Genetic Algorithms: A Survey

M. Srinivas, Motorola India Electronics Ltd.

Lalit M. Patnaik, Indian Institute of Science

Genetic algorithms provide an alternative to traditional optimization techniques by using directed random searches to locate optimal solutions in complex landscapes. This article traces GA research.

In the last five years, genetic algorithms have emerged as practical, robust optimization and search methods. Diverse areas such as music generation, genetic synthesis, VLSI technology, strategy planning, and machine learning have profited from these methods. The popularity of genetic algorithms is reflected in three biennial conferences, a new international journal, and an ever-increasing mass of literature devoted to the theory, practice, and applications of such techniques (see the sidebar "To learn more").

Genetic algorithm search methods are rooted in the mechanisms of evolution and natural genetics. The interest in heuristic search algorithms with underpinnings in natural and physical processes began as early as the 1970s, when Holland[1] first proposed genetic algorithms. This interest was rekindled by Kirkpatrick, Gelatt, and Vecchi's simulated annealing technique in 1983.[2] Simulated annealing is based on thermodynamic considerations, with annealing interpreted as an optimization procedure. Evolutionary strategies[3,4] and genetic algorithms,[5-13] on the other hand, draw inspiration from the natural search and selection processes leading to the survival of the fittest individuals. Simulated annealing, genetic algorithms, and evolutionary strategies are similar in their use of a probabilistic search mechanism directed toward decreasing cost or increasing payoff. These three methods have a high probability of locating the global solution optimally in a multimodal search landscape. (A multimodal cost function has several locally optimal solutions as well.) However, each method has a significantly different mode of operation.

Simulated annealing probabilistically generates a sequence of states based on a *cooling schedule* to ultimately converge to the global optimum. Evolutionary strategies use mutations as search mechanisms and selection to direct the search toward the prospective regions in the search space. Genetic algorithms generate a sequence of populations by using a selection mechanism, and use crossover and mutation as search mechanisms. The principal difference between genetic algorithms and evolutionary strategies is that genetic algorithms rely on crossover, a mechanism of probabilistic and useful exchange of information among solutions, to locate better solutions, while evolutionary strategies use mutation as the primary search mechanism. Although

0018-9162/94/ $4.00 © 1994 IEEE

simulated annealing, evolutionary strategies, and genetic algorithms have basically different approaches, several hybrids of these techniques that narrow the distinctions among them have been proposed in recent literature. Other optimization algorithms derived from the evolutionary paradigm have also demonstrated considerable success.

In this article we introduce the art and science of genetic algorithms and survey current issues in GA theory and practice. We do not present a detailed study, since several wonderful texts on GAs already exist.[1,7,14,15] Instead, we offer a quick guide into the labyrinth of GA research. To start, we draw the analogy between genetic algorithms and the search processes in nature. Then we describe the genetic algorithm that Holland introduced in 1975 and the workings of GAs. After a survey of techniques proposed as improvements to Holland's GA and of some radically different approaches, we survey the advances in GA theory related to modeling, dynamics, and deception.

Genetic algorithms and natural selection

In nature, individuals best suited to competition for scanty resources survive. Adapting to a changing environment is essential for the survival of individuals of each species. While the various features that uniquely characterize an individual determine its survival capacity, the features in turn are determined by the individual's genetic content. Specifically, each feature is controlled by a basic unit called a gene. The sets of genes controlling features form the chromosomes, the "keys" to the survival of the individual in a competitive environment.

Although evolution manifests itself as a succession of changes in species' features, it is the changes in the species' genetic material that form the essence of evolution. Specifically, evolution's driving force is the joint action of natural selection and the recombination of genetic material that occurs during reproduction.

In nature, competition among individuals for scant resources such as food and space and for mates results in the fittest individuals dominating over weaker ones. Only the fittest individuals survive and reproduce, a natural phenomenon called "the survival of the fittest." Hence, the genes of the fittest survive, while the genes of weaker individuals die out. Natural selection leads to the survival of the fittest individuals, but it also implicitly leads to the survival of the fittest genes.

The reproduction process generates diversity in the *gene pool*. Evolution is initiated when the genetic material (chromosomes) from two parents recombines during reproduction. New combinations of genes are generated from previous ones; a new gene pool is created. Specifically, the exchange of genetic material among chromosomes is called *crossover*. Segments of the two parent chromosomes are exchanged during crossover, creating the possibility of the "right" combination of genes for better individuals. Repeated selection and crossover cause the continuous evolution of the gene pool and the generation of individuals that survive better in a competitive environment.

Holland[1] proposed genetic algorithms in the early 1970s as computer programs that mimic the evolutionary processes in nature. Genetic algorithms manipulate a population of potential solutions to an optimization (or search) problem. Specifically, they operate on encoded representations of the solutions, equivalent to the genetic material of individuals in nature, and not directly on the solutions themselves. Holland's genetic algorithm encodes the solutions as *strings* of bits from a binary alphabet. As in nature, selection provides the necessary driving mechanism for better solutions to survive. Each solution is associated with a *fitness value* that reflects how good it is, compared with other solutions in the population. The higher the fitness value of an individual, the higher its chances of survival and reproduction and the larger its representation in the subsequent generation. Recombination of genetic material in genetic algorithms is simulated through a crossover mechanism that exchanges portions between strings. Another operation, called mutation, causes sporadic and random alteration of the bits of strings. Mutation too has a direct analogy from nature and plays the role of regenerating lost genetic material.

To learn more

Readers wishing to pursue information on genetic algorithms may be interested in the following materials.

Proceedings

Proc. Int'l Conf. Genetic Algorithms and Applications, Lawrence Erlbaum Assoc., Hillsdale, N.J., 1985 and 1987.

