; (iv) substitute the two decimal values for x and y in the fitness function F; and (v) return $F(x, y)$ as the fitness of the original string.

able). Once the order in which the parameters are to appear is determined (in the figure, x appears first and y appears second), the next step is to specify the domain for x and y (that is, the range of values for x and y that are candidate solutions). Also, because x and y can be real-valued in this example, the parameters will be discretized. The precision of the solution is determined by how many bits are used to represent each parameter. In the example, 3 bits are assigned for x and 3 for y, although 10 is a more typical number. For simplicity, x and y will vary between 0 and 7.

With this representation, the genetic algorithm generates a random population of bit strings, decodes each bit string into the corresponding decimal values for x and y, applies the fitness function [$f(x, y) = yx^2 - x^4$] to the decoded values, selects the most fit individuals [those with the highest $f(x, y)$] for copying and variation, and then repeats the process. The population will eventually converge on a set of bit strings that represents an optimal or near optimal solution. However, there will always be some variation in the population because of mutation.

There are different ways of mapping between bits and decimal numbers, so an encoding must also be chosen. The traditional binary encoding has the drawback that in some cases all the bits must be changed in order to increase a number by 1. For example, the bit pattern 011 translates to 3 in decimal, but 4 is represented by 100 (see Table 1). This can make it difficult for an individual that is close to an optimum to move even closer by mutation. Also, mutations in high-order bits (the leftmost bits) are more significant than mutations in low-order bits. This can violate the idea that bit strings in successive generations will have a better than average chance of having high fitness because mutations may often be disruptive. A different encoding, called Gray coding, addresses the first of these problems. Gray codes have the property that incrementing or decrementing any number by 1 is always a one-bit change (Table 1). Gray codes can be defined for numbers of any size, although the table shows only values for three-bit numbers. In practice, Gray-coded representations are often more successful for multiparameter function optimization applications.

Table 1. Gray codes for three-bit numbers.

Decimal	Binary code	Gray code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

There are a number of other representation tricks that are often used for function optimization, including logarithmic scaling (interpreting bit strings as the logarithm of the true parameter value), dynamic encoding (*24, 25*) (a technique that allows the number and interpretation of bits allocated to a particular parameter to vary throughout a run), variable-length representations (*26, 27*), delta coding (*28*) (the bit strings express a distance away from some previous partial solution), and a multitude of nonbinary encodings (*29, 30*).

Although a function of two variables was used as an example, the strength of the genetic algorithm lies in its ability to manipulate many parameters. This method has been used for hundreds of applications (*2, 16, 19, 20, 31*), including aircraft design, the tuning of parameters for algorithms that detect and track multiple signals in an image, and the location of regions of stability in systems of nonlinear difference equations.

Ordering problems. Another common problem involves finding an optimal ordering for a sequence of N items. Examples include a tour of cities that minimizes the distance travelled (the Travelling Salesman problem), packing boxes into a bin to minimize wasted space (the bin-packing problem), graph-coloring problems, and DNA fragment assembly. The computational complexity of these problems is thought to increase exponentially as N increases.

For example, in the Travelling Salesman problem, suppose there are four cities—1, 2, 3, and 4—each with a unique bit-string encoding. A natural way to represent tours would be to list the permutations, so that 3 2 1 4 would be one candidate tour and 4 1 2 3 would be another. This representation is problematic for the genetic algorithm because crossovers between these two candidates do not necessarily produce legal tours. For example, a cross between positions two and three in the example produces the individuals 3 2 2 3 and 4 1 1 4, both of which are illegal tours because not all of the cities are visited and some are visited more than once.

Two general methods have been proposed to address this representation problem: (i) designing specialized crossover operators that produce only legal tours and (ii) adopting a different representation. Of these, the use of specialized operators has been the prevalent method for successful applications of genetic algorithms to ordering problems such as the Travelling Salesman problem [for example, see (*32*)]. Following the second approach, a number of representations have been proposed (*29, 33–35*), including the random-key method (*36–38*). Because specialized crossover operators tend to be problem-specific, I will discuss the random-key method as an example of a general representation method for ordering problems. This method has been applied with limited success to the Travelling Salesman problem and with more success to scheduling, routing, resource allocation, and assignment problems (*38*). Although it is not well justified theoretically and has not been widely adopted within the genetic algorithm community, it is an imaginative encoding and illustrates the wide range of representations that are possible.

