

Genetic Algorithm in Search and Optimization: The Technique and Applications

Kalyanmoy Deb

Department of Mechanical Engineering
Indian Institute of Technology, Kanpur
Kanpur, UP 208 016, INDIA
E-mail: deb@iitk.ernet.in

Abstract

A genetic algorithm (GA) is a search and optimization method developed by mimicking the evolutionary principles and chromosomal processing in natural genetics. A GA begins its search with a random set of solutions usually coded in binary string structures. Every solution is assigned a fitness which is directly related to the objective function of the search and optimization problem. Thereafter, the population of solutions is modified to a new population by applying three operators similar to natural genetic operators—reproduction, crossover, and mutation. A GA works iteratively by successively applying these three operators in each generation till a termination criterion is satisfied. Over the past one decade, GAs have been successfully applied to a wide variety of problems, because of their simplicity, global perspective, and inherent parallel processing. In this paper, we outline the working principle of a GA by describing these three operators and by outlining an intuitive sketch of why the GA is a useful search algorithm. Thereafter, we apply a GA to solve a complex engineering design problem. Finally, we discuss how GAs can enhance the performance of other soft computing techniques—fuzzy logic and neural network techniques.

1 Introduction

Classical search and optimization methods demonstrate a number of difficulties when faced with complex problems. The major difficulty arises when a one algorithm is applied to solve a number of different problems. This is because each classical method is designed to solve only a particular class of problems efficiently. Thus, these methods do not have the breadth to solve different types of problems often faced by designers and practitioners. Moreover, most classical methods do not have the global perspective and often get converged to a locally optimal solution. Another difficulty is their inability to be used in parallel computing environment efficiently. Since most classical algorithms are serial in nature, not much advantage (or speed-up) can be achieved with them.

Over the few years, a number of search and optimization algorithms, which are drastically different in principle from the classical methods, are getting increasingly more attention. These methods mimic a particular natural phenomenon to solve search and optimization problems. Genetic algorithms (Holland, 1975), simulated annealing (Kirkpatrick, Gelatt, and Vecchi, 1983) and tabu search (Glover and Laguna, 1997) are a few of these methods. In this paper, we describe genetic algorithms (commonly known as GAs), which mimic the evolutionary principles and chromosomal processing of natural genetics and natural selection.

The GA technique was first conceived by Professor John Holland of University of Michigan, Ann Arbor in 1965. His first book appeared in 1975 (Holland, 1975) and till 1985, GAs have been practiced mainly by Holland and his students (Bagley, 1967; Bethke, 1981; Cavicchio, 1971; De Jong, 1975; Goldberg, 1983). Exponentially more number of researchers and practitioners became interested in GAs soon after the first International conference on GAs held in 1985. Now, there exist a number of books (Goldberg, 1989; Mitchell, 1996) and a few journals dedicated to publishing research papers on the topic (including one from MIT Press and one from IEEE). Every year, there are at least 15-20 conferences and workshops being held on the topic at various parts of the globe. The major reason for GA's popularity in various search and optimization problems is its global perspective, wide spread applicability, and inherent parallelism.

In the remainder of the paper, we shall discuss the working principle of a GA and show how a GA can be applied to an engineering design problem. A brief discussion of how a GA can be used to improve the performance of other soft computing techniques is also highlighted.

2 Classical Search and Optimization Techniques

Traditional search and optimization methods can be classified into two distinct groups: Direct and gradient-based methods (Deb, 1995; Reklaitis, Ravindran, and Ragsdell, 1983). In direct methods, only objective function and constraints are used to guide the search strategy, whereas gradient-based methods use the first and/or second-order derivatives of the objective function and/or constraints to guide the search process. Since derivative information is not used, the direct search methods are usually slow, requiring many function evaluations for convergence. For the same reason, they can be applied to many problems without a major change of the algorithm. On the other hand, gradient-based methods quickly converge to an optimal solution, but are not efficient in non-differentiable or discontinuous problems. In addition, there are some common difficulties with most of the traditional direct and gradient-based techniques:

- Convergence to an optimal solution depends on the chosen initial solution.
- Most algorithms tend to get *stuck* to a suboptimal solution.
- An algorithm efficient in solving one search and optimization problem may not be efficient in solving a different problem.
- Algorithms are not efficient in handling problems having discrete variables.
- Algorithms cannot be efficiently used on a parallel machine.

Because of the nonlinearities and complex interactions among problem variables often exist in complex search and optimization problems, the search space may have many optimal solutions, of which most are locally optimal solutions having inferior objective function values. When solving these problems, if traditional methods get attracted to any of these locally optimal solutions, there is no escape from it.

