GENETIC ALGORITHMS

Genetic algorithms (commonly known as GAs) are primarily search and optimization algorithms which work on the basis of the processing of chromosomes in natural genetics. The chromosomal processing in natural genetics and the working of GAs have at least one aspect in common. Many biologists believe that the continuing evolution of better and fitter chromosomes has taken place primarily to maximize the DNA survivability of a species (1). In GAs, better and better solutions are artificially evolved to maximize an objective or goal of the underlying search and optimization problem.

Although a number of classical search and optimization methods exist, and these methods have been used to solve many scientific and engineering problems over many years, these methods have certain shortcomings. They usually do not work well in solving problems with multiple optimal solutions or problems with discrete search space. Moreover, most of them are specialized for solving a particular class of problems and do not work as well in solving other types of problems. GAs are flexible yet efficient and do not have most of the difficulties of the classical methods. That is why GAs are gaining popularity in solving search and optimization problems in various problem domains.

GAs were developed in the early sixties by Professor John Holland of the University of Michigan. His book entitled *Adaptation in Natural and Artificial Systems* was published by MIT Press in 1975 (2). Thereafter, a number of his students and other researchers contributed to developing the field (3–8). To date, most of the GA studies are available through a number of books (9–12) and through many international conference and workshop proceedings (13–22). Although GA-related papers are now published in most reputed journals, there are now at least two dedicated journals on the topic by the same name, *Evolutionary Computation,* published by MIT Press and by IEEE.

A BRIEF INTRODUCTION TO GENETIC ALGORITHMS

The working principle of GAs is as follows. To simulate chromosomal processing, GAs first represent problem or decision variables in a string coding, resembling a chromosome. For example, a five-bit binary coding `11001` may represent a particular value (solution) of a decision variable in a predefined range. Each bit (a `1` or a `0`) may be assumed an allele of a gene denoted by its position. The previous solution has an allele `1` in its first, second, and fifth genes. Because a string represents a solution, its value with respect to the underlying objective can be evaluated. The merit of a string is known as the *fitness* in the parlance of GAs. GAs begin with a population of such strings created at random. After the fitness is evaluated for each string, GAs perform a series of operations on this initial random population to create a new (and hopefully better) population of strings. In most GAs, three operations similar to three natural genetic operations are performed. They are reproduction, crossover, and mutation.

The reproduction operator replaces worse solutions in the population by making duplicates of the better solutions. Thus, the reproduction operator acts like a sieve which allows only better solutions in a population to qualify and worse solutions to die. Although this operator improves the quality of the current population, it cannot create any new solutions. The creation of new solutions is achieved by using crossover and mutation operators.

In the crossover operation, two solutions are chosen at random from the pool of good solutions obtained after the reproduction operation, and two new solutions are created by exchanging certain information with each other. In its simplest form, both strings are cut at an arbitrary place and the right-side portions of both strings are swapped among themselves to create two new strings, as illustrated in the following:

```
Parent 1    0 0 | 0 0 0        0 0 | 1 1 1    Child
                          ⇒
Parent 2    1 1 | 1 1 1        1 1 | 0 0 0    Child
```

Although the choice of the cross-site is random, it is interesting to note that this operator can combine good features (allelic combinations) of two different solutions in one string.

The mutation operator changes a bit to its complement with a mutational probability. In its simplest form, every bit is checked for mutation with a small mutational probability. If the bit is to be mutated, the bit is changed to its complement. Otherwise it is left unaltered, as illustrated in the following:

```
00000 ⇒ 000:
```

In this example, the fourth gene has changed its value, thereby creating a new solution. After reproduction, crossover, and mutation are applied to the whole population, one cycle of GAs (usually known as a *generation*) is completed.

Like other classical search and optimization methods (23–25), GAs also work iteratively. The algorithm is terminated when a termination criterion is satisfied. The termination criterion can be set in a number of different ways: a prespecified number of generations has elapsed, a satisfactory solution has been found, or the population-best solution has not changed for a consecutive fixed number of generations.

This description indicates that the working principle of GAs is very different from that of the classical search and optimization techniques. As outlined in (10), there are four basic differences:

1. GAs do not use the problem variables. Instead they use a coding of problem variables.
2. GAs operate on a population of solutions and create another population of solutions, instead of operating on one solution and creating another solution.

3. GAs do not require that the objective function and constraints be differentiable or continuous, thereby broadening their scope of application.
4. GAs use probabilistic operators, instead of deterministic operators.

These flexibilities in the operation of GAs allow using them in a wide variety of problem domains. Moreover, they allow GAs to have a global search, which most classical methods cannot achieve (23).

Although the previous discussion may seem descriptive and qualitative, mathematical analyses for a convergence proof of GAs are now being attempted by many researchers (26–30).

GAs have been largely applied in the search and optimization problems of science, engineering, and commerce. In the field of electrical and electronics engineering, GAs have been applied to VLSI circuit layout design problems (31), power distribution systems (32), communication networking (33), among others. In solving neural networks problems, GAs have been used in two ways. The GA has been used as a learning algorithm (instead of popular back propagation or other algorithms) to find optimal weights (34). The GA has also been used to find an optimal network (35). Some studies exist where both problems of finding the optimal network and finding optimal weights for the interconnections are tackled with GAs (36). GAs are also used in applications of fuzzy sets, the performance of which depends on the proper definition of membership functions. GAs are used to find the optimal membership functions (37–39). GAs have also been used as a rule discovery mechanism in machine learning applications (10).

In many applications, it has been observed that the simple GA previously described quickly converges to a near-optimal solution. However, it takes a significant number of function evaluations to converge to the true optimal solution. To alleviate this problem of perfect convergence, researchers have suggested using a hybrid GA, a combination of GA with a traditional hill-climbing algorithm (40–44). A detailed discussion of these techniques is given later. In the following, we present a more detailed description of GAs used in function optimization problems.

GAs AS FUNCTION OPTIMIZERS

We first discuss the procedure for using GAs in solving unconstrained, single-variable optimization problems and later discuss the procedure for solving constrained, multivariable optimization problems. Let us consider the following unconstrained optimization problem:

$$\begin{aligned} &\text{Maximize} && f(x) \\ &\text{Variable bound} && x_{\min} \le x \le x_{\max} \end{aligned} \tag{1}$$

To use GAs to solve this problem, the variable x is typically coded in finite-length binary strings. The length of the string is usually determined by the accuracy of the solution desired. For example, if five-bit binary strings are used to code the variable x, then the string (`0 0 0 0 0`) is decoded to the value $x_{\min}$, the string (`1 1 1 1 1`) is decoded to the value $x_{\max}$, and any other string is decoded uniquely to a particular value in the range ($x_{\min}$, $x_{\max}$). It is worthwhile to mention here that with five bits in a string, there are only 2^5 or 32 different strings possible because each bit-position takes a value `0` or `1`. In practice, strings of one hundred or a few hundreds are common. Recently a coding with string size equal to 16,384 was used (42). Thus, with an ℓ-bit string to code the variable x, the accuracy between two consecutive strings is roughly $(x_{\max} - x_{\min})/2^{\ell}$. It is also noteworthy that, as the string length increases, the minimum possible accuracy in the solution increases exponentially. We shall see later that the choice of the string length affects the computational time required to solve the problem to a desired accuracy. With a known coding, any string can be decoded to an x value, which can then be used to find the objective functional value. A string's objective functional value $f(x)$ is known as the string's *fitness*.