The random-key method divides the bit string into N segments of k bits, where N is the number of cities in the tour and $2^k >> N$. Because the number of bits used for each segment can encode many more numbers than there are cities, the binary code for each segment can be interpreted as a random number. For example, if three bits are assigned to each segment then any bit string can be decoded into a sequence of integers between zero and seven (ties are resolved randomly). A randomly generated bit string might yield the following sequence: 5 3 1 7. Now, these keys are decoded to a tour by identifying the position of the smallest element. The smallest element is 1 and it is in the third position, so city 3 becomes the first city on the tour. This method produces the tour 3 2 1 4 for the example string (Fig. 3).

The random-key encoding has the advantage that any bit string represents a legal tour, which eliminates the need for specialized crossover operators. However, domain-independent representations such as this are not always successful on hard combina-

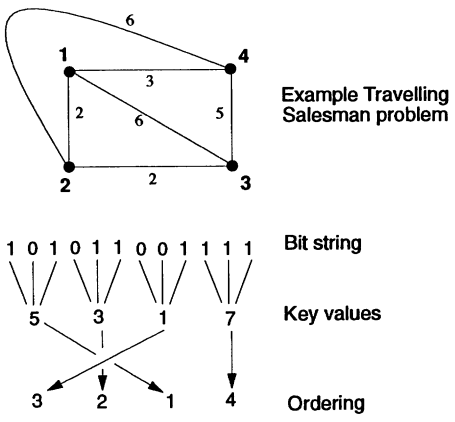

Fig. 3. The random-key representation for ordering problems. The example shows four cities (represented as nodes in the graph). Labels on the arcs denote the distance between cities.

torial problems, such as the Travelling Salesman problem. When combined with domain-specific knowledge, the algorithms can be quite effective on problems of this class (*34, 39*).

Automatic programming. Genetic algorithms have recently been used to evolve a special kind of computer program (*12*). These programs are written in a subset of the programming language Lisp. Lisp programs can naturally be represented as trees (Fig. 4). Populations of random program trees are generated and evaluated as in the standard genetic algorithm. All other details are similar to those described for binary genetic algorithms with the exception of crossover. Instead of exchanging substrings, genetic programs exchange subtrees between individual program trees. This modified form of crossover appears to have many of the same advantages as traditional crossover (such as preserving partial solutions).

Genetic programming has the potential to be extremely powerful because Lisp is a general-purpose programming language and genetic programming eliminates the need to devise a chromosomal representation. In practice, however, genetic programs are built from subsets of Lisp tailored to particular problem domains, and at this point considerable skill is required to select just the right subset for a particular problem. Although the method has been tested on a wide variety of problems, it has not been used extensively in real applications.

The genetic programming method is particularly intriguing because its solutions are so different from human-designed programs for the same problem. Humans try to design elegant and general computer programs, whereas genetic programs are often needlessly complicated, not revealing the underlying algorithm. For example, a human-designed program for computing cos $2x$ might be $1 - 2\sin^2 x$, expressed in Lisp as

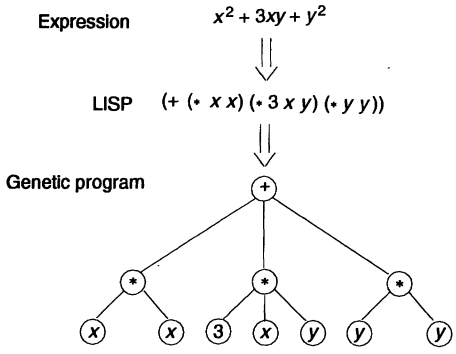

Fig. 4. Tree representation of computer programs. The displayed tree corresponds to the expression $x^2 + 3xy + y^2$. Operators for each expression are displayed as a root, and the operands for each expression are displayed as children.