Many traditional methods are designed to solve a specific type of search and optimization problems. For example, geometric programming (GP) method is designed to solve only posynomial-type objective function and constraints (Duffin, Peterson, and Zener, 1967). GP is efficient in solving such problems but can not be applied suitably to solve other types of functions. Conjugate direction method has a convergence proof for solving quadratic functions, but they are not expected to work well in problems having multiple optimal solutions. Frank-Wolfe method (Reklaitis, Ravindran, and Ragsdell, 1983) works efficiently on linear-like function and constraints, but the performance largely depends on

the chosen initial conditions. Thus, one algorithm may be best suited for one problem and may not be even applicable to a different problem. This requires designers to know a number of optimization algorithms.

In many search and optimization problems, problem variables are often restricted to take discrete values only. To solve such problems, an usual practice is to assume that the problem variables are real-valued. A classical method can then be applied to find a real-valued solution. To make this solution feasible, the nearest allowable discrete solution is chosen. But, there are a number of difficulties with this approach. Firstly, since many infeasible values of problem variables are allowed in the optimization process, the optimization algorithm is likely to take many function evaluations before converging, thereby making the search effort inefficient. Secondly, for each infeasible discrete variable, two values (the nearest lower and upper available sizes) are to be checked. For N discrete variables, a total of 2^N such additional solutions need to be evaluated. Thirdly, two options checked for each variable may not guarantee the optimal combination of all variables. All these difficulties can be eliminated if *only* feasible values of the variables are allowed during the optimization process.

Many search and optimization problems require use of a simulation software, involving finite element technique, computational fluid mechanics approach, solution of nonlinear equations, and others, to compute the objective function and constraints. The use of such softwares is time-consuming and may require several minutes to hours to evaluate one solution. Because of the availability of parallel computing machines, it becomes now convenient to use parallel machines in solving complex search and optimization problems. Since most traditional methods use point-by-point approach, where one solution gets updated to a new solution in one iteration, the advantage of parallel machines cannot be exploited.

The above discussion suggests that traditional methods are not good candidates for an efficient search and optimization algorithm. In the following section, we describe the genetic algorithm which works according to principles of natural genetics and evolution, and which has been demonstrated to solve various search and optimization problems.

3 Motivation from Nature

Most biologists believe that the main driving force behind the natural evolution is the Darwin's survival-of-the-fittest principle (Dawkins, 1976; Eldredge, 1989). In most situations, the nature ruthlessly follows two simple principles:

1. If by genetic processing an above-average offspring is created, it is going to survive longer than an average individual and thus have more opportunities to produce children having some of its traits than an average individual.
2. If, on the other hand, a below-average offspring is created, it does not survive longer and thus gets eliminated from the population.

The renowned biologists Richard Dawkins explains many evolutionary facts with the help of Darwin's survival-of-the-fittest principle in his seminal works (Dawkins, 1976; 1986). He argues that the tall trees that exist in the mountains were only a feet tall during early ages of evolution. By genetic processing if one tree had produced an offspring an inch taller than all other trees, that offspring enjoyed more sunlight and rain and attracted more insects for pollination than all other trees. With extra benefits, that lucky offspring had an increased life and more importantly had produced more offspring like it (with tall feature) than others. Soon enough, it occupies most of the mountain with trees having its genes and the competition for survival now continues with other trees, since the available resource (land) is limited. On the other hand, if a tree had produced an offspring with an

inch smaller than others, it was less fortunate to enjoy all the facilities other neighboring trees had enjoyed. Thus, that offspring could not survive longer. In a genetic algorithm, this feature of natural evolution is introduced through its operators.

The principle of emphasizing good solutions and deleting bad solutions is a nice feature a population-based approach should have. But one may wonder about the real connection between an optimization procedure and natural evolution! Has the natural evolutionary process tried to maximize a utility function of some sort? Truly speaking, one can imagine a number of such functions which the nature may be thriving to maximize: Life span of a species, quality of life of a species, physical growth, and others. However, any of these functions is nonstationary in nature and largely depends on the evolution of other related species. Thus, in essence, the nature has been really optimizing much more complicated objective functions by means of natural genetics and natural selection than search and optimization problems we are interested in solving. A genetic algorithm is an abstraction of the complex natural genetics and natural selection process. The simple version of a GA described in the following section aims to solve stationary search and optimization problems. Although a GA is a simple abstraction, it is robust and has been found to solve various search and optimization problems of science, engineering, and commerce.