The following is a pseudocode for a genetic algorithm:

```
begin
    Initialize population;
    Evaluate population;
    repeat
        Reproduction;
        Crossover;
        Mutation;
        Evaluate population;
    until (termination criteria);
end.
```

As mentioned earlier, GAs begin with a population of strings (or x values) created at random. Thereafter, each string in the population is evaluated. Then the population is operated by three main operators, reproduction, crossover, and mutation, hopefully, to create a better population. The population is further evaluated and tested for termination. If the termination criteria are not met, the population is again operated on by the three operators and evaluated. This procedure is continued until the termination criteria are met. One cycle of these operators and the evaluation procedure is known in GA terminology as a *generation.*

Reproduction is usually the first operator applied on a population. Reproduction selects good strings in a population and forms a mating pool.A number of reproduction operators exists in the GA literature (45), but the essential idea is that above-average strings are picked from the current population and duplicates of them are inserted in the mating pool. The commonly used reproduction operator is the proportionate selection operator, where a string in the current population is selected with a probability proportional to the string's fitness. Thus, the ith string in the population is selected with a probability proportional to f_i. Because the population size is usually fixed in a simple GA, the cumulative probability for all strings in the population must be 1. Therefore, the probability for selecting the ith string is $f_i/\sum_{j=1}^{N} f_j$, where N is the population size. One way to achieve proportionate selection is to use a roulette wheel whose circumference is marked for each string proportional to the string's fitness. The roulette wheel is spun N times, each time keeping an instance of the string, selected by the roulette-wheel pointer, in the mating pool. Because the circumference of the wheel is marked according to a string's fitness, this roulette-wheel mechanism is expected to make $f_i/\overline{f}$ copies of the ith string, where $\overline{f}$ is the average fitness of the population. This version of roulette-wheel selection is somewhat noisy: other more stable versions exist in the literature (10). As discussed later, the proportional selection scheme is inherently slow. One fix is to use a ranking selection scheme (45). All N strings in a population are first ranked according to the ascending order of string fitness. Then each string is assigned a rank from 1 (worst) to N (best) and proportional selection is used with rank values. This eliminates the functional dependency in the performance of the proportional selection. Recently, the tournament selection scheme is becoming popular because of its simplicity and controlled takeover property (45). In its simplest form, two strings are chosen at random for a tournament, and the better of the two is selected according to the string's fitness value. If done systematically, the best string in a population gets exactly two copies in the mating pool.

The crossover operator is applied next to the strings of the mating pool. Like the reproduction operator, a number of crossover operators exist in the GA literature (46,47), but in almost all crossover operators, two strings are picked from the mating pool at random, and some portion of the strings is exchanged between the strings. A single-point crossover operator was described earlier. It is intuitive from the construction that good substrings from either parent string can be combined to form a better child string if an appropriate site is chosen. Because the knowledge of an appropriate site is usually not known, a random site is usually chosen. With a random site, the children strings produced may or may not have a combination of good substrings from parent strings depending on whether or not the crossing site falls in an appropriate place. But we do not worry about this aspect too much, because, if good strings are created by crossover, there are more copies of them in the next mating pool generated by the reproduction operator. But if good strings are not created by crossover, they do not survive beyond the next generation, because reproduction usually does not select bad strings for the next mating pool. In a *two-point* crossover operator, two random sites are chosen and the contents bracketed by these sites are exchanged between two parents. This idea can be extended to create a multi-point crossover operator, and the extreme of this extension is what is known as a *uniform* crossover operator (47). In a uniform crossover for binary strings, each bit from either parent is selected with a probability of 0.5. It is worthwhile to note that the purpose of the crossover operator is two-fold. The main purpose of the crossover operator is to search the parameter space. The other aspect is that the search needs to be performed to preserve maximally the information stored in the parent strings, because these parent strings are instances of good strings selected by the reproduction operator. In the single-point crossover operator, the search is not extensive, but the maximum information is preserved from parent to children. On the other hand, in the uniform crossover, the search is very extensive but minimum information is preserved between parent and children strings. Even though some studies to find an optimal crossover operator exist (46), considerable doubts prevail whether those results can be generalized for all problems. Before any results from theoretical studies are obtained, the choice of the crossover operator is still a matter of personal preference. To preserve some of the previously found good strings, not all strings in the population are entered into the crossover operation. If a crossover probability of p_c is used, then $100p_c\%$ strings in the population are used in the crossover operation and $100(1 - p_c)\%$ of the population is simply copied to the new population. Even though the best $100(1 - p_c)\%$ of the current population is copied deterministically to the new population, this is usually performed stochastically.

The crossover operator is mainly responsible for the search aspect of genetic algorithms, even though the mutation operator is also used sparingly for this purpose. The mutation operator changes a `1` to a `0` and vice versa with a small mutational probability p_m. Mutation is needed to maintain diversity in the population. For example, if all strings in the population have a value `0` in a particular position along the string length and a `1` is needed in that position to obtain the optimum solution, then neither the reproduction nor crossover operator described previously can create a `1` in that position. The inclusion of mutation introduces some probability of turning that `0` into a `1`. Furthermore, mutation is useful for the local improvement of a solution.

These three operators are simple and straightforward. The reproduction operator selects good strings and the crossover operator recombines good substrings from two good strings, hopefully to form a better substring. The mutation operator alters a string locally to create a better string, hopefully. Even though none of these claims are guaranteed and/or tested while creating a new population of strings, it is expected that, if bad strings are created, they are eliminated by the reproduction operator in the next generation, and, if good strings are created, they are emphasized. Interestingly, biological and natural evolution are believed to be based on this principle. GAs are search algorithms designed to work along this principle of natural evolution (1). Later, we discuss some intuitive reasoning as to why GAs with these simple operators constitute potential search algorithms.

A Simple Example

To illustrate the working of GA operators, we consider a simple sinusoidal function which is to be maximized in a given interval:

$$\begin{aligned} \text{Maximize} \quad & \sin(x) \\ \text{Variable bound} \quad & 0 \le x \le \pi \end{aligned} \tag{2}$$

For illustrative purposes, we use 5-bit binary strings to represent the variable x, so that there are only 2^5 or 32 strings in the search space. We use a linear mapping between the decoded value of any string s and the bounds on the variable x: $x = \pi/31$ decode(s), where decode(s) is the decoded value of the string, s. The decoded value of a string s of length ℓ is calculated as $\sum_{i=0}^{\ell-1} 2^i s_i$, where $s_i \in$ `(0,1)` and the string s is represented as $(s_{\ell-1}s_{\ell-2} \ldots s_2s_1s_0)$. For example, the five-bit string `(0 1 0 1 1)` has a decoded value equal to 2^0`(1)` + 2^1`(1)` + 2^2`(0)` + 2^3`(1)` + 2^4`(0)` or 11. Thus, with this mapping, the string `(0 0 0 0 0)` represents the solution $x = 0$, and the string `(1 1 1 1 1)` represents the solution $x = \pi$. Let us also assume that we use a population of size four, proportional selection, single-point crossover with probability 1, and bitwise mutation with a probability 0.01. To start the GA simulation, we create a random initial population, evaluate each string, and use three GA operators as shown in Table 1. All strings are created at random. The first string has a decoded value equal to 9, and, after mapping this value in the variable range, the following value of $x = 0.912$, which corresponds to a functional value equal to $\sin(0.912) = 0.791$. Similarly, the

Table 1. One Generation of a GA Simulation on Function Sin(x)

Initial Population								New Population			
String	DV[a]	x	$f(x)$	f_i/f	AC[b]	Mating Pool	CS[c]	String	DV	x	$f(x)$
01001	9	0.912	0.791	1.39	1	01001	3	01000	8	0.811	0.725
10100	20	2.027	0.898	1.58	2	10100	3	10101	21	2.128	0.849
00001	1	0.101	0.101	0.18	0	10100	2	11100	28	2.838	0.299
11010	26	2.635	0.485	0.85	1	11010	2	10010	18	1.824	0.968
	Average, f		0.569						Average, f		0.711

[a] DV stands for decoded value of the string.
[b] AC stands for actual count of strings in the population.
[c] CS stands for cross site.

other three strings are also evaluated. Because the proportional reproduction scheme assigns a number of copies according to a string's fitness, the expected number of copies for each string is calculated in the fifth column. When a roulette-wheel selection scheme is actually implemented, the number of copies allocated to the strings is shown in the sixth column. The seventh column shows the mating pool. It is noteworthy that the third string in the initial population has a very small fitness compared with the average fitness of the population and thus has been eliminated by the selection operator. On the other hand, the second string, being a potential string, made two copies in the mating pool. Crossover sites are chosen at random, and the four new strings created after crossover are shown in the ninth column. Because a small mutational probability is considered, none of the bits are altered. Thus, the ninth column represents the new population. Thereafter, each of these stings is decoded, mapped, and evaluated. This completes one cycle of GA simulation. The average fitness of the new population is 0.711, an improvement from that in the initial population. It is interesting to note that, even though string selection and all string operations are performed using random numbers, the average performance of the population increases because of the application of all three GA operators. This is not a magic. GA operators are designed to have a directed search toward good regions but with some randomness in their actions so as to make GAs flexible enough not to get stuck locally at best solutions.

Every good optimization method needs to balance the extent of exploration of the information obtained up to the current time with the extent of exploitation of the search space required to obtain new and better point(s) (10,45). If the solutions obtained are exploited too much, premature convergence is expected. On the other hand, if too much stress is placed on search, the information obtained thus far has not been used properly. Therefore, the solution time may be enormous and the search is similar to a random search method. Most traditional methods have fixed transition rules and hence have fixed amounts of exploration and exploitation considerations. In contrast, we see later that the exploitation and exploration aspects of GAs are controlled almost independently. This provides enormous flexibility in applying GAs to solve optimization problems.