(−1 (* 2 (* sin (sin x)))), but genetic programming discovered (*9*, p. 241)

```
(sin (− (− 2 (* x 2))
  (sin (sin (sin (sin (sin (sin (* (sin (sin 1))
                (sin (sin 1)))))))))))
```

For anyone who has studied computer programming, this is apparently a major drawback because the evolved programs are inelegant, redundant, inefficient, difficult for a human to read, and do not reveal the underlying structure of the algorithm. However, genetic programs do resemble the kinds of ad hoc solutions that evolve in nature through gene duplication, mutation, and modifying structures from one purpose to another. There is some evidence that the "junk" components of a genetic program sometimes turn out to be useful components in other contexts. Thus, if the genetic programming endeavor is successful, it could revolutionize software design.

Mathematical Analysis of Genetic Algorithms

Although there are many problems for which the genetic algorithm can evolve a good solution in reasonable time, there are also problems for which it is inappropriate (such as problems in which it is important to find the exact global optimum). It would be useful to have a mathematical characterization of how the genetic algorithm works that is predictive. Research on this aspect of genetic algorithms has not produced definitive answers. The domains for which one is likely to choose an adaptive method such as the genetic algorithm are precisely those about which we typically have little analytical knowledge; they are complex, noisy, or dynamic (changing over time). These characteristics make it virtually impossible to predict with certainty how well a particular algorithm will perform on a particular problem, especially if the algorithm is nondeterministic, as is the case with the genetic algorithm. In spite of this difficulty, there are fairly extensive theories about how and why genetic algorithms work in idealized settings.

Analysis of genetic algorithms begins with the concept of a search space. The genetic algorithm can be viewed as a procedure for searching the space of all possible binary strings of a fixed length l (denoted as $\{0, 1\}^l$). Under this interpretation, the algorithm is searching for points in the l-dimensional space $\{0, 1\}^l$ that have high fitness. The search space is identical for all problems of the same size (same l), but the locations of good points will generally differ. The surface defined by the fitness of each point is sometimes referred to as the fitness landscape. The longer the bit strings, corresponding to higher values of l, the larger the search space is, growing exponentially with the length of l. For problems with a sufficiently large l, it is not feasible for any algorithm to examine more than a small fraction of the search space. For example, $l = 64$ defines a search space that is too large to search exhaustively with current computer technology. Because only a small fraction of a search space this size can be examined, it is unreasonable to expect an algorithm to locate the global optimum in the space. A more reasonable goal is to search for good regions of the search space corresponding to regularities in the problem domain. Holland (*1*) introduced the notion of a "schema" to explain how genetic algorithms search for regions of high fitness. Schemas are theoretical constructs used to explain the behavior of genetic algorithms and are not processed directly by the algorithm. The following description of schema processing is excerpted from (*40*).

A schema is a template, defined over the alphabet {0, 1, *}, that describes a pattern of bit strings in the search space $\{0, 1\}^l$ (the set of strings of length l). For each of the l bit positions, the template either specifies the value (allele) at that position (the allele is 1 or 0) or indicates by the symbol * (referred to as "don't care") that either value is allowed.

For example, the two strings A and B have several bits in common. We can use schemas to describe the patterns these two strings share.

```
A = 100111
B = 010011
    **0*11
    ****11
    **0***
    **0**1
```

A bit string x that matches the pattern of a schema s is said to be an instance of s; for example, A and B are both instances of the schemas shown above. In schemas, a 1 or 0 is referred to as a defined bit; the order of a schema is the number of defined bits in that schema; and the defining length of a schema is the distance between the leftmost and rightmost defined bits in the string (for example, the defining length of **0**1 is 3).

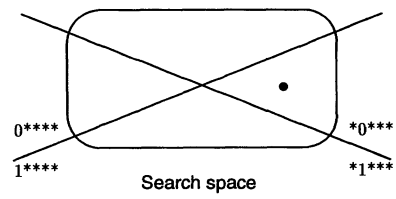

Fig. 5. Schemas define hyperplanes in the search space.

Schemas define hyperplanes in the search space $\{0, 1\}^l$. Figure 5 shows four hyperplanes, corresponding to the schemas 0****, 1****, *0***, and *1***. Any point in the space is simultaneously an instance of two of these schemas. For example, the point shown in Fig. 5 is an instance of both 1**** and *0*** (and also of 10***).