4 Genetic Algorithm: The Technique

In this section, we first describe the working principle of a genetic algorithm. Thereafter, we shall show a simulation of a genetic algorithm for one iteration on a simple optimization problem. Later, we shall give intuitive reasoning of why a GA is a useful search and optimization procedure.

4.1 Working Principle

Genetic algorithm (GA) is an iterative optimization procedure. Instead of working with a single solution in each iteration, a GA works with a number of solutions (collectively known as a population) in each iteration. A flowchart of the working principle of a simple GA is shown in Figure 1. In the absence of any knowledge of the problem domain, a

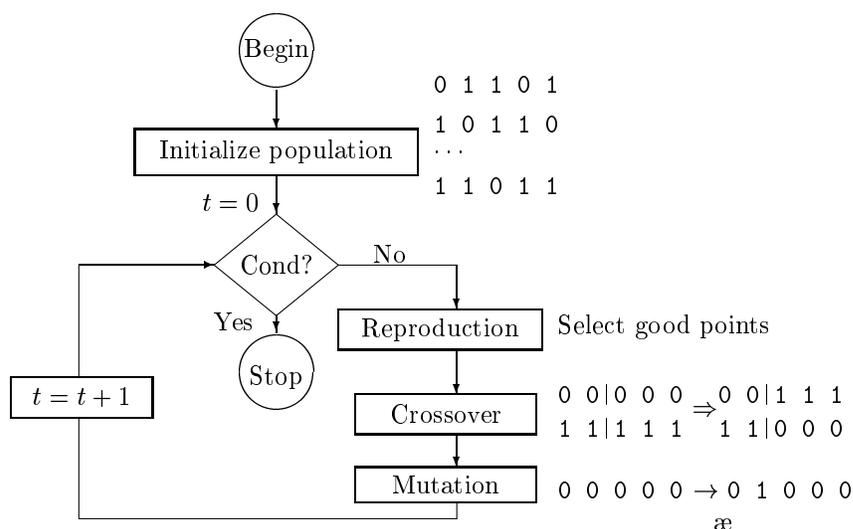


Figure 1: A flowchart of working principle of a genetic algorithm

GA begins its search from a random population of solutions. As shown in the figure, a solution in a GA is represented using a string coding of fixed length. We shall discuss about the details of the coding procedure a little later. But for now notice how a GA processes these strings in a iteration. If a termination criterion is not satisfied, three different operators—reproduction, crossover, and mutation—are applied to update the population of strings. One iteration of these three operators is known as a generation in the parlance of GAs. Since the representation of a solution in a GA is similar to a natural chromosome and GA operators are similar to genetic operators, the above procedure is named as genetic algorithm. We now discuss the details of the coding representation of a solution and GA operators in details in the following subsections.

4.1.1 Representation

In a binary-coded GA, every variable is first coded in a fixed-length binary string. For example, the following is a string, representing N problem variables:

$$\underbrace{11010}_{x_1} \quad \underbrace{1001001}_{x_2} \quad \underbrace{010}_{x_3} \quad \dots \quad \underbrace{0010}_{x_N}$$

The i -th problem variable is coded in a binary substring of length ℓ_i , so that the total number of alternatives allowed in that variable is 2^{ℓ_i} . The lower bound solution x_i^{\min} is represented by the solution (00...0) and the upper bound solution x_i^{\max} is represented by the solution (11...1). Any other substring s_i decodes to a solution x_i as follows:

$$x_i = x_i^{\min} + \frac{x_i^{\max} - x_i^{\min}}{2^{\ell_i} - 1} \text{DV}(s_i), \quad (1)$$

where $\text{DV}(s_i)$ is the decoded value¹ of the substring s_i . The length of a substring is usually decided by the precision needed in a variable. For example, if three decimal places of accuracy is needed in the i -th variable, the total number of alternatives in the variable must be $(x_i^{\max} - x_i^{\min})/0.001$, which can be set equal to 2^{ℓ_i} and ℓ_i can be computed as follows:

$$\ell_i = \log_2 \left(\frac{x_i^{\max} - x_i^{\min}}{\epsilon_i} \right). \quad (2)$$

Here, the parameter ϵ_i is the desired precision in the i -th variable. The total string length of a N -variable solution is then $\ell = \sum_{i=1}^N \ell_i$. Representing a solution in a string of bits (0 or 1) resembles a natural chromosome which is a collection of genes having particular allele values.