GAs for Multivariable Optimization

To handle multiple variables, GAs use a string of concatenating multiple substrings, each coding a separate variable. For example, if three variables x_1, x_2, and x_3 are coded in 3-, 5-, and 4-bit substrings, a complete string is a (3 + 5 + 4) or 12-bit string as follows:

$$\underbrace{011}_{x_1}\ \underbrace{01001}_{x_2}\ \underbrace{1001}_{x_3}$$

Once such 12-bit strings are created at random in the initial population, the corresponding values of x_1, x_2, and x_3 are determined by knowing the lower and upper bounds ($x_i^{\min}$, $x_i^{\max}$) and substring length (ℓ_i) of each variable i:

$$x_i = x_i^{\min} + \frac{x_i^{\max} - x_i^{\min}}{2^{\ell_i} - 1}\ \mathrm{decode}(s_i)$$

Now the string is evaluated by knowing the x_i values. The reproduction operator works as usual. The crossover and mutation operators are usually applied to the complete string. Although some studies exist where crossover is performed in each substring separately, such a strategy may be too destructive, resulting in a random search.

GAs for Constrained Optimization

Genetic algorithms have also been used to solve constrained optimization problems. Although different methods to handle constraints have been suggested, the *penalty function* method has been used mostly (10,23,48). In the penalty function method, a penalty term corresponding to the constraint violation is added to the objective function. In most cases, a bracket operator penalty term $\langle\alpha\rangle$ (=α, if α is negative; zero otherwise) is used. In a constrained minimization nonlinear programming (NLP) problem

$$\left.\begin{array}{lll} & \text{Minimize } f(x) & \\ \text{subject to} & & \\ & g(x) \ge 0, & j = 1, 2, \ldots, J; \\ & h_k(x) = 0, & k = 1, 2, \ldots, K; \\ & x_i^{(L)} \le x_i \le x_i^{(U)}, & i = 1, 2, \ldots, N. \end{array}\right\} \quad (3)$$

the objective function $f(x)$ is replaced by the unconstrained penalized function:

$$P(x) = f(x) + \sum_{j=1}^{J} u_j \langle g_j(x)\rangle^2 + \sum_{k=1}^{K} v_k [h_k(x)]^2 \quad (4)$$

where u_j and v_k are penalty coefficients, which are usually constant throughout the GA simulation. In the traditional penalty function method, the penalty parameters u_j and v_k are gradually increased from small initial values. This is done to avoid convergence to a suboptimal solution, a phenomenon which usually is caused by the formation of a distorted penalized function as the penalty parameters increase (23). Because GAs handle distorted or multimodal functions better than the traditional methods, a fixed value of penalty parameters is usually adequate.

To illustrate the working of GAs on a two-variable constrained optimization problem, we consider the following constrained problem:

$$\text{Minimize } f(x) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$$

subject to

$$g_1(x) = 26 - (x_1 - 5)^2 - x_2^2 \geq 0$$
$$x_1, x_2 \geq 0$$

With a population of 30 points, a crossover probability of 0.9 and a mutation probability of 0.01, we perform a GA simulation for 30 generations with a penalty parameter $u_1 = 100$. Figure 1 shows the initial population (empty boxes) and the population at generation 30 (empty circles) on the contour plot of the NLP problem. The figure shows that initial population is fairly spread out on the search space. After 30 generations, the complete population is in the feasible region and is placed close to the true optimum point.

Recently, a number of other penalty and nonpenalty function methods used in the context of GAs have been evaluated (48). Although many sophisticated methods have been proposed, it is concluded in that study that the previews simple strategy is most successful as a generic technique for handling constraints in GAs.

Why GAs Work

The working principle described previously is simple, and GA operators involve string copying and substring exchange and occasional alteration of bits. It is surprising that, with any such simple operators and mechanisms, any potential search is possible. We present intuitive reasoning as to why GAs work and remind the reader that a number of studies are currently underway to find a rigorous mathematical convergence proof for GAs (18,20,28,30,49,50). Even though the operations are simple, GAs are highly nonlinear, massively multifaceted, stochastic, and complex. There have been some studies using Markov chain analysis that involves deriving transitional probabilities from one state to another and manipulating them to find the convergence time and solution.

To investigate why GAs work, let us reconsider the one-cycle GA application to the function sin(x). The string copying and substring exchange are all very interesting and improve the average performance of a population, but let us investigate what has been processed in one cycle of GA operators. It is interesting to observe from Table 1 that there are some similarities in different string positions in the population of multiple strings. By applying three GA operators, we observe that the number of strings with such similarities has either increased or decreased. These similarities are called *schema* (schemata, in plural) in the GA literature. More specifically, a schema represents a set of strings with similarities at certain string positions. To formalize the concept, a schema for binary codings is represented by a triplet (`1, 0,` and $*$). A $*$ represents both `1` or `0`. Thus a schema H_1 = (`1 0 * * *`) represents eight strings with a `1` in the first position and a `0` in the second position. From Table 1, we observe that there is only one string contained in this schema in the initial population and there are two strings contained in this schema in the new population. On the other hand, even though there was one representative string of the schema H_2 = (`0 0 * * *`) in the initial population, there is none in the new population. There could be a number of other schemata that we may investigate and conclude whether or not the number of strings they represent is increased from the initial population to the new population. But what do these schemata mean anyway?

Because a schema represents certain similar strings, a schema can represent a certain region in the search space. For the previous function, the schema H_1 represents strings with x values varying from 1.621 to 2.331 and with function values varying from 0.999 to 0.725. On the other hand, the schema H_2 represents strings with x values varying from 0.0 to 0.709 and function values varying from 0.0 to 0.651. Because our objective is to maximize the function, we would like to have more copies of strings representing schema H_1 than H_2. This is what we have accomplished in Table 1 without having to count all of these schema competitions, without the knowledge of the complete search space, and by manipulating only a few instances of the search space. The schema H_1 for the previous example has only two defined positions (the first two bits), and both defined bits are tightly spaced (very close to each other) and contain the possible near-optimal solution [the string (`1 0 0 0 0`) is the optimal string in this problem]. The short and above-average schemata are known as *building blocks.* Although GA operators are applied on a population of strings, a number of such building blocks in various parts along the string (like H_1 in the previous example) are emphasized. Finally, such small building blocks are combined by the combined action of GA operators to form bigger and better building blocks and finally converge to the optimal solution. To avoid discussions on rigorous convergence proofs,

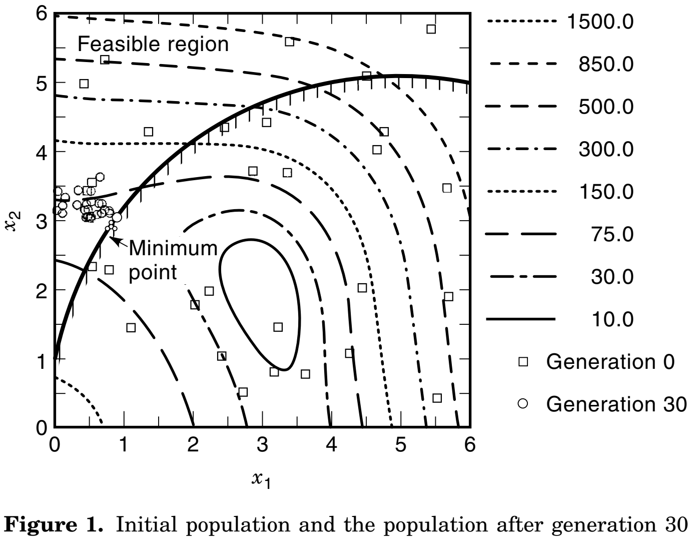

Figure 1. Initial population and the population after generation 30 shown on a contour plot of the NLP problem.

this is what can be hypothesized as the reason for GA's success. This hypothesis is largely known as the *Building Block Hypothesis.*

GA GUIDELINES

The building block hypothesis gives intuitive and qualitative reasoning as to what makes GAs work. But it tells nothing about what values of various GA parameters make GA work or not work. In this section, we present some guidelines for successfully applying GAs. It is important to note that the key insight in Holland's discovery of genetic algorithms is the processing of building blocks under genetic operators. It is an established fact (albeit with some contradictions) that genetic algorithms work by processing building blocks. Therefore, adequate supply, growth, and mixing of building blocks are essential features for a successful GA. Goldberg et al. (51) categorized these factors as follows:

1. GAs process building blocks. Therefore, a clear understanding of the underlying building blocks in any given search and optimization problem is needed. The knowledge of building blocks in a problem assists in designing proper coding for GA simulation.
2. Adequate supply of building blocks (either initially or temporally) must be ensured.
3. The population must be large enough to allow building block competition to occur.
4. The reproduction operator must be designed to allow adequate growth of building blocks in successive generations.
5. The search operators (crossover, mutation, and others) must be designed to allow proper mixing and combination of building blocks in successive generations.