Proc. Int'l Conf. Genetic Algorithms and Applications, Morgan Kaufmann, San Mateo, Calif., 1989, 1991, and 1993.

Proc. Workshop on Foundations Genetic Algorithms, Morgan Kaufmann, San Mateo, Calif., 1991 and 1992.

Proc. Workshop Parallel Problem-Solving from Nature, Springer-Verlag, Berlin, 1990.

International journal

Evolutionary Computation, MIT Press, Boston, Mass.

A simple genetic algorithm

In the literature, Holland's genetic algorithm is commonly called the Simple Genetic Algorithm or SGA. Essential to the SGA's working is a population of binary strings. Each string of 0s and 1s is the encoded version of a solution to the optimization problem. Using genetic operators — crossover and mutation — the algorithm creates the subsequent generation from the strings of the current population. This generational cycle is re-

peated until a desired termination criterion is reached (for example, a predefined number of generations are processed).

Figure 1 summarizes the working of the SGA, which has the following components:

- a population of binary strings,
- control parameters,
- a fitness function,
- genetic operators (crossover and mutation),
- a selection mechanism, and
- a mechanism to encode the solutions as binary strings.

Encoding mechanism. Fundamental to the GA structure is the encoding mechanism for representing the optimization problem's variables. The encoding mechanism depends on the nature of the problem variables. For example, when solving for the optimal flows in a transportation problem, the variables (flows in different channels) assume continuous values, while the variables in a traveling-salesman problem are binary quantities representing the inclusion or exclusion of an edge in the Hamiltonian circuit. In each case the encoding mechanism should map each solution to a unique binary string.

A large number of optimization problems have real-valued continuous variables. A common method of encoding them uses their integer representation. Each variable is first linearly mapped to an integer defined in a specified range, and the integer is encoded using a fixed number of binary bits. The binary codes of all the variables are then concatenated to obtain a binary string. For example, consider a continuous variable defined in a range from –1.28 to 1.28. We could encode this continuous variable with an accuracy of two decimal places by multiplying its real value by 100 and then discarding the decimal portion of the product. Thus the value that the variable attains is linearly mapped to integers in the range [–128, 128]. The binary code corresponding to each integer can be easily computed.

Fitness function. The objective function, the function to be optimized, provides the mechanism for evaluating each string. However, its range of values varies from problem to problem. To maintain uniformity over various problem domains, we use the *fitness function* to normalize the objective function to a convenient range of 0 to 1. The normalized value of the objective function is the *fitness* of the string, which the selection mechanism uses to evaluate the strings of the population.

```
Simple Genetic Algorithm ()
[
initialize population;
evaluate population;
while termination criterion not reached
   [
   select solutions for next population;
   perform crossover and mutation;
   evaluate population;
   ]
]
```

Figure 1. Simple Genetic Algorithm structure.

Selection. Selection models nature's survival-of-the-fittest mechanism. Fitter solutions survive while weaker ones perish. In the SGA, a fitter string receives a higher number of offspring and thus has a higher chance of surviving in the subsequent generation. In the *proportionate selection scheme*, a string with fitness value f_i is allocated $f_i/\bar{f}$ offspring, where $\bar{f}$ is the average fitness value of the population. A string with a fitness value higher than the average is allocated more than one offspring, while a string with a fitness value less than the average is allocated less than one offspring.

The proportionate selection scheme allocates fractional numbers of offspring to strings. Hence the number $f_i/\bar{f}$ represents the string's expected number of offspring. Since in the final allocation some strings have to receive a higher number of offspring than $f_i/\bar{f}$ and some less than $f_i/\bar{f}$, allocation methods include some randomization to remove methodical allocation biases toward any particular set of strings. The allocation technique controls the extent to which the actual allocation of offspring to strings matches the expected number of offspring $f_i/\bar{f}$.

The SGA uses the *roulette wheel selection scheme*[1] to implement proportionate selection. Each string is allocated a sector (slot) of a roulette wheel with the angle subtended by the sector at the center of the wheel equaling $2\pi\ f_i/\bar{f}$. A string is allocated an offspring if a randomly generated number in the range 0 to 2π falls in the sector corresponding to the string. The algorithm selects strings in this fashion until it has generated the entire population of the next generation. Roulette wheel selection could generate large sampling errors in the sense that the final number of offspring allocated to a string might vary significantly from the expected number. The allocated number of offspring approaches the expected number only for very large population sizes.

Crossover. After selection comes crossover, SGA's crucial operation. Pairs of strings are picked at random from the population to be subjected to crossover. The SGA uses the simplest approach — single-point crossover. Assuming that l is the string length, it randomly chooses a crossover point that can assume values in the range 1 to $l - 1$. The portions of the two strings beyond this crossover point are exchanged to form two new strings. The crossover point may assume any of the $l - 1$ possible values with equal probability. Further, crossover is not always effected. After choosing a pair of strings, the algorithm invokes crossover only if a randomly generated number in the range 0 to 1 is greater than p_c, the crossover rate. (In GA literature, the term crossover rate is also used to denote the probability of crossover.) Otherwise the strings remain unaltered. The value of p_c lies in the range from 0 to 1. In a large population, p_c gives the fraction of strings actually crossed.

Mutation. After crossover, strings are subjected to mutation. Mutation of a bit involves flipping it: changing a 0 to 1 or vice versa. Just as p_c controls the probability of a crossover, another parameter, p_m (the mutation rate), gives the probability that a bit will be flipped. The bits of a string are independently mutated — that is, the mutation of a bit does not affect the probability of mutation of other bits. The SGA treats mutation only as a secondary operator with the role of restoring lost genetic material. For example, suppose all the strings in a population have converged to a 0 at a given position and the optimal solution has a 1 at that position. Then crossover cannot regenerate a 1 at that position, while a mutation could.