The fitness of any bit string in the population gives some information about the average fitness of the 2^l different schemas of which it is an instance, so an explicit evaluation of a population of M individual strings is also an implicit evaluation of a much larger number of schemas. This is referred to as implicit parallelism. At the explicit level, the genetic algorithm searches through populations of bit strings, but the genetic algorithm's search can also be interpreted as an implicit schema sampling process. Feedback from the fitness function, combined with selection and recombination, biases the sampling procedure over time away from those schemas that give negative feedback (low average fitness) and toward those that give positive feedback (high average fitness). Ultimately, the search procedure should identify regularities, or patterns, in the environment that lead to high fitness, and because the space of possible patterns is larger than the space of possible individuals (3^l versus 2^l), implicit parallelism is potentially advantageous.

An important theoretical result about genetic algorithms is the Schema Theorem (*1*, *2*), which states that the observed best schemas are expected to receive an exponentially increasing number of samples in successive generations. Figure 6 illustrates the rapid convergence on fit schemas by the genetic algorithm. This strong convergence property of the genetic algorithm is a two-edged sword. On the one hand, the fact that the genetic algorithm can close in on a fit part of the space very quickly is a powerful property; on the other hand, because the genetic algorithm always operates on finite size populations, there is inherently some sampling error in the search, and in some cases the genetic algorithm can magnify a small sampling error, causing premature convergence on local optima (*2*).

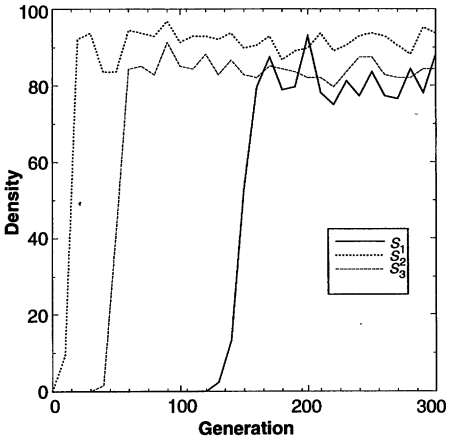

Fig. 6. Schema frequencies in the population over time for three schemas

s_1 = 1111111111111111**********
****
s_2 = **********1111111111111111
****
s_3 = **********
11111111111111111111111111111111

(each schema is 64 bits long and is displayed over two lines). The function plotted was a "royal road" function (*63*), for which the optimum value is the string of all 1's.

According to the building blocks hypothesis (*1*, *2*), the genetic algorithm initially detects biases towards higher fitness in some low-order schemas (those with a small number of defined bits) and converges on this part of the search space. Over time, it detects biases in higher order schemas by combining information from low-order schemas by means of crossover and eventually converges on a small region of the search space that has high fitness. The building blocks hypothesis states that this process is the source of the genetic algorithm's power as a search and optimization method. If this hypothesis about how genetic algorithms work is true, then crossover is of primary importance, and it distinguishes genetic algorithms from other similar methods, such as simulated annealing and greedy algorithms. A number of authors have questioned the adequacy of the building blocks hypothesis as an explanation for how genetic algorithms work (*41*, *42*), and there are several active research efforts studying schema processing in genetic algorithms. Nevertheless, the explanation of schemas and recombination presented here stands as the most common account of why genetic algorithms perform as they do.

The building blocks hypothesis suggests an analogy between the way genetic algorithms work and Fourier analysis, in which an arbitrary curve can be approximated by the sum of sines and cosines of progressively higher frequencies. Walsh functions are a complete orthogonal set of basis functions that provide a representation similar to Fourier transforms (*43*), and Bethke applied Walsh functions to the study of schema processing in genetic algorithms (*44*). He developed the Walsh-Schema transform, in which discrete versions of Walsh functions are used to calculate schema average fitnesses efficiently. He then used this transform to characterize functions as easy or hard for the genetic algorithm to optimize. Bethke's work was further developed and explicated by Goldberg (*45*), Tanese (*46*), Holland (*47*), and Forrest and Mitchell (*40*).