In the initial population, ℓ -bit strings are created at random (at each of ℓ positions, there is a equal probability of creating a 0 or a 1). Once such a string is created, the first ℓ_1 bits can be extracted from the complete string and corresponding value of the variable x_1 can be calculated using Equation 1 and using the chosen lower and upper limits of the variable x_1 . Thereafter, the next ℓ_2 bits can be extracted from the original string and the variable x_2 can be calculated. This process can be continued until all N variables are obtained from the complete string. Thus, an ℓ -bit string represents a complete solution specifying all N variables uniquely. Once these values are known, the objective function $f(x_1, \dots, x_N)$ can be computed.

In a GA, each string created either in the initial population or in the subsequent generations must be assigned a *fitness* value which is related to the objective function value. For maximization problems, a string's fitness can be equal to the string's objective function value. However, for minimization problems, the goal is to find a solution having

¹The decoded value of a binary substring $S \equiv (S_{\ell-1}S_{\ell-2} \dots S_2S_1S_0)$ is calculated as $\sum_{j=0}^{\ell} -12^j S_j$, where $S_j \in (0, 1)$.

the minimum objective function value. Thus, the fitness can be calculated as the reciprocal of the objective function value so that solutions with smaller objective function value get larger fitness. Usually, the following transformation function is used for minimization problems:

$$\text{Fitness} = \frac{1}{1 + f(x_1, \dots, x_N)}. \quad (3)$$

There are a number of advantages of using a string representation to code variables. First, this allows a shielding between the working of GA and the actual problem. What GA processes is ℓ -bit strings, which may represent any number of variables, depending on the problem at hand. Thus, the same GA code can be used for different problems by only changing the definition of coding a string. This allows a GA to have a wide spread applicability. Second, a GA can exploit the similarities in string coding to make its search faster, a matter which is important in the working of a GA and is discussed in Subsection 4.3.

4.1.2 Reproduction

Reproduction is usually the first operator applied on a population. Reproduction selects good strings in a population and forms a mating pool. There exists a number of reproduction operators in the GA literature (Goldberg and Deb, 1990), 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 i -th string in the population is selected with a probability proportional to f_i . Since the population size is usually kept fixed in a simple GA, the cumulative probability for all strings in the population must be one. Therefore, the probability for selecting i -th string is $f_i / \sum_{j=1}^N f_j$, where N is the population size. One way to achieve this proportionate selection is to use a roulette-wheel with the circumference marked for each string proportionate 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. Since the circumference of the wheel is marked according to a string's fitness, this roulette-wheel mechanism is expected to make f_i / \bar{f} copies of the i -th string, where \bar{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 (Goldberg, 1989). As will be discussed later, the proportionate selection scheme is inherently slow. One fix-up is to use a *ranking* selection scheme (Goldberg, 1989). All N strings in a population is first ranked according to ascending order of string's fitness. Each string is then assigned a rank from 1 (worst) to N (best) and an linear fitness function is assigned for all the strings so that the best string gets two copies and the worst string gets no copies after reproduction. Thereafter, the proportionate selection is used with these fitness values. This ranking reproduction scheme eliminates the function-dependency which exists in the proportionate reproduction scheme.

Recently, the *tournament* selection scheme is getting popular because of its simplicity and controlled takeover property (Goldberg and Deb, 1990). In its simplest form (binary tournament selection), 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. It is important to note that this reproduction operator does not require a transformation of the objective function to calculate fitness of a string as suggested in Equation 3 for minimization problems. The better of two strings can be judged by choosing the string with the smaller objective function value.

4.1.3 Crossover

Crossover operator is applied next to the strings of the mating pool. Like reproduction operator, there exists a number of crossover operators in the GA literature (Spears and De Jong, 1991; Syswerda, 1989), but in almost all crossover operators, two strings are picked from the mating pool at random and some portion of the strings are exchanged between the strings. In a single-point crossover operator, both strings are cut at an arbitrary place and the right-side portion of both strings are swapped among themselves to create two new strings, as illustrated in the following:

$$\begin{array}{rcc|ccc} \text{Parent1} & 0 & 0 & 0 & 0 & 0 & \Rightarrow & 0 & 0 & 1 & 1 & 1 & \text{Child1} \\ \text{Parent2} & 1 & 1 & 1 & 1 & 1 & & 1 & 1 & 0 & 0 & 0 & \text{Child2} \end{array}$$

It is interesting to note 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. Since the knowledge of an appropriate site is usually not known, a random site is usually chosen. However, it is important to realize that the choice of a random site does not make this search operation random. With a single-point crossover on two ℓ -bit parent strings, the search can only find at most $2(\ell - 1)$ different strings in the search space, whereas there are a total of 2^ℓ strings in the search space. With a random site, the children strings produced may or may not have a combination of good substrings from parent strings depending on whether the crossing site falls in the appropriate place or not. But we do not worry about this aspect too much, because if good strings are created by crossover, there will be more copies of them in the next mating pool generated by the reproduction operator. But if good strings are not created by crossover, they will not survive beyond next generation, because reproduction will 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 (Syswerda, 1989). 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. Other aspect is that the search needs to be performed in a way to preserve the information stored in the parent strings maximally, because these parent strings are instances of good strings selected using 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. However, in order to preserve some of the previously-found good strings, not all strings in the population are participated in 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 are simply copied to the new population. Even though best $100(1 - p_c)\%$ of the current population can be copied deterministically to the new population, this is usually performed stochastically.