Population P1:

String	Fitness value
0000011100	0.3
1000011111	0.6
0110101011	0.6
1111111011	0.9

Population P2 : After selection

String	Fitness value
1000011111	0.6
0110101011	0.6
1111111011	0.9
1111111011	0.9

Population P3 : After crossover

String	Fitness value
10000|11011	0.5
0110101011	0.6
1111111011	0.9
11111|11111	1.0

Population P4 : After mutation

String	Fitness value
1000011011	0.5
0110111011	0.7
1111111011	0.9
0111111111	0.9

Figure 2. A generational cycle of the Simple Genetic Algorithm.

Generational cycle. Figure 2 shows a generational cycle of the genetic algorithm with a population (P1) of four strings with 10 bits each. In the example, the objective function, which can assume values in the range 0 to 10, gives the number of 1s in the string. The fitness function performs a "divide by 10" operation to normalize the objective function to the range 0 to 1. The four strings thus have fitness values of 0.3, 0.6, 0.6, and 0.9. Ideally, the proportional selection scheme should allocate 0.5, 1.0, 1.0 and 1.5 offspring to the strings. However, in this case, the final allocation of offspring is 0, 1, 1, and 2. In Figure 2 the population P2 represents this selected set of strings. Next, the four strings are paired randomly for crossover. Strings 1 and 4 form one pair, while strings 2 and 3 form the other pair. At a crossover rate of 0.5, only the pair of strings 1 and 4 is actually crossed, while the other pair of strings is left intact. The crossover point falls between the fifth and sixth bits of the strings, and portions of strings 1 and 4 beyond the fifth bit are swapped. Population P3 represents the set of strings after crossover. The action of mutation on population P3 can be seen in population P4 on the sixth bit of string 2 and the first bit of string 4: Only two bits out of 40 have been mutated, representing an effective mutation rate of 0.05. Population P4 represents the next generation. (In effect, P1 and P4 are the populations, while P2 and P3 represent intermediate stages in the generational cycle.)

The example in Figure 2 is only for illustration. Typically the SGA uses a population size of 30 to 200, crossover rates from 0.5 to 1.0, and mutation rates from 0.001 to 0.05. These parameters — the population size, mutation rate, and crossover rate — are together referred to as the control parameters of the SGA and must be specified before its execution.

To terminate execution of the SGA, we must specify a stopping criterion. It could be terminated after a fixed number of generations, after a string with a certain high fitness value is located, or after all the strings in the population have attained a certain degree of homogeneity (a large number of strings have identical bits at most positions).

How do genetic algorithms work?

Despite successful use of GAs in a large number of optimization problems, progress on the theoretical front has been rather slow. A very clear picture of the workings of GAs has not yet emerged, but the *schema theory* and the *building-block hypothesis* of Holland and Goldberg[1,7] capture the essence of GA mechanics.

Similarity template. A schema is a similarity template describing a subset of strings with similarities at certain positions.[1,7] In other words, a schema represents a subset of all possible strings that have the same bits at certain string positions. As an example, consider strings with five bits. A schema **000 represents strings with 0s in the last three positions: the set of strings 00000, 01000, 10000, and 11000. Similarly, a schema 1*00* represents the strings 10000, 10001, 11000, and 11001. Each string represented by a schema is called an *instance* of the schema. Because the symbol * signifies that a 0 or a 1 could occur at the corresponding string position, the schema ***** represents all possible strings of five bits. The *fixed positions* of a schema are the string positions that have a 0 or a 1: in **000, the third, fourth, and fifth positions. The number of fixed positions of a schema is its *order*: **000 is of order 3. A schema's *defining length* is the distance between the outermost fixed positions. Hence, the defining length of **000 is 2, while the defining length of 1*00* is 3. Any specific string is simultaneously an instance of 2^l schemata (l is the string length).

Since a schema represents a subset of strings, we can associate a fitness value with a schema: the *average fitness of the schema*. In a given population, this is determined by the average fitness of instances of the schema. Hence, a schema's average fitness value varies with the population's composition from one generation to another.

Competition. Why are schemata important? Consider a schema with k fixed positions. There are $2^k - 1$ other schemata with the same fixed positions that can be obtained by considering all permutations of 0s and 1s at these k positions. Altogether, for k fixed positions, there are 2^k distinct schemata that generate a partitioning of all possible strings. Each such set of k fixed positions generates a *schema competition*, a survival competition among the 2^k schemata. Since there are 2^l possible combinations of fixed positions, 2^l distinct schema competitions are possible. The execution of the GA thus generates 2^l simultaneous schema competitions. The GA simultaneously, though not independently, attempts to solve all the 2^l schema competitions and locate the best schema for each set of fixed positions.

We can visualize the GA's search for the optimal string as a simultaneous competition among schemata to increase the number of their instances in the population. If we describe the optimal string as the juxtaposition of schemata with short defining lengths and high average fitness values, then the winners of the individual schema competitions could potentially form the optimal string. Such schemata with high fitness values and small defining lengths are appropriately called *building blocks*. The notion that

strings with high fitness values can be located by sampling building blocks with high fitness values and combining the building blocks effectively is called the building-block hypothesis.[1,7]

Building blocks. The genetic operators — crossover and mutation — generate, promote, and juxtapose building blocks to form optimal strings. Crossover tends to conserve the genetic information present in the strings to be crossed. Thus, when the strings to be crossed are similar, its capacity to generate new building blocks diminishes. Mutation is not a conservative operator and can generate radically new building blocks. Selection provides the favorable bias toward building blocks with higher fitness values and ensures that they increase in representation from generation to generation. GAs' crucial and unique operation is the juxtaposition of building blocks achieved during crossover, and this is the cornerstone of GA mechanics.