As an example of the relevance of schemas to function optimization, consider the function shown in Fig. 7. The function is defined over the integers on the interval [0, 31] (here, $l = 5$), so the x axis represents the bit string argument (input to the function), and the y axis shows the function's value, or fitness. In this example, the x value will always be between 0 and 31. For example, the string 10000 would be interpreted as 16, and 01000 would be interpreted as 8. Likewise, the schema 0**** (dotted line) includes all points less than 16, the schema 1**** (also indicated with a dotted line) includes all points greater than or equal to 16, and the schema *0*** (dashed lines) specifies the integers on the intervals [0, 7] and [16, 23]. With this example, it is easy to see how an individual that was an instance of the schema 0**** could be combined through crossover with an instance of the schema *0*** to yield an instance of 00***, which corresponds to the most fit region of the space (shaded region). That is, 0**** and *0*** are partial solutions.

Schema analysis can be used to predict which fitness landscapes are well suited for genetic algorithms and which are not. Goldberg expanded the work of Bethke by introducing the term "deception" and characterizing genetic algorithm difficulty in terms of deception. In deceptive functions, low-order schemas lead the genetic algorithm "away" from good high-order schemas. For example, the following might be the most fit order 1 schemas: 0*****, *0****, **0***, and so on; but the point 111111 might turn out to be the global optimum. The concept of deception and its

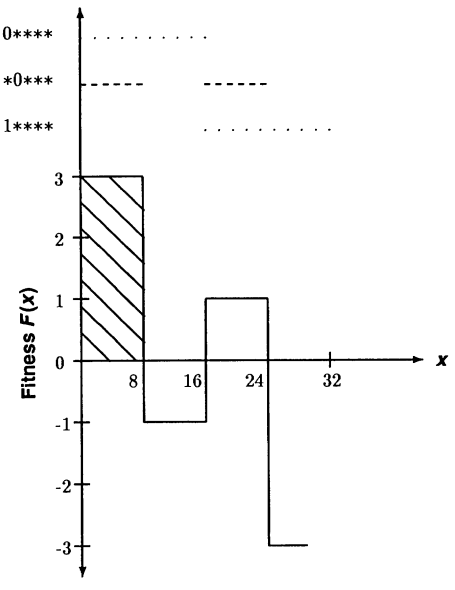

Fig. 7. The example function is indicated by solid line. The dashed and dotted lines indicate the range of the noted schemas. The hatched region is the most fit region of the space.

implications for the performance of genetic algorithms have been a major area of research in recent years.

There are several other approaches to the mathematical analysis of the behavior of genetic algorithms: models developed for population genetics (48, 49), algebraic models (50), signal-to-noise analysis (51), landscape analysis, (52), and methods based on Probably Approximately Correct (PAC) learning (53).

Genetic Algorithms for Making Models

Genetic algorithms have been used as models of a wide variety of dynamic processes, including induction in psychology (54), natural evolution in ecosystems (55), evolution in immune systems (56), and imitation in social systems (57, 58). Making computer models of evolution is somewhat different from many conventional models because the models are highly abstract. The data produced by these models are unlikely to make exact numerical predictions. Rather, they can reveal the conditions under which certain qualitative behaviors are likely to rise: diversity of phenotytes in resource-rich (or -poor) environments, cooperation in competitive nonzero-sum games, and so forth. Thus, the models described here are being used to discover qualitative patterns of behavior, and in some cases, critical parameters in which small changes have drastic effects on the outcomes. Such modeling is common in nonlinear dynamics and in artificial intelligence but is much less accepted in other disciplines. Both of the following examples, ecological modeling and immune systems, represent exploratory research projects that are currently under active investigation but have not as yet produced concrete results. For examples of more mature modeling projects, see (54, 57, 59).

Modeling ecological systems. The Echo system (55) shows how genetic algorithms can be used to model ecologies. The major differences between Echo and standard genetic algorithms are (i) there is no explicit fitness function, (ii) individuals have local storage (they consist of more than their genome); and (iii) the genetic representation is based on a higher cardinality alphabet than binary strings. In Echo, fitness evaluation takes place implicitly; that is, individuals in the population (called agents) are allowed to make copies of themselves anytime they acquire enough "resources" to replicate their genome. Different resources are modeled by different letters of the alphabet (say A, B, C, D), and genomes are constructed out of those same letters. However, these resources can exist independently of the agent's genome, either free in the environment or stored internally by the agent. Agents acquire resources by interacting with other agents through trade and combat. Echo thus relaxes the constraint that an explicit fitness function must return a numerical evaluation of each agent. This "endogenous" fitness function is much closer to the way fitness is assessed in natural settings. In addition to trade and combat, a third form of interaction between agents is "mating." Mating provides opportunities for agents to exchange genetic material through crossover, thus creating hybrids. Mating, together with mutation, provides the mechanism by which new types of agents evolve.