Because these issues are important for understanding a GA, we discuss them in detail.

Building Block Processing

In most GA applications, the design or decision variables are coded in some string structures. Although binary-coded strings are used mostly, gray-coded strings and strings with alphabets of higher cardinality have also been used (52). In multivariable optimization problems, the substrings corresponding to each variable are joined to form a complete string. An obvious question arises in this coding: Which variables should be coded contiguously? It turns out that, if a random ordering of the variables is coded side-by-side, the action of a single-point crossover destroys good building-block combinations. Thus, a coding that helps successful propagation of the building blocks must be used (27). It is shown elsewhere (53) that when GAs are used to solve a problem with three different codings, the best performance is observed for that coding which respects the building-block propagation the most.

Adequate Supply of Building Blocks

If there are insufficient building blocks in a population, GAs do process them to finally form the true optimum solution. The supply of building blocks is provided both initially and/or temporally. The temporal supply of building blocks depends on the genetic operators and is discussed later (when we discuss the growth and mixing issues). However, the initial supply is provided by choosing a statistically large initial random population or by using a biased initial population. The former approach for building-block supply requires a large initial population and is discussed in the next subsection. The latter approach of building-block supply requires a comparatively smaller initial population, but demands some knowledge of good regions in the search space. Because there is some intuitive knowledge of the locations of the optima in many engineering optimization problems, this approach is often used in large-scale optimization problems. A 128 × 128 binary optical filter design problem was solved with a small initial population size but with a population biased toward a good region in the search space (42). A signal-to-noise (SNR) ratio was used as a fitness measure of a filter. When a random initial population was used, a population of 1,000 random filters produced a filter with an SNR of 9.3 after 200 generations. Whereas, when a biased population of the same size was used, a filter with an SNR of 310.0 was created after 200 generations. Because the initial population contains many building blocks in the latter case, the GA is able to combine the building blocks to form a near-optimal solution.

Adequate Population Size

Population size is an important parameter in the successful application of GAs. Although some empirical guidelines exist for choosing a population size ($1.65 \times 2^{0.21\ell}$, where ℓ is the string length (54)), ideally the population size must depend on the difficulty (nonlinearity, multimodality, and others) involved in the objective function. If a random initial population is used, the population size must be adequate to allow GAs to extract the required building blocks (of necessary size) to solve the problem. For example, if the maximum nonlinearity exists with any k bits in an ℓ-bit problem (that is, no more than k bits are related to each other in a nonlinear manner), the population size must be large enough to have all 2^k combinations of substrings in the initial random population. Because the nonlinearity in the subproblems could be so severe that best k-bit combination is difficult to generate by genetic operations. To solve the problem, the best k-bit substring must be in the initial population, and the initial population must have at least a few copies of each k-bit building block in the initial random population. Thinking along these lines and keeping in mind the signal (d) to be detected in a problem with variance (σ^2), we have devised the following expression for population size for binary-coded genetic algorithms (51):

$$n = 2c2^k(m-1)\frac{\sigma^2}{d^2} \tag{5}$$

where k is the order of nonlinearity in the problem and m is the total number of necessary building blocks. The parameter c is a statistical parameter. The original study details how this sizing equation was developed and how this equation is used to size populations (51). This sizing equation suggests that, for problems of fixed degree of nonlinearity (k is fixed), the population size is proportional to the squared noise-to-signal ratio (σ^2/d^2) in building blocks, and in problems with the number of necessary building blocks proportional to string length, the population sizing is of the order $O(\ell)$.

To demonstrate the use of the equation for population sizing, we discuss the results of one test problem. A 30-bit problem was constructed by concatenating five copies of a six-bit, 22-peaked, difficult subfunction (55). The 30-bit function has a total of 22^5 or 5,153,632 optima, of which 32 are global. The purpose of this study was to find one of the 32 global optima. This is a very difficult problem for any search and optimization method. When the population was sized according to the previous equation (by finding the signal and variance of the subfunction), it was found that a population of size 391 is adequate for solving the problem to global optimality. Simulation results of several GA runs suggest that GAs with a population of size of 300 sometimes cannot find the global optimum string (in those cases the GA got stuck to one of other 5,153,600 other suboptima), whereas a population size of 400 finds the global optimum in all runs.

Adequate Growth of Building Blocks

One of the necessary conditions for a successful GA run is that, under genetic operators, building blocks must multiply in each iteration. Even though the initial population contains the necessary building blocks, if a weak reproduction operator is used, the building blocks do not grow adequately or, if a weak recombination operator is used, the building blocks do not have the scope to combine before the population prematurely terminates to a suboptimal solution. The parameter responsible for faster growth of building blocks, called the *selection pressure,* is loosely quantified as the number of copies the best string gets during the reproduction operation (45). The amount of selection pressure in a reproduction operator is an inherent characteristic of the operator. If a reproduction operator with a small selection pressure is used, the growth of the building blocks is hampered. Thus, one of the criteria for choosing a reproduction operator is its selection pressure. In a study (45), the selection pressure for a number of selection schemes was calculated (Table 2). The table shows that the selection pressure in tournament and ranking selection are fixed at every generation and, therefore, controlled experiment can be performed with these operators. The *takeover time* is a measure of how fast the best solution in the initial population overpopulates the population with the reproduction operator alone. The table shows that ranking and tournament selection have a better takeover property than that of the proportional selection. The comparison of the *time complexity* of the three operators suggests that tournament selection requires minimal computational time.

Adequate Mixing of Building Blocks

As mentioned above, the choice of a selection operator depends on the choice of the recombination operator. As the individual building blocks are grown adequately by the selection operator, they also must combine to form bigger and better building blocks. Recently, a *control map* was found for values of the selection pressure s (the number of copies allocated to the best string in the population) versus the crossover probability p_c (the extent of search) for bitwise linear problems using a computational model that equates the degree of the characteristic time of convergence of the selection and the crossover operators alone (56). The analysis showed that for GAs to work even on simple bitwise linear problems, the following inequality among GA parameters must be satisfied:

$$p_c \geq \frac{e^{\ell/N}}{N \log N} \log s \tag{6}$$

where N is the population size. The essence of this relationship is that, if a selection operator with a large selection pressure is used, a crossover operator with more search power must be used. Simulation results on some test problems agree with this theoretical prediction. Similar studies were also performed to find control maps for nonlinear problems (57).

Proper understanding of these functionally decomposed models of GA dynamics provides better insights into the working of complex processing of schemata under GA operators. Knowing more about these pieces of the GA puzzle will help users to properly choose GA parameters.

In the next section, we present some advanced GA techniques which are increasingly applied in many fields, particularly in engineering.

REAL-CODED GAs

Because binary-coded GAs use a coding of variables, they work on discrete search spaces. In dealing with a continuous search space, a binary-coded GA converts it into a discrete set of points. Thus, to obtain the optimum point with desired accuracy, strings of sufficient length must be chosen. GAs have also been developed to work directly with continuous variables (instead of discrete variables). In those GAs, binary strings are not used; instead, the variables are used directly. Once a population of a random set of solutions is created, a reproduction operator is used to select good strings in the population. To create new strings, however, the crossover and mutation operators described earlier cannot be used efficiently. Even though the simple single-point crossover is used directly on the variable vector by forcing the cross-sites to fall only on the variable boundaries, the search is obviously not adequate. With such a crossover operator, the success of the search process mainly depends on the mutation operator. This type of GA has been used in earlier studies (58). Recently, new and efficient crossover operators have been designed, so that a search on an individual variable vector is also allowed. Let us consider that $x_i^{(j)}$ and $x_i^{(k)}$ are values of

Table 2. A Comparison of Three Selection Schemes[a]

Scheme	Selection Pressure	Take-Over Time	Time Complexity
Proportionate	$f_{\max}/f_{\text{avg}}$	$O(n \ln n)$	$O(n \ln n)$
Linear ranking	2 (usually)	$O(\ln n)$	$O(n \ln n)$
Binary tournament	2	$O(\ln n)$	$O(n)$

[a] The parameter n is the population size.