The building-block hypothesis assumes that the juxtaposition of good building blocks yields good strings. This is not always true. Depending on the nature of the objective function, very bad strings can be generated when good building blocks are combined. Such objective functions are referred to as *GA-deceptive functions*, and they have been studied extensively. (We discuss them in more detail in a later section.)

Schema theorem. When we consider the effects of selection, crossover, and mutation on the rate at which instances of a schema increase from generation to generation, we see that proportionate selection increases or decreases the number in relation to the average fitness value of the schema. Neglecting crossover, a schema with a high average fitness value grows exponentially to win its relevant schema competition. However, a high average fitness value alone is not sufficient for a high growth rate. A schema must have a short defining length too. Because crossover is disruptive, the higher the defining length of a schema, the higher the probability that the crossover point will fall between its fixed positions and an instance will be destroyed. Thus, schemata with high fitness values and small defining lengths grow exponentially with time. This is the essence of the schema theorem, first proposed by Holland as the "fundamental theorem of genetic algorithms."[1] (See the sidebar.)

The following equation is a formal statement of the schema theorem:

$$N(\mathbf{h},t+1) \geq N(\mathbf{h},t)\frac{f(\mathbf{h},t)}{\bar{f}(t)}\left[1 - p_C \frac{\delta(\mathbf{h})}{l-1} - p_m o(\mathbf{h})\right] \quad (1)$$

where

$f(\mathbf{h}, t)$: average fitness value of schema **h** in generation t

$\bar{f}(t)$: average fitness value of the population in generation t

p_c: crossover probability

p_m: mutation probability

$\delta(\mathbf{h})$: defining length of the schema

$o(\mathbf{h})$: order of the schema **h**

$N(\mathbf{h}, t)$: expected number of instances of schema **h** in generation t

l: the number of bit positions in a string

The factor:

$$p_c \frac{\delta(\mathbf{h})}{l-1}$$

gives the probability that an instance of the schema **h** is disrupted by crossover, and $p_m o(\mathbf{h})$ gives the probability that an instance is disrupted by mutation.[7]

The GA samples the building blocks at a very high rate. In a single generational cycle the GA processes only P strings (P is the population size), but it implicitly evaluates approximately P^3 schemata.[7] This capacity of GAs to simultaneously process a large number of schemata, called *implicit parallelism*, arises from the fact that a string simultaneously represents 2^l different schemata.

Modifications to the SGA

Over the last decade, considerable research has focused on improving GA performance. Efficient implementations of the proportionate selection scheme such as the stochastic remainder technique and the stochastic universal sampling technique have been proposed to reduce sampling errors. Selection mechanisms such as rank-based selection, elitist strategies, steady-state selection, and tournament selection have been proposed as alternatives to proportional selection. Crossover mechanisms such as two-point, multi-point, and uniform have been proposed as improvements on the traditional single-point crossover technique. Gray codes and dynamic encoding have overcome some problems associated with fixed-point integer encoding. Departing from the traditional policy of static control parameters for the GA, adaptive techniques dynamically vary the control parameters (crossover and mutation rates). Significant innovations include the distributed genetic algorithms and parallel genetic algorithms. The rest of this section surveys these developments.

Selection mechanisms and scaling. The proportionate selection scheme allocates offspring based on the ratio of a string's fitness value to the population's average fitness value. In the initial generations of the GA, the population typically has a low average fitness value. The presence of a few strings with relatively high fitness values causes the proportionate selection scheme to allocate a large number of offspring to these "superstrings," and they take over the population, causing premature convergence. A different problem arises in the later stages of the GA when the population has converged and the variance in string fitness values becomes small. The proportionate selection scheme allocates approximately equal numbers of offspring to all strings, thereby depleting the driving force that promotes better strings. Scaling mechanisms and rank-based selection schemes overcome these two problems.

Scaling of fitness values involves readjustment of string fitness values. Linear scaling computes the scaled fitness value as

$$f' = af + b$$

where f is the fitness value, f' is the scaled fitness value, and a and b are suitably chosen constants. Here a and b are calculated in each generation to ensure that the maximum value of the scaled fitness value is a small number, say 1.5 or 2.0 times the average fitness value of the population. Then the maximum number of offspring allocated to a string is 1.5 or 2.0. Sometimes the scaled fitness values may become negative for strings that have fitness values far smaller than the average fitness of the population. In such cases, we must recompute a and b appropriately to avoid negative fitness values.

One way to overcome the problem of negative scaled fitness values is simply to remove these "troublemakers" from the competition. The *sigma truncation scheme* does exactly this by considering the standard deviation of fitness values before scaling them. Hence the fitness values of strings are determined as follows:

$$f' = f - (\bar{f} - c\sigma)$$

where $\bar{f}$ is the average fitness value of the population, σ is the standard deviation of fitness values in the population, and c is a small constant typically ranging from 1 to 3.

Strings whose fitness values are less than c standard deviations from the average fitness value are discarded. This approach ensures that most strings in the population (those whose fitness values are within c standard deviations of the average) are considered for selection, but a few strings that could potentially cause negative scaled fitness values are discarded.

An alternate way to avoid the twin problems that plague proportional selection is rank-based selection, which uses a fitness value-based rank of strings to allocate offspring. The scaled fitness values typically vary linearly with the rank of the string. The absolute fitness value of the string does not directly control the number of its offspring. To associate each string with a unique rank, this approach sorts the strings according to their fitness values, introducing the drawback of additional overhead in the GA computation.