Populations in Echo exist on a two-dimensional grid of sites. Many agents can "cohabit" one site and agents can migrate between sites. Each site is the source of certain renewable resources. On each time step of the simulation, a fixed amount of resources at a site becomes available to the agents located at that site. Different sites may produce different amounts of different resources. For example, one site might produce ten A's and five B's each time step, and its neighbor might produce five A's, zero B's, and five C's. The idea is that an agent will do well (reproduce often) if it is located at a site whose renewable resources match well with its genomic makeup.

In preliminary simulations, the Echo system has demonstrated surprisingly complex behaviors (including something resembling a biological arms race in which two competing species develop progressively more complex offensive and defensive combat strategies), ecological dependencies among different species, and sensitivity (in terms of the number of different phenotypes) to differing levels of renewable resources. Although the Echo system is largely untested, it does show how the fundamental ideas of genetic algorithms can be incorporated into a system that captures important features of natural ecological systems.

Immune systems. In another recent project, the genetic algorithm is used to model certain aspects of the immune system, specifically, clonal selection and the evolution of the antibody V-region gene libraries (56, 60, 61). The models are based on an abstract universe of binary strings in which interactions among strings represent molecular binding (62). The binding affinity between real antigens and real antibodies is primarily determined by molecular shape and electrostatic surface charge, both of which are complementary when the molecules have high affinity. In the artificial model, binding takes place when an antibody bit string and an antigen bit string have complementary binary patterns. Binding between these idealized antibodies and antigens is defined by a matching function that rewards more specific matches over less specific ones; this constraint is related to the immune system's ability to distinguish self from other because recognition of other must be fairly specific in order to avoid recognizing self.

One population of antibodies and one of antigens are constructed, each from bit strings. Antigens are "presented" to the antibody population one at a time, and high-affinity antibodies have their fitness increased. The antibody population is then evolved by the genetic algorithm on the basis of its success at matching antigens.

This model has been used to study both the ability of the genetic algorithm to detect common patterns (schemas) in a noisy environment and its ability to maintain diversity within its population (56). Both of these capabilities are important because natural immune systems are able to recognize an enormous number of foreign molecules with limited resources.

In one set of experiments, the model evolved an antibody type, represented as a population of similar antibodies, that matched multiple antigens through the identification of a common schema. This problem is analogous to the problem the immune system faces in identifying bacteria that, although different in detail, may use a similar polysaccharide in the construction of their cell walls. In a second set of experiments, we studied the model's ability to maintain coverage of the space of antigens while under the selective pressure of the genetic algorithm, as required by clonal selection. By matching an antigen with multiple antibodies and then giving the fitness score to the best matching antibody, the algorithm allowed a stable population to evolve that contained representatives of different antibodies. In a third set of experiments, investigators used bit strings to represent the genetic encoding of V-region libraries and studied the evolution of these libraries under the genetic algorithm (61). The experiments showed that the model evolves a set of highly dissimilar library entries, even when started with completely homogeneous entries and when a very small fraction of the repertoire of possible antibodies is expressed at any one time. These preliminary results suggest that the genetic algorithm can be used to model evolution in the immune system.

Future Prospects

The idea of using evolution to solve difficult problems and to model natural phenomena is promising. The genetic algorithms described here are the first steps in this direction. Necessarily, they have abstracted out much of the richness of biology, and in the

future, we can expect a wide variety of evolutionary systems based on the principles of genetic algorithms but less closely tied to these specific mechanisms. For example, more elaborate representation techniques, including those that use complex genotype-to-phenotype mappings, and increased use of nonbinary alphabets, can be expected. Endogenous fitness functions, similar to the one described for Echo, may become more common, as well as dynamic and coevolutionary fitness functions. More generally, biological mechanisms of all kinds are being incorporated into computational systems, including viruses, parasites, and immune systems.