design variables x_i in two parent strings j and k. The crossover between these two values produces the following new value:

$$x_i^{\text{new}} = (1-\lambda)x_i^{(j)} + \lambda x_i^{(k)}, \quad 0 \le \lambda \le 1 \tag{7}$$

The parameter λ is a random number between zero and one. This equation calculates a new value bracketing $x_i^{(j)}$ and $x_i^{(k)}$. The calculation is performed for all variables in the vector. This crossover has a uniform probability of creating a point inside the region bounded by two parents. An extension to this crossover also creates points outside the range bounded by the parents. Eshelman and Schaffer (59) have suggested a blend crossover operator (BLX-α), in which a new point is created uniformly at random from a larger range extending an amount $\alpha|x_j^{(i)} - x_j^{(k)}|$ on either side of the region bounded by the two parents. The crossover operation depicted in Eq. (7) is also used to perform BLX-α by varying λ in the range ($-\alpha$, $1 + \alpha$). In a number of test problems, Eshelman and Schaffer observed that $\alpha = 0.5$ provides good results. One interesting feature of this type of crossover operator is that the created point depends on the location of both parents. If both parents are close to each other, the new point is also close to the parents. On the other hand, if the parents are far from each other, the search is more like a random search. The random search feature of these crossover operators is relaxed by using a distribution other than the uniform distribution between the parents. A recent study shows that, using a polynomial probability distribution with a bias towards near-parent points, performance is better than (BLX-0.5) in a number of test problems (60). Moreover, this crossover operator has a *search power* (27) similar to that in a single-point crossover. For two parent points, two children points ($c_i^{(j)}$ and $c_i^{(k)}$) are created using the following probability distribution:

$$P(\beta) = \begin{cases} 0.5(\eta+1)\beta^{\eta} & \text{if } 0 \le \beta \le 1 \\ 0.5(\eta+1)/\beta^{\eta+2} & \text{if } \beta > 1 \end{cases} \tag{8}$$

The parameter β is a spread factor defined as

$$\beta = \left| \frac{c_i^{(j)} - c_i^{(k)}}{x_i^{(j)} - x_i^{(k)}} \right| \tag{9}$$

The previous distribution allows creating near-parent points with comparatively larger probability than points far away. The parameter η is a distribution index that controls the extent of search of the operator. The probability of creating a far-away point for a small η is comparatively larger than that for a large η. This parameter is similar to the inverse of the *temperature* parameter used in simulated annealing studies (61). Ideally, a GA should be started with a small η so that almost any point can be created in the search space, thereby spreading the search well. Once a good search region is found, larger values of η must be used to concentrate the search in the region found. However, in most GA simulations a constant η of 2 is found satisfactory. A real-coded mutation operator is also created on the basis of a similar principle.

Based on the success of binary-coded GAs and real-coded GAs, a combined optimization algorithm was recently proposed for solving mixed-integral programming problems often encountered in engineering design (62). The algorithm uses flexible coding which codes integral or discrete variables by binary strings and codes continuous variables directly. Although the reproduction is used directly, crossover and mutation operators are applied according to the variable type. If a integral or discrete variable is to be crossed, a single-point crossover is used. Otherwise the real-coded crossover previously discussed is used. On several mechanical engineering design problems, this combined algorithm outperformed many traditional optimization algorithms (62).

HYBRID GAs

The generic GA operators previously discussed sometimes take a large number of function evaluations to converge to the exact optimum solution. A hill-climbing strategy is often started from the solution found by GAs to improve the solution locally. The implementation of such a hybrid GA is achieved in two ways:

1. A GA is applied first beginning from an initial random population. Thereafter, a traditional search and optimization algorithm (a steepest descent method (23) or a heuristic method (40)) is used from the best solution found from the GA. The difficulty with traditional methods is that their search depends on the initial solution. In the case of complex, nonlinear, and multimodal problems, traditional methods often stick at a suboptimal solution. However, if the initial solution lies in the global basin (a region where the problem is unimodal and the global optimal solution is the only optimum solution), traditional methods are the quickest of all optimization algorithms to converge to the optimal solution. Because the GA is likely to find a near-optimal solution even in multimodal problems, a traditional method begins its search from a near-optimal solution and helps converge to the true global solution quickly. Researchers have found such a hybrid algorithm useful in many engineering design problems (41).
2. The GA previously described is modified by using some problem-specific information. For example, instead of beginning a GA from a random population, a biased initial population is used. If information about a good search region is available, the initial population is formed around that region. In a complex, binary optical filter design problem (42), it was observed that the performance of a binary-coded GA improves significantly if the population is initialized by perturbing a binary version of the *matched* filter obtained with the fundamental principles. Problem information is also used to modify the GA operators, so that feasible and good solutions are always created. In a robot path planning problem (43), the crossover operator is allowed only between points closer to each other. In solving a traveling salesman problem using GAs (44), the distances between two cities are used to find suitable crossover points.

Although many other implementations exist, a hybrid study, either a GA or a bit hill-climbing method chosen on the fly depending on its success in previous iterations, is promising (63).

EVOLUTION STRATEGY

Although genetic algorithms were developed mostly in the United States, evolution strategy (ES) was independently developed in Germany. Because ES and GAs are both evolutionary algorithms, there are more similarities between them than discrepancies. A population approach was not used in early works of ES (64). The method [now known as (1 + 1)-ES] begins from a point and creates a new neighboring point using a Gaussian probability distribution with its mean at the current point and a prespecified fixed variance. This operation is similar to a combined reproduction and mutation operation. If the new point is better than the current point, the new point is chosen, and the procedure is continued. In a later version of this method, the variance is varied by the one-fifth rule, which states the following (65). If at least one better point is created in n consecutive iterations, the variance is reduced. Otherwise, if no new better point is created in n consecutive iterations, the variance is increased. Otherwise, the variance is kept unchanged. Realizing the advantage of working with a population of points instead of a single point, ES researchers devised two different algorithms, $(\mu + \lambda)$-ES and (μ,λ)-ES. In the former, the algorithm begins with μ parent points, and λ children points are created using the mutation operator described earlier. Thereafter, the best μ points from a combined population of μ parent and λ children points are chosen as the parent points for the next iteration. In the latter method, μ points are chosen only from λ children points. ES with this selection operator and with a naive crossover operator and the previously mentioned adaptive mutation operator, has successfully solved many numerical and engineering design problems (65).

GENETIC PROGRAMMING

An algorithm similar to a genetic algorithm is being used to find optimal LISP (a computer programming language) programs for solving different tasks (66). Genetic programming (GP) begins with a population of random LISP programs to solve a task. For each problem, a set of fundamental functions (+, −, *, /, sine, cosine, exponential, etc.) and terminals (numerics or variables) are chosen to create programs. Then each program is tested with a number of prespecified input-output data. The fitness is measured as the number of test cases correctly solved by the LISP program. Based on these fitness values, the proportional reproduction operator is used. The crossover operator is applied by exchanging certain meaningful portions of the program (chosen at random) between two programs. The mutation operator is also applied by replacing a function or a terminal with other suitable functions or terminals. Recent applications use a number of other meta-operators which improve the performance of GP. The most interesting aspect of GP is that the same algorithm is used to solve many different problems by just changing the function and terminal set (66). The problems include a Boolean multiplexor problem, an artificial ant problem, symbolic differentiation and integration, optimal control problems, and others.

MULTIMODAL FUNCTION OPTIMIZATION

One advantage of a population-based search technique is that, if required, a number of different solutions can be captured simultaneously in the population. GAs have exploited this feature to find multiple optimal solutions simultaneously in multimodal, function optimization problems (67,68). Consider the five-peaked function shown in Fig. 2. Because all five maximum points have equal function values, one may be interested in finding all five optimum points (or as many as possible). The knowledge of multiple optimal points enables designers to adapt to different optimal solutions, as and when required. The population in a GA simulation is adaptively divided into separate subpopulations corresponding to each optimum point by using *sharing functions* (68). The procedure is briefly described in the following.

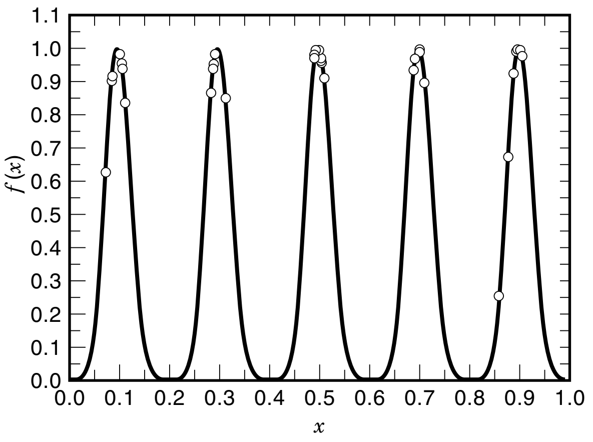

Figure 2. A function with five maximum points. A population of 50 points after 100 generations of a GA simulation with sharing functions shows that all optimum points are found.