4.1.4 Mutation

Crossover operator is mainly responsible for the search aspect of genetic algorithms, even though mutation operator is also used for this purpose sparingly. Mutation operator changes a 1 to a 0 and vice versa with a small mutation probability, p_m :

$$00000 \Rightarrow 00010$$

In the above example, the fourth gene has changed its value, thereby creating a new solution. The need for mutation is to maintain diversity in the population. For example, if in a particular position along the string length all strings in the population have a value 0, and a 1 is needed in that position to obtain the optimum or a near-optimum solution, then the crossover operator described above will be able to create a 1 in that position. The inclusion of mutation introduces some probability of turning that 0 into a 1. Furthermore, for local improvement of a solution, mutation is useful.

After reproduction, crossover, and mutation are applied to the whole population, one generation of a GA is completed. These three operators are simple and straightforward. Reproduction operator selects good strings and crossover operator recombines good substrings from two good strings together to hopefully form a better substring. Mutation operator alters a string locally to hopefully create a better string. 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 will be eliminated by the reproduction operator in the next generation and if good strings are created, they will be emphasized. Later, we shall discuss some intuitive reasoning as to why a GA with these simple operators may constitute a potential search algorithm.

4.2 A Simple Simulation

To illustrate the working of GA operators, we consider a simple sinusoidal function that is to be maximized (Deb, 1996):

$$\begin{aligned} & \text{Maximize } \sin(x) \\ & \text{Variable bound } 0 \leq x \leq \pi. \end{aligned} \tag{4}$$

For the illustration purpose, 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 the linear mapping rule (Equation 1) between the decoded value of any string, s and the bounds on the variable: $x = \frac{\pi}{31} \text{decode}(s)$, where $\text{decode}(s)$ is the decoded value of the string s , so that 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 shall use a population of size four, proportionate selection, single-point crossover with probability one, and bit-wise 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 this string corresponds to

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.

a solution $x = 0.912$, which has a function value equal to $\sin(0.912) = 0.791$. Similarly, other three strings are also evaluated. Since the proportionate reproduction scheme assigns number of copies according to a string's fitness, the expected number of copies for each

string is calculated in column 5. When a roulette-wheel selection scheme is actually implemented the number of copies allocated to the strings are shown in column 6. Column 7 shows the mating pool. It is noteworthy that the third string in the initial population had a fitness very small compared to the average fitness of the population and thus been eliminated by the selection operator. On the other hand, the second string being a good string made two copies in the mating pool. Crossover sites are chosen at random and the four new strings created after crossover is shown in column 9. Since a small mutation probability is considered, none of the bits is altered. Thus, column 9 represents the new population. Thereafter, each of these strings is decoded, mapped, and evaluated. This completes one generation of GA simulation. The average fitness of the new population is found to be 0.711, an improvement from the initial population. It is interesting to note that even though all operators use random numbers, there is a directed search and the average performance of the population usually increases from one generation to another.

4.3 Schema Processing

The working principle described above 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 try to give an intuitive answer to this doubt and remind the reader that a number of studies is currently underway to find a rigorous mathematical convergence proof for GAs (Davis and Principe, 1991; Rudolph, 1994; Vose, 1990; Vose and Liepins, 1991; Whitley, 1992). Even though the operators are simple, GAs are highly nonlinear, massively multi-faceted, stochastic, and complex.

In order to intuitively answer 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 improved the average performance of a population, but what has actually been processed in one cycle of GA operators? If we investigate carefully we may observe that there are some similarities in string positions among good strings and by the application of three GA operators the number of strings with those similarities has increased from the initial population to the new population. These similarities are called *schema* (schemata, in plural) in the GA literature. More specifically, a schema represents a set of strings with certain similarity at certain string positions. To represent a schema for binary codings, a triplet (1, 0, and *) is used. 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 the number of strings they represent is increased from the initial population to the new population or not. But what does these schemata mean anyway?