From an algorithmic perspective, genetic algorithms join a broader class of stochastic methods for solving problems. An important area of future research is to understand carefully how these algorithms relate to one another and which algorithms are best for which problems.

REFERENCES AND NOTES

1. J. H. Holland, *Adaptation in Natural and Artificial Systems* (Univ. of Michigan Press, Ann Arbor, MI, 1975; reprinted by MIT Press, Cambridge, MA, 1992).
2. D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning* (Addison-Wesley, Reading, MA, 1989).
3. G. E. P. Box, *J. R. Stat. Soc. Ser. C* **6**, 81 (1957).
4. J. H. Holland, *J. Assoc. Comput. Mach.* **3**, 297 (1962).
5. L. J. Fogel, A. J. Owens, M. J. Walsh, *Artificial Intelligence Through Simulated Evolution* (Wiley, New York, 1966).
6. I. Rechenberg, "Cybernetic solution path of an experimental problem," *Royal Aircraft Establishment Transl. No. 1122*, B. F. Toms, Transl. (Ministry of Aviation, Royal Aircraft Establishment, Farnborough, Hants., United Kingdom, 1965).
7. T. Back and H. P. Schwefel, *Evol. Comput.* **1**, 1 (1993).
8. R. E. Smith and D. E. Goldberg, *Complex Syst.* **6**, 251 (1992).
9. W. D. Hillis, *Physica D* **42**, 228 (1990).
10. L. Davis and S. Coombs, in *Proceedings of the Second International Conference on Genetic Algorithms*, J. J. Grefenstette, Ed. (Erlbaum, Hillsdale, NJ, 1987), pp. 252–256.
11. D. E. Glover, in *Genetic Algorithms and Simulated Annealing*, L. Davis, Ed. (Pitman, London, and Morgan Kaufmann, Los Altos, CA, 1987), pp. 12–32.
12. J. R. Koza, *Genetic Programming* (MIT Press, Cambridge, MA, 1992).
13. R. Dawkins, in *Artificial Life*, C. G. Langton, Ed. (Addison-Wesley, Redwood City, CA, 1989), pp. 201–220.
14. K. Sims, *Comput. Graphics* **25**, 319 (July 1991).
15. J. D. Schaffer, in *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, J. J. Grefenstette, Ed. [Naval Center for Applied Research on Artificial Intelligence (NCARAI), Washington, DC, and Texas Instruments, Dallas, TX, 1985], pp. 93–100.
16. L. Davis, Ed., *Handbook of Genetic Algorithms* (Van Nostrand Reinhold, New York, 1991).
17. J. J. Grefenstette, Ed., *Proceedings of an International Conference on Genetic Algorithms and their Applications* (NCARAI, Washington, DC, and Texas Instruments, Dallas, TX, 1985).
18. ———, *Proceedings of the Second International Conference on Genetic Algorithms* (Erlbaum, Hillsdale, NJ, 1987).
19. J. D. Schaffer, Ed., *Proceedings of the Third International Conference on Genetic Algorithms* (Morgan Kaufmann, Los Altos, CA, 1989).
20. R. K. Belew and L. B. Booker, Eds., *Proceedings of the Fourth International Conference on Genetic Algorithms* (Morgan Kaufmann, Los Altos, CA, 1991).
21. N. H. Packard, *Complex Syst.* **4** (no. 5) (1990).
22. L. B. Booker, R. L. Riolo, J. H. Holland, in *Basic Paradigms, Learning Representation Issues, and Integrated Architectures, vol. 1 of Artificial Intelligence and Neural Networks*, V. Honavar and L. Uhr, Eds. (Academic Press, Cambridge, MA, in press).
23. K. A. DeJong, thesis, University of Michigan, Ann Arbor, MI (1975).
24. C. G. Shaefer, in (*10*), pp. 50–58.
25. N. N. Schraudolph and R. K. Belew, *Mach. Learn.* **9**, 9 (1991).