For each solution (say, ith string) in the population, a distance measure d_{ij} is computed with each other solution (jth string) in the population. Thereafter, a sharing function value is computed with each d_{ij} value, as follows:

$$Sh(d_{ij}) = \begin{cases} 1 - \left(\dfrac{d_{ij}}{\sigma}\right)^{\alpha} & \text{if } d_{ij} \leq \sigma \\ 0 & \text{otherwise} \end{cases} \qquad (10)$$

The parameter σ indicates the maximum value of d_{ij} allowed to have a fitness sharing between two solutions. Guidelines are available to compute a reasonable σ for single and multivariable problems (67). The parameter α is an exponent and is usually set at 1. For each string, all $Sh(d_{ij})$ values are added to compute the niche count: $m_i = \sum_{j=1}^{N} Sh(d_{ij})$. The niche count of a string roughly estimates the number of solutions around the string. Thereafter, the fitness of the ith string is modified by dividing the fitness by the niche count: $f_i' = f_i/m_i$. Then the reproduction operator is performed with f_i' as the fitness. Because the fitness of a string is degraded by the number of solutions around it, the population maintains a stable subpopulation of many optimum solutions. If, for some reasons, one optimum contains many instances in the population, the fitness of each of these solutions is largely degraded compared with solutions in other optima, and a balance is restored. The complexity of niche count computation is reduced by using a random set of individuals (of size $\eta \ll N$) instead of the complete population. It is shown elsewhere (69) that $\eta = 0.1N$ is adequate for solving many multimodal problems.

Many researchers have used the sharing functions to solve multimodal problems. Here we present an application of sharing on a five-peaked function:

$$\text{Maximize } \sin^6(5\pi x) \qquad 0 \le x \le 1$$

GAs with an initial population of 50 random points have converged to a population shown in Fig. 2 after 100 generations, finding all five optimum points simultaneously.

MULTIOBJECTIVE FUNCTIONAL OPTIMIZATION

Many engineering design problems involve simultaneous solution of multiple objective functions. The most common problem which arises in engineering design is to minimize the overall cost of manufacturing and simultaneously minimize the associated accident rate (failures or rejections). Consider Fig. 3, where a number of plausible solutions to a hypothetical design problem are shown. Solution A costs less but is more accident-prone. Solution B, on the other hand, costs more, but is less prone to accidents. In their respective ways, both solutions are useful, but solution C is not as good as solution B in both objectives. It incurs more cost and more accidents. Thus, in multiobjective optimization problems a number of solutions exist (like solutions A, B, and D in the figure) which are optimum in some sense. These solutions constitute a Pareto-optimal front shown by the thick dashed line in the figure. Because, any point on this front is an optimum point, it is desirable to find as many such points as possible. Recently, a number of extensions to simple GAs have been tried to find many Pareto-optimal points simultaneously in various multiobjective optimization problems (69–71).

In the nondominated sorting GA approach (NSGA) adopted by Srinivas and Deb (70), all population members are divided into different nondominated fronts. Each solution (say, ith solution) in the population is compared with other solutions according to all objectives. If one solution is found, which is superior to the ith solution in all objectives, then the ith solution is tagged as a dominated solution. After all population members are compared with each other, the solutions without a dominated tag are considered nondominated solutions and are assumed to be the members of the first nondominated front. Thereafter, these solutions are temporarily counted out for determining further fronts, and a similar procedure is continued. After all population members are divided into nondomination levels, each solution in a front is assigned the same dummy fitness. The fronts from the first to last front are assigned a dummy fitness in descending order of magnitude. Finally, solutions in each front are shared, as described previously, to maintain a spread of solutions within the front. The reproduction operator is performed with the dummy fitness values. Crossover and mutation operators are used as usual.

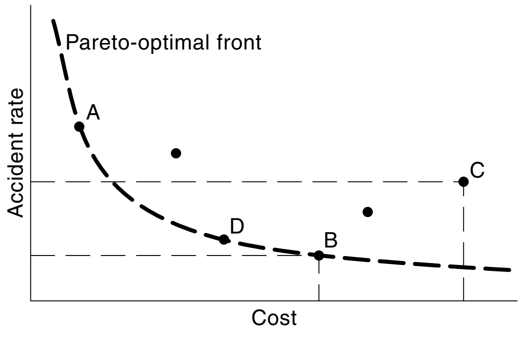

Figure 3. A typical two-objective problem. Solutions A, B, and D are Pareto-optimal solutions.

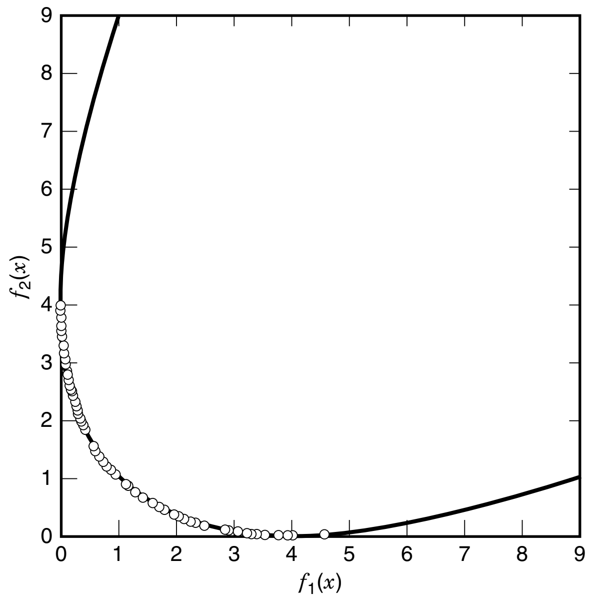

Figure 4. Population of 100 points at generation 100 shows that most of the Pareto-optimal solutions are found with NSGA.

Consider the following two objective functions for minimization:

$$f_1(x) = x^2$$
$$f_2(x) = (x-2)^2$$

in the interval $-1000 \le x \le 1000$. The Pareto-optimal front for this problem is the complete region $0 \le x \le 2$. With a random initial population of 100 points in the range $-1000 \le x \le 1000$, the modified GA converges to the range $0 \le x \le 2$ with a population distribution shown in Fig. 4 (70). The modified GA has also been successfully applied to other test functions and a truss-structure optimal design problem (69).

PARALLEL GAs

Many real-world engineering design problems require considerable amounts of computational time for evaluating objective functions. Because any search and optimization algorithm requires comparison of several designs, the overall computational time required to find an optimal or a near-optimal solution is often large. One way to alleviate this problem is to compute several designs or solutions in parallel. However, traditional methods use a point-by-point approach where one solution must first be evaluated before the next solution is determined. In other words, the traditional methods are serial in nature. Thus, there is a limitation in using the traditional methods on a parallel machine.

GAs, on the other hand, are easily implemented on a parallel machine. Population members are allocated equally to each processor and are evaluated simultaneously. If a tournament selection operator is used, two solutions are sent to each processor for comparison. Because the crossover operator requires only two solutions, it is also be performed in a distributed manner. Mutation requires modification in only one string at a time. Thus, GAs are ideal algorithms for use with parallel architectures. Parallel GAs are also implemented with a network model, in which a GA is run on each processor individually. A communication network between the processors is prespecified (72,73). While GAs are running on each processor, occasional migration of the best few solutions from each processor is made to other connecting processors. This allows parallel processing of the search space to find the optimal solution. Researchers have reported better performance (sometimes superlinear performance) with such implementations than with sequential GAs (74–78). More research is needed in this direction to find optimal network configurations and optimal frequency and number of solutions for migration.

GENETIC ALGORITHM-BASED MACHINE LEARNING (GBML)

Genetic algorithm is also used as a discovery mechanism in machine learning applications (10). In the classifier-based machine learning system, learning is achieved by discovery of a set of good rules (or classifiers) in an arbitrary environment. The essential components of a classifier-based GBML are the rule and message system, an apportionment of credit system, and a genetic algorithm, although a simplified version of this approach was proposed recently (79).

The rule and message system consists of a coding-decoding mechanism for converting environmental messages into string-like messages, a message list, and a set of classifiers. A classifier is an if-then rule specifying condition and action in ternary (`1`, `0`, and `#`) and binary alphabets (`1` and `0`), respectively. For example, a classifier `1##0:0001` matches a four-bit message `1100`, because the message matches the condition (`1##0`) of the classifier (a `#` matches both `1` and `0`). Because the classifier has matched the message, in turn, it *fires* the message `0001`.