Since a schema represents certain similar strings, a schema can be thought of representing certain region in the search space. For the above function the schema H_1 represents strings with x values varying from 1.621 to 2.331 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 with function values varying from 0.0 to 0.651. Since our objective is to maximize the function, we would like to have more copies of strings representing schema H_1 than H_2 . By one generation of three genetic operators, this is what we have accomplished in Table 1 without having to count all these schema competitions, without the knowledge of the complete search space, and by manipulating only a few instances of the search space. Holland (1975) and later Goldberg (1989) have argued that by processing only N strings in a generation, A GA effectively processes $O(N^3)$ schemata. This

leverage gives a GA its search power and provides the *implicit parallelism* in its search.

The schema H_1 for the above example has only two defined positions (the first two bits) and both defined bits are tightly spaced (very close to each other) and contains the possible near-optimal solution (the string (1 0 0 0 0) is the optimal string in this problem). The schemata that are short and above-average are known as *building blocks*. While GA operators are applied on a population of strings, a number of such building blocks in various parts along the string get emphasized, like H_1 in the above example. Finally, these little building blocks are combined together due to combined action of GA operators to form bigger and better building blocks. This building process continues until the optimal solution or a near-optimal solution is formed. In the absence of any rigorous convergence proofs, this is what is hypothesized to be the reason for GA's success. This hypothesis is largely known as *Building Block Hypothesis* (Goldberg, 1989).

5 GA Parameter Setting

The building block hypothesis gives an intuitive and qualitative reasoning to what might cause GAs to work. But it tells nothing about for what values of various GA parameters GAs would work. In this subsection, we first present some guidelines for successful application of GAs and later discuss procedures of setting appropriate GA parameters for successful application of a GA. Before we present the guidelines, it is important to realize that the key insight to Holland's discovery of genetic algorithms is the respect of building blocks under genetic operators. It is an established fact (albeit 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, 1993):

1. GAs process building blocks. Therefore, a proper understanding of the underlying building blocks in any given search optimization problem needs to be clearly understood. The knowledge of building blocks in a problem can assist in designing a 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 the building block competitions 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.

5.1 Population size

While applying a GA to a search and optimization problem, the first task ahead of the user is to set an appropriate population size. There exists a number of studies which either uses simulations or uses schema processing to suggest an adequate population size. Goldberg (1985) suggested a population sizing which only depended on the string length, ℓ :

$$N = 1.65 \times 2^{0.21\ell}. \quad (5)$$

Schaffer et al (1989) have conducted extensive simulations on a number of test functions and concluded that a small population of size 20 to 30, a crossover probability in the range 0.75 to 0.95, and a mutation probability in the range 0.005 to 0.01 perform well. Later,

Goldberg, Deb, and Clark (1992) have calculated a population sizing which depended on the nonlinearity in the problem. They did a statistical analysis based on correct schema processing and found the following population sizing equation:

$$N \approx O(2^k \frac{\sigma^2}{d^2}), \quad (6)$$

where k is the order of nonlinearity, σ^2 is the variance of the problem and d is the difference in fitness values between local and global optimal solutions. This equation suggests that the population sizing would be more if the signal to be detected over noise is small. Also, as the order of nonlinearity increases, GA requires a large population size. Harik et al. (1996) have modified the above population sizing recently ($N \approx O(2^k \sigma/d)$), which is found to agree better with simulation results on many controlled test problems.

One important outcome of above studies is that for problems of bounded difficulty, the population sizing must be of the order of the string length ($N \approx O(\ell)$). This suggests that if the chosen string length is, say, 30 or so, at least a population of size 30 to 50 is adequate, whereas for a string length of 200 or so, a population size of 100 to 200 is necessary, and so on. However, one question is often asked: Is a larger population size always better in a GA? We try to answer this question practically and then show GA simulation results in favor of our argument.

Let us pose the problem practically. Say we have to solve a problem using a GA and we are allowed to use a total of S function evaluations (recall that function evaluations in complex search optimization problems are the most time consuming operations). Now, we have to make a decision about an adequate population size. If we choose to use a population of size N , we are prepared to run a GA for a total of S/N generations. Thus, if we choose a smaller population size, we are allowed to run GA for longer number of generations, and vice versa. What remains to be answered is then will any population size solve the problem with a high confidence? We answer this question by showing GA simulation results on a couple of two-variable optimization problems.