Another mechanism is *tournament selection*. For selection, a string must win a competition with a randomly selected set of strings. In a k-ary tournament, the best of k strings is selected for the next generation.

In either proportionate selection (with or without scaling) or rank-based selection, the expected number of offspring is not an integer, although only integer numbers of offspring may be allocated to strings. Researchers have proposed several implementations to achieve a distribution of offspring very close to the expected numbers of offspring.

Considerable research has focused on improving GA performance. Innovations include distributed and parallel GAs.

The *stochastic remainder technique* deterministically assigns offspring to strings based on the integer part of the expected number of offspring. It allocates the fractional parts in a roulette wheel selection (stochastic selection) to the remaining offspring, thus restricting randomness to only the fractional parts of the expected numbers of offspring.

Each iteration of the simple GA creates an entirely new population from an existing population. GAs that replace the entire population are called *generational GAs*. GAs that replace only a small fraction of strings at a time are called *steady-state GAs*. Typically, new strings created through recombination replace the worst strings (strings with the lowest fitness values). Functionally, steady-state GAs differ from generational GAs in their use of populational elitism (preservation of the best strings), large population sizes, and high probabilities of crossover and mutation. The elitist selection strategy balances the disruptive effects of high crossover and mutation rates.

Crossover mechanisms. Because of their importance to GA functioning, much of the literature has been devoted to different crossover techniques and their analysis. This section discusses the important techniques.

Traditionally, GA researchers set the number of crossover points at one or two. In the two-point crossover scheme, two crossover points are randomly chosen and segments of the strings between them are exchanged. Two-point crossover eliminates the single-point crossover bias toward bits at the ends of strings.

An extension of the two-point scheme, the multipoint crossover, treats each string as a ring of bits divided by k crossover points into k segments. One set of alternate segments is exchanged between the pair of strings to be crossed.

Uniform crossover exchanges bits of a string rather than segments. At each string position, the bits are probabilistically exchanged with some fixed probability. The exchange of bits at one string position is independent of the exchange at other positions.

Recent GA literature has compared various techniques, particularly single-point and two-point crossover on the one hand, and uniform crossover on the other. To classify techniques, we can use the notions of *positional* and *distributional biases*. A crossover operator has positional bias if the probability that a bit is swapped depends on its position in the string. Distributional bias is related to the number of bits exchanged by the crossover operator. If the distribution of the number is nonuniform, the crossover operator has a distributional bias. Among the various crossover operators, single-point crossover exhibits the maximum positional bias and the least distributional bias. Uniform crossover, at the other end of the spectrum, has maximal distributional bias and minimal positional bias.

Empirical and theoretical studies have compared the merits of various crossover operators, particularly two-point and uniform crossover. At one end, uniform crossover swaps bits irrespective of their position, but its higher disruptive nature often becomes a drawback. Two-point and single-point crossover preserve schemata because of their low disruption rates, but they become less exploratory when the population becomes homogeneous.

A related issue is the interplay between the population size and the type of crossover. Empirical evidence suggests that uniform crossover is more suitable

for small populations, while for larger populations, the less disruptive two-point crossover is better. Uniform crossover's disruptiveness helps sustain a highly explorative search in small populations. The inherent diversity in larger populations reduces the need for exploration and makes two-point crossover more suitable.

A rather controversial issue strikes at the heart of GA workings: Is crossover an essential search mechanism, or is mutation alone sufficient for efficient search? Experimental evidence shows that for some objective functions mutation alone can locate the optimal solutions, while for objective functions involving high epistaticity (nonlinear interactions among the bits of the strings), crossover performs a faster search than mutation. On the other hand, crossover has long been accepted as more useful when optimal solutions can be constructed by combining building blocks (schemata with short defining lengths and high average fitness values), indicating which requires linear interactions among the string bits. The question is whether the experimental evidence and the general consensus about the utility of crossover are contradictory. Or is crossover beneficial in most objective functions that have either linear or nonlinear interactions? These questions are far from being resolved, and considerable theoretical and empirical evidence must be gathered before any definite conclusions can be drawn.

Control parameters. We can visualize the functioning of GAs as a balanced combination of exploration of new regions in the search space and exploitation of already sampled regions. This balance, which critically controls the performance of GAs, is determined by the right choice of control parameters: the crossover and mutation rates and the population size.

The choice of the optimal control parameters has been debated in both analytical and empirical investigations. Here we point out the trade-offs that arise:

- Increasing the crossover probability increases recombination of building blocks, but it also increases the disruption of good strings.
- Increasing the mutation probability tends to transform the genetic search into a random search, but it also helps reintroduce lost genetic material.
- Increasing the population size increases its diversity and reduces the probability that the GA will prematurely converge to a local optimum, but it also increases the time required for the population to converge to the optimal regions in the search space.

We cannot choose control parameters until we consider the interactions between the genetic operators. Because they cannot be determined independently, the choice of the control parameters itself can be a complex nonlinear optimization problem. Further, it is becoming evident that the optimal control parameters critically depend on the nature of the objective function.

Nontraditional techniques including dynamic and adaptive strategies have also been proposed to improve performance.

Although the choice of optimal control parameters largely remains an open issue, several researchers have proposed control parameter sets that guarantee good performance on carefully chosen testbeds of objective functions. Two distinct parameter sets have emerged: One has a small population size and relatively large mutation and crossover probabilities, while the other has a larger population size, but much smaller crossover and mutation probabilities. Typical of these two categories are

- crossover rate: 0.6, mutation rate: 0.001, population size: 100;[16] and
- crossover rate: 0.9, mutation rate: 0.01, population size: 30.[8]

The first set of parameters clearly gives mutation a secondary role, while the second makes it more significant. The high crossover rate of 0.9 in the second set also indicates that a high level of string disruption is desirable in small populations.