The apportionment of credit system (Holland called it a *bucket brigade* algorithm) allocates appropriate strength to each classifier depending on the number of messages it fired. Every time a classifier is matched with a message, a portion of its strength is deducted and added to the classifier(s) which posted that message. Thus, repetitive application of the bucket brigade algorithm increases the strength of good classifiers and reduces the strength of bad classifiers. This and a few other auction and bidding mechanisms are performed by the bucket brigade algorithm.

The GBML algorithm works as follows. At the start of the GBML application, a set of classifiers is created randomly. All classifiers are assigned the same strength. Depending on the environmental message(s), the classifiers are matched and their messages are posted in the message list. Each of these messages, in turn, is checked with other classifiers and messages of matched classifiers are posted in the message list. This procedure is continued until a steady state is reached. The bucket brigade algorithm does not create a new classifier, it only updates the strengths of initially chosen classifiers. Genetic algorithms are used for this purpose. After a few iterations of the bucket brigade algorithm (when large enough iterations have passed to reach near the steady-state strength), a generation of GA is applied to create a few new classifiers. The GA used in GBML is similar to that described earlier, except for a few modifications. The reproduction operation is performed with the strength value of the classifiers. The crossover and mutation operations are modified to apply them on both the condition and action components of the classifier. In addition, each child classifier created by using the tripartite GA used to replace a parent classifier which maximally resembles the child classifier. This procedure and a few variations have been used to solve many machine learning problems (79–82).

BIBLIOGRAPHY

1. R. Dawkins, *River Out of Eden: A Darwinian View of Life,* London: Weidenfeld and Nicholson, 1995.
2. J. H. Holland, *Adaptation in Natural and Artificial Systems,* Ann Arbor, MI: University of Michigan Press, 1975.
3. J. D. Bagley, The behavior of adaptive systems which employ genetic and correlation algorithms (Doctoral dissertation), *Dissertation Abstracts International,* **28** (12): 5106B, (University Microfilms No. 68-7556), 1967.
4. R. B. Hollstien, Artificial genetic adaptation in computer control systems (Doctoral dissertation), *Dissertation Abstracts International,* **32** (3): 1510B, (University Microfilms No. 71-23,773), 1971.
5. A. D. Bethke, Genetic algorithms as function optimizers (Doctoral dissertation), *Dissertation Abstracts International,* **41** (9): 3503B, (University Microfilms No. 8106101), 1981.
6. L. B. Booker, Intelligent behavior as an adaptation to the task environment (Doctoral Dissertation), *Dissertation Abstracts International,* **43** (2): 469B, (University Microfilms No. 8214966), 1982.
7. D. E. Goldberg, Computer-aided gas pipeline operation using genetic algorithms and rule learning (Doctoral dissertation), *Dissertation Abstracts International,* **44** (10): 3174B, (University Microfilms No. 8402282), 1983.
8. K. A. De Jong, An analysis of the behavior of a class of genetic adaptive systems, (Doctoral dissertation). *Dissertation Abstracts International,* **36** (10): 5140B, 1975.
9. L. Davis (ed.), *Handbook of Genetic Algorithms,* New York: Van Nostrand Reinhold, 1991.
10. D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning,* Reading, MA: Addison-Wesley, 1989.
11. Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs,* New York: Springer Verlag, 1994.
12. M. Mitchell, *An Introduction to Genetic Algorithms,* Cambridge, MA: MIT Press, 1996.
13. J. J. Grefenstette (ed.), *Proceedings of An International Conference on Genetic Algorithms and Their Applications, Pittsburgh, PA, July 24–26, 1985,* Hillsdale, NJ: Lawrence Erlbaum Associates, 1985.
14. J. J. Grefenstette (ed.), *Proceedings of the 2nd International Conference on Genetic Algorithms, Cambridge, MA, July 28–31, 1987,* Hillsdale, NJ: Lawrence Erlbaum Associates, 1987.
15. J. D. Schaffer (ed.), *Proceedings of the 3rd International Conference on Genetic Algorithms, Washington, DC, June 4–7, 1989,* San Mateo, CA: Morgan Kaufmann, 1989.

16. R. K. Belew and L. B. Booker (eds.), *Proceedings of the 4th International Conference on Genetic Algorithms, July 13–16, 1991, San Diego,* San Mateo, CA: Morgan Kaufmann, 1991.

17. S. Forrest (ed.), *Proceedings of the 5th International Conference on Genetic Algorithms, Urbana, IL, July 17–21, 1993,* San Mateo, CA: Morgan Kaufmann, 1993.

18. L. Eshelman (ed.), *Proceedings of the 6th International Conference on Genetic Algorithms, Pittsburgh, PA, July 15–19, 1995,* San Mateo, CA: Morgan Kaufmann, 1995.

19. G. J. E. Rawlins (ed.), *Foundations of Genetic Algorithms,* San Mateo, CA: Morgan Kaufmann, 1991.

20. D. Whitley (ed.), *Foundations of Genetic Algorithms II,* San Mateo, CA: Morgan Kaufmann, 1992.

21. H.-P. Schwefel and R. Manner (eds.), *Parallel Problem Solving from Nature,* Berlin: Springer-Verlag, 1990.

22. R. Manner and B. Manderick (eds.), *Parallel Problem Solving from Nature, 2,* Amsterdam: North-Holland, 1992.

23. K. Deb, *Optimization for Engineering Design: Algorithms and Examples,* New Delhi: Prentice–Hall, 1995.

24. S. S. Rao, *Optimization Theory and Applications,* New Delhi: Wiley Eastern, 1984.

25. G. V. Reklaitis, A. Ravindran, and K. M. Ragsdell, *Engineering Optimization—Methods and Applications,* New York: John Wiley & Sons, 1983.

26. N. J. Radcliffe, Genetic set recombination, *Foundations of Genetic Algorithms,* **II**: 230–219, 1993.

27. N. J. Radcliffe, Formal analysis and random respectful recombination, *Proc. 4th Int. Conf. Genetic Algorithms,* San Diego, CA: July 13–16, 1991, pp. 222–229.

28. G. Rudolph, Convergence analysis of canonical genetic algorithms, *IEEE Trans. Neural Netw.,* **5**: 96–101, 1994.

29. M. D. Vose, Generalizing the notion of schema in genetic algorithms, *Artificial Intelligence,* **50**: 385–396, 1990.

30. M. D. Vose and G. E. Liepins, Punctuated equilibria in genetic search, *Complex Systems,* **5** (1): 31–44, 1991.

31. M. P. Fourman, Compaction of symbolic layout using genetic algorithms, *Proc. Int. Conf. Genetic Algorithms, Pittsburgh, PA, July 24–26,* 1985, pp. 141–153.

32. T. C. Fogarty, F. Vavak, and P. Cheng, Use of the genetic algorithm for load balancing of sugar beet presses, *Proc. 6th Int. Conf. Genetic Algorithms, Pittsburgh, PA, July 15–19,* 1995, pp. 617–624.

33. S. Coombs and L. Davis, Genetic algorithms and communication link speed design: Constraints and operators, *Proc. 2nd Int. Conf. Genetic Algorithms, Cambridge, MA, July 28–31,* 1987, pp. 257–260.

34. D. Whitley, S. Dominic, and R. Das, Genetic reinforcement learning with multilayer neural networks, *Proc. 4th Int. Conf. Genetic Algorithms, San Diego, CA, July 13–16,* 1991, pp. 562–569.

35. G. F. Miller, P. M. Todd, and S. U. Hegde, Designing neural networks using genetic algorithms, *Proc. 3rd Int. Conf. Genetic Algorithms, Washington DC, June 4–7,* 1989, pp. 379–384.

36. S. G. Romaniuk, Evolutionary grown semi-weighted neural networks, *Proc. 6th Int. Conf. Genetic Algorithms, Pittsburgh, PA, July 15–19,* 1995, pp. 444–451.

37. C. L. Karr, Design of an adaptive fuzzy logic controller using genetic algorithms, *Proc. 4th Int. Conf. Genetic Algorithms, San Diego, CA, July 13–16,* 1991, pp. 450–457.

38. C. Z. Janikow, A genetic algorithm for optimizing fuzzy decision trees, *Proc. 6th Int. Conf. Genetic Algorithms, Pittsburgh, PA, July 15–19,* 1995, pp. 421–428.