The first function has a minimum solution at $x^* = (3, 2)$ with a function value zero:

$$\begin{aligned} \text{Minimize } f_1(x_1, x_2) &= (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 \\ 0 \leq x_1, x_2 &\leq 6. \end{aligned} \quad (7)$$

In the feasible region, the function is unimodal (with only one minimum solution). We use different population sizes and run GAs using 50 different initial populations in each case. A performance measure is defined as the ratio of successful GA runs (which satisfies only the first termination criterion above) and the maximum number of runs (50, in this case). A GA run is terminated if any of the following two cases is satisfied:

1. A solution in a small neighborhood (± 0.01) at the optimum solution is found in any generation, or
2. A maximum of S/N generations have been performed. The parameter S is chosen so that a GA uses a maximum of 5% of the total feasible search space. For this problem, $S = 0.05 \times \frac{6}{0.02} \times \frac{6}{0.02} = 4,500$. This also means that a random search method will succeed in only 5% of the runs (with a performance measure equal to 0.05).

Each variable is coded in 12 bit strings. Binary tournament selection, single-point crossover (with $p_c = 0.9$), and bit-wise mutation (with $p_m = 0.02$) are used. Figure 2 shows the performance measure versus population size, when applied to the unimodal function f_1 . A performance measure of one means GAs successfully converge near the optimum solution in *all* 50 runs. The figure shows a typical pattern of a GA's performance to unimodal (or simpler) problems. There are four phases depicted in the figure:

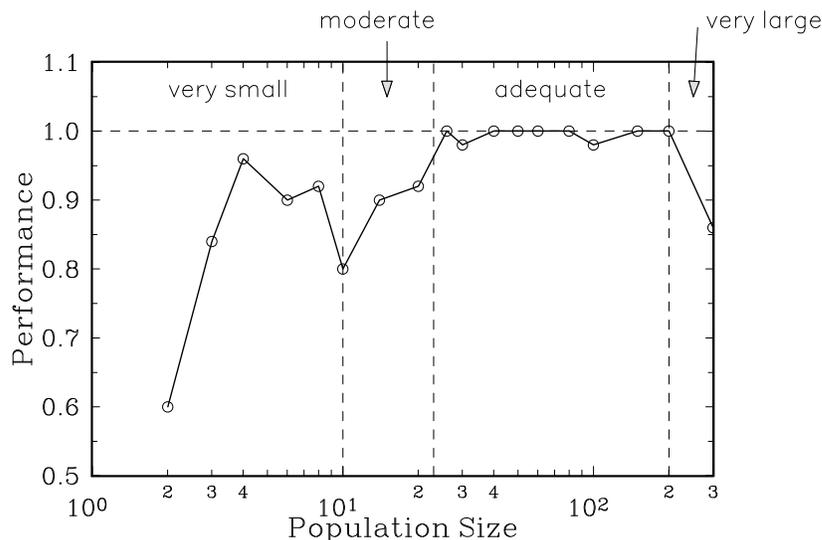


Figure 2: GA's performance on different population size for the unimodal function f_1

Very small population size: The run with $N = 2$ is essentially a non-recombinative GA, where reproduction and mutation together constitute the search effort. GAs with a small population size (but more than 2) is similar to an intelligent hill-climbing strategy and perform well in unimodal problems (shown in the figure). However, as expected, the GA works rather slowly under the primary mutation operator. If large enough generations are allowed, this GA will eventually find a solution near the true optimum. For example, with $N = 4$, a maximum of $4,500/4$ or 1,125 generations are allowed and GA succeeds in 96% simulations. However, as the population size is increased, the number of allowable generations reduces (since the total number allowed function evaluation is fixed) and GA's performance declines. At this phase, no real advantage of recombination operator of GA is exploited.

Moderate population size: At this population sizes, GA's recombination operator starts to get adequate population members to find useful recombinations. GA's performance begins to improve with an increase in population size.

Adequate population size: The population size is adequate to allow necessary schema processing and a GA performs successfully in almost all simulations. As the population size increases, GA may require more function evaluations to find the first solution near the optimum solution, but a GA works at its best with these population sizes.

Very large population size: Since the total number of function evaluations S is kept the same in all simulations in this study, for very large populations GAs are only allowed to run for a small number of generations. Since their runs are terminated prematurely, their performances begin to reduce. However, if the constant function evaluation assumption is relieved, GAs with a population size larger than needed

should also find near-optimal solutions, but probably with more function evaluations than needed.

Good performances at very small population sizes (4 or 5) may suggest the use of small population in a GA. However, we would like to reiterate that such a small population size may be beneficial in simpler problems (mostly unimodal), where a hill-climbing strategy is enough to find the optimal solution. It is worth mentioning here that the micro-GA (which uses a population size of 4 or 5) suggested by Krishnakumar (1989) has been found adequate to solve some engineering design problems. However, it is also interesting to note that the GA with an adequate population sizing (say 40 in the above problem) requires only an average of 1,031 function evaluations to find the first solution near the true optimum solution compared to 1,733 in $N = 4$ case.