Encodings. Critical to GA performance is the choice of the underlying encoding for solutions of the optimization problem. Traditionally, binary encodings have been used because they are easy to implement and maximize the number of schemata processed. The crossover and mutation operators described in the previous sections are specific only to binary encodings. When alphabets other than [0,1] are used, the crossover and mutation operators must be tailored appropriately.

A large number of optimization problems have continuous variables that assume *real* values. A common technique for encoding continuous variables in the binary alphabet uses a fixed-point integer encoding — each variable is encoded using a fixed number of binary bits. The binary codes of all the variables are concatenated to obtain the strings of the population. A drawback of encoding variables as binary strings is the presence of *Hamming cliffs:* large Hamming distances between the binary codes of adjacent integers. For example, 01111 and 10000 are the integer representations of 15 and 16, respectively, and have a Hamming distance of 5. For the GA to improve the code of 15 to that of 16, it must alter all bits simultaneously. Such Hamming cliffs present a problem for the GA, as both mutation and crossover cannot overcome them easily. *Gray codes* suggested to alleviate the problem ensure that the codes for adjacent integers always have a Hamming distance of 1. However, the Hamming distance does not monotonously increase with the difference in integer values, and this phenomenon introduces Hamming cliffs at other levels.

Nontraditional techniques in GAs. The previous sections described selection and crossover techniques developed as natural extensions of the simple GA. Hence the techniques still have the traditional mold: binary encodings, statically defined control parameters, and fixed-length encodings. Recently, a wide spectrum of variants has broken away from the traditional setup. The motivation has been the performance criterion: to achieve better GA performance on a wide range of application problems. We refer to these as nontraditional techniques.

Dynamic and adaptive strategies. In practical situations, the static configurations of control parameters and encod-

ings in GAs have some drawbacks. Parameter settings optimal in the earlier stages of the search typically become inefficient during the later stages. Similarly, encodings become too coarse as the search progresses, and the fraction of the search space that the GA focuses its search on becomes progressively smaller. To overcome these drawbacks, several dynamic and adaptive strategies for varying the control parameters and encodings have been proposed. One strategy exponentially decreases mutation rates with increasing numbers of generations, to gradually decrease the search rate and disruption of strings as the population converges in the search space. Another approach considers dynamically modifying the rates at which the various genetic operators are used, based on their performance. Each operator is evaluated for the fitness values of strings it generates in subsequent generations.

Very often, after a large fraction of the population has converged (the strings have become homogeneous), crossover becomes ineffective in searching for better strings. Typically, low mutation rates (0.001 to 0.01) are inadequate for continuing exploration. In such a situation, a dynamic approach for varying mutation rates based on the Hamming distance between strings to be crossed can be useful. The mutation rate increases as the Hamming distance between strings decreases. As the strings to be crossed resemble each other to a greater extent, the capacity of crossover to generate new strings decreases, but the increased mutation rate sustains the search.

The dynamic encoding of variables in several implementations (DPE, Argot, and Delta Encoding) increases the search resolution as the GA converges. While strings are encoded using the same number of bits, the size of the search space in which strings are sampled is progressively reduced to achieve a higher search resolution.

Another adaptive strategy of encoding ("messy" GAs) explicitly searches low-order, high-fitness value schemata in the initial stages and then juxtaposes the building blocks with a splicing operator to form optimal strings. This technique has successfully optimized deceptive functions, which can cause the Simple GA to converge to local optima.

Distributed and parallel GAs. Distributed GAs and parallel GAs decentralize the processing of strings. Although they sound similar, the two approaches are basically different. Distributed GAs have a number of weakly interacting subpopulations, and each carries out an independent search. Parallel GAs are parallel implementations of the "sequential" GA on several computation engines to speed execution.

Distributed GAs distribute a large population into several smaller subpopulations that evolve independently. Thus, the exploration arising from a large population is evident, but the convergence rates of the subpopulations are also high. To ensure global competition among strings, the best strings of the subpopulations are exchanged. A distributed GA can be implemented on a single computation engine or in parallel with each subpopulation processed by a different engine.

Researchers are developing models of GA dynamics, analyzing problems difficult for GAs, and studying how GAs work.

Parallel GAs have emerged primarily to enable execution on parallel computers. Issues such as local and global communication, synchronization, and efficacy of parallel computation have led to modifications of the GA structure. Techniques such as local-neighborhood selection have been introduced to increase computation speed.

Advances in theory

The emergence of new GA implementations for better performance has been accompanied by considerable theoretical research, especially in developing models of GA dynamics, analyzing problems that are hard for GAs, and, most important, gaining a deeper understanding of how GAs work.

To analyze the working of the simple GA, Holland compared it with the k-armed bandit problem.[1] This problem discusses the optimal allocation of trials among k alternatives, each of which has a different payoff, to maximize the total payoff in a fixed number of trials. The payoff of each alternative is treated as a random variable. The distribution of payoffs from the different alternatives is not known a priori and must be characterized based on the payoffs observed during the trials. Holland demonstrated that the GA simultaneously solves a number of such k-armed bandit problems.