39. F. Herrera and J. L. Verdegay (eds.), *Genetic Algorithms and Soft Computing,* Heidelberg: Physica-Verlag, 1996.

40. D. J. Powell, S. S. Tong, and M. M. Skolnick, EnGENEous domain independent, machine learning for design optimization, *Proc. 3rd Int. Conf. Genetic Algorithms, Washington DC, June 4–7,* 1989, pp. 151–159.

41. D. H. Ackley, *A Connectionist Machine for Genetic Hillclimbing,* Boston, MA: Kluwer Academic, 1987.

42. K. Deb, Genetic algorithms in optimal optical filter design, *Proc. Int. Conf. Computing Congress, Hyderabad, India, Dec. 15–18,* 1993, pp. 29–36.

43. Y. Davidor, Analogous crossover, *Proc. 3rd Int. Conf. Genetic Algorithms, Washington DC, June 4–7,* 1989, pp. 98–103.

44. J. J. Grefenstette, R. Gopal, B. J. Rosamaita, and D. Van Gucht, Genetic algorithms for traveling salesman problem, *Proc. Int. Conf. Genetic Algorithms, Pittsburgh, PA, July 24–26,* 1985, pp. 160–168.

45. D. E. Goldberg and K. Deb, A comparison of selection schemes used in genetic algorithms, *Foundations of Genetic Algorithms,* 69–93, 1991.

46. W. M. Spears and K. A. De Jong, An analysis of multi-point crossover, *Foundations of Genetic Algorithms,* 310–315, 1991.

47. G. Syswerda, Uniform crossover in genetic algorithms, *Proc. 3rd Int. Conf. Genetic Algorithms, Washington DC, June 4–7,* 1989, pp. 2–9.

48. Z. Michalewicz and M. Schoenauer, Evolutionary algorithms for constrained parameter optimization problems, *Evolutionary Computation,* **4** (1): 1–32, 1996.

49. A. Nix and M. D. Vose, Modeling genetic algorithms with Markov chains, *Annals of Mathematics and Artificial Intelligence,* **5**: 79–88, 1992.

50. T. E. Davis and J. C. Principe, A simulated annealing-like convergence theory for the simple genetic algorithm, *Proc. 4th Int. Conf. Genetic Algorithms, San Diego, CA, July 13–16,* 1991, pp. 174–181.

51. D. E. Goldberg, K. Deb, and J. H. Clark, Genetic algorithms, noise, and the sizing of populations, *Complex Systems,* **6**: 333–362, 1992.

52. J. D. Schaffer, R. A. Caruana, L. Eshelman, and R. Das, A study of control parameters affecting online performance of genetic algorithms for function optimization, *Proc. 3rd Int. Conf. Genetic Algorithms, Washington DC, June 4–7,* 1989, pp. 51–60.

53. D. E. Goldberg, B. Korb, and K. Deb, Messy genetic algorithms: Motivation, analysis, and first results, *Complex Systems,* **3**: 493–530, 1989.

54. D. E. Goldberg, Optimal initial population size for binary-coded genetic algorithms (TCGA Report No. 85001). Tuscaloosa: University of Alabama, The Clearinghouse for Genetic Algorithms.

55. D. E. Goldberg, K. Deb, and J. Horn, Massive multimodality, deception, and genetic algorithms, *Parallel Problem Solving from Nature,* **II**: 37–46, 1992.

56. D. E. Goldberg, K. Deb, and D. Theirens, Toward a better understanding of mixing in genetic algorithms, *J. SICE,* **32** (1), 1991.

57. D. Thierens and D. E. Goldberg, Mixing in genetic algorithms, *Proc. 5th Int. Conf. Genetic Algorithms, Urbana, IL, July 17–21,* 1993, pp. 38–45.

58. A. Wright, Genetic algorithms for real parameter optimization, *Foundations of Genetic Algorithms,* 205–220, 1991.

59. L. Eshelman and J. D. Schaffer, Real-coded genetic algorithms and interval-schemata, *Foundations of Genetic Algorithms,* **II**: 187–202, 1993

60. K. Deb and R. B. Agrawal, Simulated binary crossover for continuous search space, *Complex Systems,* **9**: 115–148, 1995.

61. E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing,* Chichester: Wiley, 1989.

62. K. Deb, GeneAS: A Robust optimal design technique for mechanical component design. In *Evolutionary Algorithms in Engineering Applications,* New York: Springer-Verlag, 1997.

63. F. G. Lobo and D. E. Goldberg, Decision making in a hybrid genetic algorithm, *1997 IEEE International Conference on Evolutionary Computation, Indianapolis, April 13–16,* 1997, pp. 121–125.

64. I. Rechenberg, *Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution,* Stuttgart: Frommann–Holzboog Verlag, 1973.

65. T. Back, F. Hoffmeister, and H.-P. Schwefel, A survey of evolution strategies, *Proc. 4th Int. Conf. Genetic Algorithms, San Diego, CA, July 13–16,* 1991, pp. 2–9.

66. J. Koza, *Genetic Programming,* Cambridge, MA: MIT Press, 1992.

67. K. Deb and D. E. Goldberg, An investigation of niche and species formation in genetic function optimization, *Proc. 3rd Int. Conf. Genetic Algorithms, Washington DC, June 4–7,* 1989, pp. 42–50.

68. D. E. Goldberg and J. Richardson, Genetic algorithms with sharing for multimodal function optimization, *Proc. 2nd Int. Conf. Genetic Algorithms, Cambridge, MA, July 28–31,* 1987, pp. 41–49.

69. K. Deb and A. Kumar, Real-coded genetic algorithms with simulated binary crossover: Studies on multimodal and multiobjective problems, *Complex Systems,* **9**: 431–454, 1995.

70. N. Srinivas and K. Deb, Multiobjective function optimization using nondominated sorting genetic algorithms, *Evolutionary Computation,* **2** (3): 221–248, 1995.

71. C. M. Fonseca and P. J. Fleming, Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization, *Proc. 5th Int. Conf. Genetic Algorithms, Urbana, IL, July 17–21,* 1993, pp. 416–423.

72. H. Muhlenbein, M. Schomisch, and J. Born, The parallel genetic algorithm as function optimizer, *Parallel Computing,* **17**: 619–632, 1991.

73. R. Shonkwiler, Parallel genetic algorithms, *Proc. 5th Int. Conf. Genetic Algorithms, Urbana, IL, July 17–21,* 1993, pp. 199–205.

74. F. F. Easton and N. Mansour, A distributed genetic algorithm for employee staffing and scheduling problems, *Proc. 5th Int. Conf. Genetic Algorithms, Urbana, IL, July 17–21,* 1993, pp. 360–367.

75. V. Gordon and D. Whitley, Serial and parallel genetic algorithms as function optimizers, *Proc. 5th Int. Conf. Genetic Algorithms, Urbana, IL, July 17–21,* 1993, pp. 177–183.

76. M. Gorges-Schleuter, ASPARAGOS: An asynchronous parallel genetic optimization strategy, *Proc. 3rd Int. Conf. Genetic Algorithms, Washington DC, June 4–7,* 1989, pp. 422–427.

77. J. J. Grefenstette, Parallel adaptive algorithms for function optimization (Technical Report No. CS-81-19), Nashville: Vanderbilt University, Computer Science Department, 1981.

78. R. Shonkwiler, F. Mendivil, and A. Deliu, Genetic algorithms for the 1-D fractal inverse problem, *Proc. 4th Int. Conf. Genetic Algorithms, San Diego, CA, July 13–16,* 1991, pp. 495–501.

79. S. W. Wilson, ZCS: A zeroth level classifier system, *Evolutionary Computation,* **2** (1): 1–18, 1994.

80. T. A. Sedbrook, H. Wright, and R. Wright, Application of a genetic classifier for patient triage, *Proc. 4th Int. Conf. Genetic Algorithms, San Diego, CA, July 13–16,* 1991, pp. 334–338.

81. R. E. Smith and M. Valenzuela-Rendon, A study of rule set development in a learning classifier system, *Proc. 3rd Int. Conf. Genetic Algorithms, Washington DC, June 4–7,* 1989, pp. 340–346.

82. S. W. Wilson, Classifier system and the Animat problem, *Machine Learning,* **2** (3): 199–228, 1987.

KALYANMOY DEB
Indian Institute of Technology,
Kanpur

GENETIC ALGORITHMS. See COMPUTATIONAL INTELLIGENCE; DISPATCHING; TRAVELING SALESPERSON PROBLEMS.

GEOGRAPHICALLY DISTRIBUTED MACHINING. See TELECONTROL.