26. D. E. Goldberg, B. Korb, K. Deb, *Complex Syst.* **3**, 493 (1990).
27. Y. Davidor, *Genetic Algorithms and Robotics* (World Scientific, River Edge, NJ, 1991).
28. D. Whitley, T. Starkweather, D. Shaner, in (*16*), pp. 350–372.
29. D. E. Goldberg, *Complex Syst.* **5**, 139 (1991).
30. H. J. Antonisse and K. S. Keller, in (*10*), pp. 69–76.
31. D. E. Goldberg, *Commun. ACM*, in press.
32. H. Muhlenbein, M. Gorges-Schleuter, O. Kramer, *Parallel Comput.* **6**, 65 (1988).
33. D. Smith, in (*15*), pp. 202–206.
34. J. J. Grefenstette, R. Gopal, B. J. Rosmaita, D. Van Gucht, *ibid.*, pp. 160–168.
35. D. E. Goldberg and R. Lingle, Jr., *ibid.*, pp. 154–159.
36. G. Syswerda, in (*19*).
37. J. D. Schaffer, R. A. Caruana, L. J. Eshelman, in (*19*).
38. J. C. Bean, *ORSA J. Comput.*, in press.
39. N. Ulder, E. Aarts, H. Bandelt, P. van Laarhoven, E. Pesch, in *Parallel Problem Solving from Nature*, H. P. Schwefel and R. Manner, Eds. (Springer-Verlag, New York, 1991), pp. 109–116.
40. S. Forrest and M. Mitchell, *Mach. Learn.*, in press.
41. J. J. Grefenstette and J. E. Baker, in (*19*).
42. D. B. Fogel and J. W. Atmar, *Biol. Cybern.* **63**, 111 (1990).
43. J. L. Walsh, *Am. J. Math.* **55**, 5 (1923).
44. A. D. Bethke, thesis, University of Michigan, Ann Arbor, MI (1980).
45. D. E. Goldberg, *Complex Syst.* **3**, 153 (1989).
46. R. Tanese, thesis, University of Michigan, Ann Arbor, MI (1989).
47. J. H. Holland, in *Evolution, Learning, and Cognition*, Y. C. Lee, Ed. (World Scientific, River Edge, NJ, 1988).
48. L. B. Booker, in *Foundations of Genetic Algorithms 2*, L. D. Whitley, Ed. (Morgan Kaufmann, Los Altos, CA, 1993), pp. 29–44.
49. A. Bergman and M. W. Feldman, *Physica D* **56**, 57 (1992).
50. G. E. Liepins and M. D. Vose, *J. Exp. Theor. Artif. Intell.* **2**, 101 (1990).
51. D. E. Goldberg, K. Deb, J. H. Clark, *Complex Syst.* **6**, 333 (1992).
52. B. Manderick, M. de Weger, P. Spiessens, in (*20*).
53. J. P. Ros, in (*48*), pp. 257–276.
54. J. H. Holland, K. J. Holyoak, R. E. Nisbett, P. Thagard, *Induction: Processes of Inference, Learning, and Discovery* (MIT Press, Cambridge, MA, 1986).
55. J. H. Holland, in *Integrative Themes*, G. Cowan, D. Pines, D. Meltzer, Eds. (Addison-Wesley, Reading, MA, in press).
56. S. Forrest, B. Javornik, R. E. Smith, A. Perelson, *Evol. Comput.*, in press.
57. R. Axelrod, *Am. Polit. Sci. Rev.* (December 1986), p. 80.
58. ———, in (*11*).
59. S. W. Wilson, *Mach. Learn.* **2**, 199 (1987).
60. R. E. Smith, S. Forrest, A. S. Perelson, *Evol. Comput.*, in press.
61. R. Hightower, S. Forrest, A. S. Perelson, in *Proceedings of the Second European Conference on Artificial Life*, in press.
62. J. D. Farmer, N. H. Packard, A. S. Perelson, *Physica D* **22**, 187 (1986).
63. S. Forrest and M. Mitchell, in (*48*), pp. 109–126.

64. The author acknowledges the National Science Foundation (grant IRI-9157644), Sandia University Research Program (grant AE-1679), and the Alfred P. Sloan Foundation (grant B1992-46). Many people read and contributed comments on the manuscript. In particular, I would like to thank K. DeJong, D. Goldberg, J. Holland, and R. Palmer. D. Mathews and L. Desjarlais prepared the figures.