Consider the competition among schemata of order m that have the same fixed positions. There are 2^m competing schemata, and the GA allocates trials to them to locate the fittest. Totally, there are 2^l (l is the string length) such competitions occurring in parallel, with the GA attempting to solve all simultaneously. The exponential allocation of trials to the fittest strings by the GA is a near optimal allocation strategy, as it resembles the optimal solution to the k-armed bandit problem.[7]

The schema theorem[1] calculates a lower bound on the expected number of schemata under the action of selection, crossover, and mutation. Although the schema theorem captures the essence of the GA mechanism, its applicability in estimating the proportions of various schemata in the population is limited. Attempts to refine the schema theorem model the effects of crossover between instances of the same schema. To make the schema theorem more useful, expressions for the percentage of schema instances generated by crossover and mutation have been derived. The additional terms have extended the inequality of the schema theorem into an equation. However, the abstract nature of the calculations involved in computing these terms reduces the applicability of the schema "equation."

A generalization of schemata defined by Holland has been proposed. It views as a *predicate* the condition for a string to be included as an instance of a schema. This general definition allows

2^{2^l}

predicates to exist, compared with the 3^l Holland schemata for strings of length l. While the schema theorem remains valid for these generalized predicates, we can study several new interesting properties regarding their stability and dominance under the action of the genetic operators.

GA dynamics. The GA's population dynamics are controlled by the parameters population size, mutation rate, and crossover rate. Characterizing the dynamics — not a simple task — is important for understanding the conditions under which the GA converges to the global optimum.

Most work related to the dynamics of GAs looks at convergence results from one of two perspectives:

(1) finite versus infinite population results, or
(2) homogeneous versus inhomogeneous convergence results.

The first classification is self-explanatory. The second arises from the state-transition probabilities of the Markov processes that model the GA. If the state-transition probabilities are invariant over generations, we have a homogeneous Markov chain.

For the finite population case, we can consider each distinct population as a possible state of a Markov chain, with the state-transition matrix indicating the probabilities of transitions between the populations due to the genetic operators. When the mutation probability is not zero, every population can be *reached* from every other population with some nonzero probability. This property guarantees the existence of a unique *fixed point* (a limiting distribution) for the distribution of populations. For a zero mutation probability (with only selection and crossover), any population consisting of multiple copies of a single string is a possible fixed point of the random process modeling the GA.

Consider a nonstandard replacement operator after crossover that ensures the following property: For every bit position i there exist strings in the population having a 0 and a 1 at the position i. With this we can show that every population is reachable in a finite number of generations. The replacement operator substitutes one of the strings in the population with another string so the population satisfies the defined property. Further, the property also guarantees convergence of the GA to the global optimum with the probability of 1.0.

While these results summarize the homogeneous case, the main inhomogeneous result for finite populations is the demonstration of an exponential annealing schedule that guarantees convergence of the GA to one of the fixed points of the homogeneous case without mutation. However, this does not mean that the population corresponding to this fixed point contains only the global optimum. Empirical evidence suggests that as the population size increases, the probability mass of the limit distributions is concentrated at the optimal populations.

In infinite populations, we need model only the proportions of strings. We can model the evolution of populations as the interleaving of a quadratic operator representing crossover and mutation, and a linear operator representing selection. When only selection and crossover are considered, all limit points of the probability distribution have mass only at the most fit strings. With mutation and uniform selection, the uniform distribution is the unique fixed point.

We are faced with an important question: what problems mislead GAs to local optima?

Although it is important to establish the global convergence of GAs, it is equally important to have GAs with good rates of convergence to the global optimum. We believe that a major direction for future research on the dynamics of GAs is the establishment of bounds on the convergence rates of the GA under various conditions.

Deception. An important control on the dynamics of GAs is the nature of the search landscape. We are immediately confronted with a question: What features in search landscapes can GAs exploit efficiently? Or more to the point: What problems mislead GAs to local optima?

GAs work by recombining low-order, short schemata with above-average fitness values to form high-order schemata. If the low-order schemata contain the globally optimal solution, then the GA can potentially locate it. However, with functions for which the low-order high-fitness value schemata do not contain the optimal string as an instance, the GA could converge to suboptimal strings. Such functions are called *deceptive*.[7] Recently, considerable research has focused on the analysis and design of deceptive functions.

The simplest deceptive function is the *minimal deceptive problem,* a two-bit function. Assuming that the string "11" represents the optimal solution, the following conditions characterize this problem:

$$\begin{aligned} &f(11) > f(00) \\ &f(11) > f(01) \\ &f(11) > f(10) \\ &f(*0) > f(*1) \text{ or } f(0*) > f(1*) \end{aligned} \quad (2)$$

The lower order schemata $0*$ or $*0$ do not contain the optimal string 11 as an instance and lead the GA away from 11. The minimal deceptive problem is a partially deceptive function, as both conditions of Equation 2 are not satisfied simultaneously. In a *fully deceptive problem*, all the lower order schemata that contain the optimal string have lower average fitness values than their competitors (other schemata with the same fixed positions).

The minimal deceptive problem can easily be extended to higher string lengths. GA literature abounds with analyses of deceptive functions, conditions for problems to be deceptive, and ways of transforming deceptive functions into nondeceptive ones.

Some recent studies have investigated the implications of GA deceptiveness in the context of problems that are *hard* — that is, difficult for GAs to optimize. While it appears that a deceptive objective function offers some measure of difficulty for GAs, there has been some recent consensus that deception is neither a sufficient nor a necessary condition for a problem to be hard. At the heart of this argument is the observation that the definition of deception in GAs derives from a static hyperplane analysis which does not account for the potential difference of GAs' dynamic behavior from static predictions. Empirical work shows that some nondeceptive functions cannot be optimized easily by GAs, while other deceptive functions are easily optimized. Essentially, other features such as improper problem representations, the disruptive nature of crossover and mutation, finite population sizes, and multimodal landscapes could be potential causes of hardness.

Invented in the early 1970s, genetic algorithms only recently have gained considerable popularity as general-purpose robust optimization and search techniques. The failure of traditional optimization techniques in searching complex, uncharted and vast-payoff landscapes riddled with multimodality and complex constraints has generated interest in alternate approaches. Genetic algorithms are particularly attractive because instead of a naive "search and select" mechanism they use crossover to exchange information among existing solutions to locate better solutions.

Despite the algorithms' success, some open issues remain:

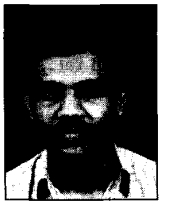
- the choice of control parameters,
- the exact roles of crossover and mutation,
- the characterization of search landscapes amenable to optimization, and
- convergence properties.

Limited empirical evidence points to the efficacy of distributed and parallel GAs and the adaptive strategies for varying control parameters. However, more experimental evidence is needed before we draw any definite conclusions about comparative performance.

GAs are emerging as an independent discipline, but they demand considerable work in the practical and theoretical domains before they will be accepted at large as alternatives to traditional optimization techniques. We hope this article stimulates interest in GAs and helps in their establishment as an independent approach for optimization and search. ■

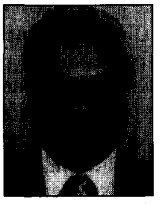
References

1. J.H. Holland, *Adaptation in Natural and Artificial Systems*, Univ. of Michigan Press, Ann Arbor, Mich., 1975.
2. S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi, "Optimization by Simulated Annealing," *Science*, Vol. 220, No. 4598, May 1983, pp. 671-681.
3. I. Rechenberg, *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologische Evolution [Evolutionary Strategy: Optimization of Technical Systems According to the Principles of Biological Evolution]*, Frommann Holzboog Verlag, Stuttgart, Germany, 1973.
4. H.P. Schwefel, *Numerical Optimization of Computer Models*, Wiley, Chichester, UK, 1981.
5. K.A. DeJong, *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, doctoral dissertation, Univ. of Michigan, Ann Arbor, Mich., 1975.
6. S. Forrest and M. Mitchell, "What Makes a Problem Hard for a Genetic Algorithm? Some Anomalous Results and their Explanation," in *Machine Learning*, Vol. 13, 1993, pp. 285-319.
7. D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, Mass., 1989.
8. J.J. Grefenstette, "Optimization of Control Parameters for Genetic Algorithms," *IEEE Trans. Systems, Man, and Cybernetics*, Vol. SMC-16, No. 1, Jan./Feb. 1986, pp. 122-128.
9. H. Mühlenbein et al., "The Parallel GA as a Function Optimizer," *Proc. Fourth Int'l Conf. Genetic Algorithms*, Morgan Kaufmann, San Mateo, Calif., 1991, pp. 279-288.
10. J.D. Schaffer et al., "A Study of Control Parameters Affecting On-line Performance of Genetic Algorithms for Function Optimization," *Proc. Third Int'l Conf. Genetic Algorithms*, Morgan Kaufmann, San Mateo, Calif., 1989, pp. 51-60.
11. M. Srinivas and L.M. Patnaik, "Adaptive Probabilities of Crossover and Mutation in Genetic Algorithms," *IEEE Trans. Systems, Man, and Cybernetics*, Apr. 1994.
12. M. Srinivas and L.M. Patnaik, "Binomially Distributed Populations for Modeling Genetic Algorithms," *Proc. Fifth Int'l Conf. Genetic Algorithms*, Morgan Kaufmann, San Mateo, Calif., 1993, pp. 138-145.
13. D. Whitley and T. Starkweather, "Genitor-II: A Distributed Genetic Algorithm," *J. Experimental Theoretical Artificial Intelligence*, Vol. 2, 1990, pp. 189-214.
14. L. Davis, ed., *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, 1991.
15. Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolutionary Programs*, Springer-Verlag, Berlin, 1992.
16. K.A. DeJong and W.M. Spears, "An Analysis of the Interacting Roles of Population Size and Crossover in Genetic Algorithms," *Proc. First Workshop Parallel Problem Solving from Nature*, Springer-Verlag, Berlin, 1990, pp. 38-47.



M. Srinivas is employed at Motorola India Electronics Ltd., where his research interests are in theory and design of genetic algorithms, neural networks, stochastic optimization, and optimization in VLSI CAD algorithms. He has also worked for the Centre for Development of Advanced Computing.

Srinivas is a PhD candidate in computer science and automation at the Indian Institute of Science, Bangalore.



Lalit M. Patnaik is a professor in the Electrical Sciences Division of the Indian Institute of Science, where he directs a research group in the Microprocessor Applications Laboratory. His teaching, research, and development interests are in parallel and distributed computing, computer architecture, computer-aided design of VLSI systems, computer graphics, theoretical computer science, real-time systems, neural computing, and genetic algorithms. In the areas of parallel and distributed computing and neural computing, he has been a principal investigator for government-sponsored research projects and a consultant to industry.

Patnaik received his PhD for work in real-time systems in 1978 and his DSc in computer systems and architectures in 1989, both from the Indian Institute of Science. He is a fellow of the IEEE, Indian National Science Academy, Indian Academy of Sciences, National Academy of Sciences, and Indian National Academy of Engineering. For the last two years, he has served as chair of the IEEE Computer Society chapter, Bangalore section.

Srinivas can be reached at Motorola India Electronics Ltd., No. 1, St. Marks Road, Bangalore 560 001, India; e-mail: msrini@master.miel.mot.com. Patnaik can be contacted at the Microprocessor Applications Laboratory, Indian Institute of Science, Bangalore 560 012, India; e-mail: lalit@micro.iisc.ernet.in.