What is most striking is that the high-performing GAs with very small population size observed in the above study does not exist when a complex problem is tried. This is because complex problems are not solvable by simple hill-climbing strategies and require the true recombinative power of GAs to solve them. We change the above test function to the following function which introduces four optimal solutions of which only one is the global optimal solution (still at (3,2)):

$$\begin{aligned} \text{Minimize } f_2(x_1, x_2) &= (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 + 0.1[(x_1 - 3)^2 + (x_2 - 2)^2] \\ &-6 \leq x_1, x_2 \leq 6. \end{aligned} \tag{8}$$

A contour plot of the above function is shown in Figure 3.

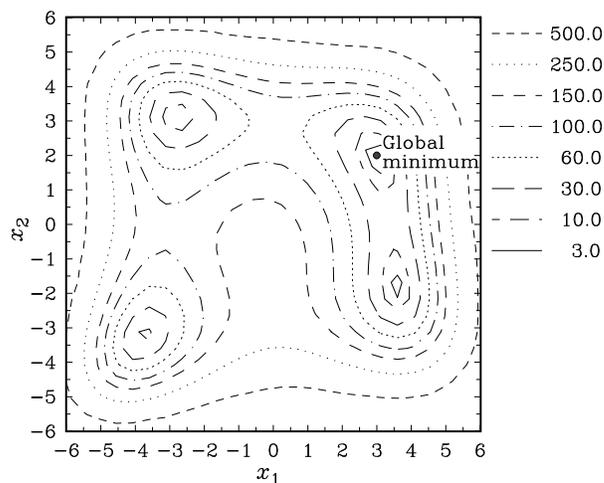


Figure 3: A contour plot of the multimodal function f_2 with four minima

This problem is more difficult to solve than the previous function, simply because of three other locally optimal solutions where a GA can get attracted to. We use the same 12-bit strings to represent each variable in the range $(-6, 6)$. We now terminate a GA run if one of the following conditions is met:

1. A solution in a small neighborhood (± 0.02) of the optimum solution is found in any generation, or
2. A maximum of S/N generations have been performed, where S is chosen so that a GA uses a maximum of 10% total feasible search space (In this case, $S = 9,000$).

All other parameters are the same as before. Figure 4 shows the performance measure of GAs versus population size. This is a typical performance of GAs for a complex

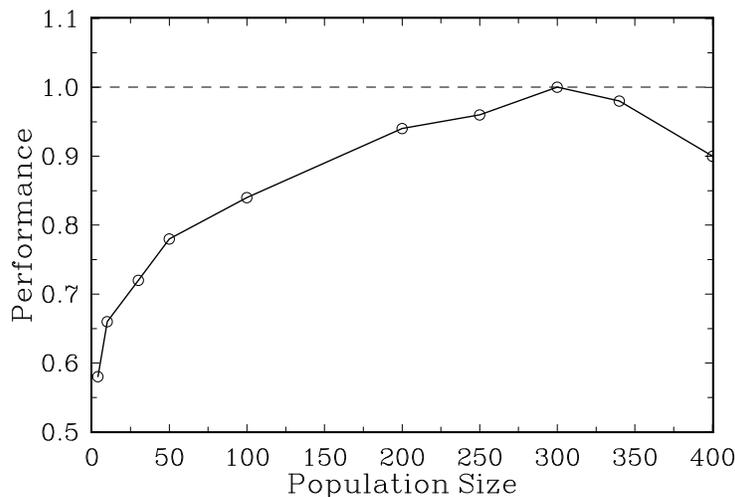


Figure 4: GA's performance on different population sizes for the multimodal function f_2

problem. There are three regions in the figure. The effect of very small population size is absent in these problems. As the population size is increased, GA's performance increases due to increased probabilities of correct schema processing. At a critical population size, GA starts finding the global optimal solutions in almost all simulations. The performance reduces for very large populations due to inadequate number of generations allowed in the simulations. However, if enough generations are allowed, GAs with very large populations can also solve these problems with much confidence, but with more function evaluations.

All these simulations suggest that for a generic problem and given number of function evaluations, there is a critical range of population size, at which GAs will work more reliably for successfully. However, if the number of function evaluations is not a concern, larger the population size, better would be the GA's ability to solve the complex problems.

5.2 Reproduction Parameters

Reproduction is primarily responsible for providing a direction of search in a GA. Given a set of solutions in a population, it emphasizes good solutions by making duplicates of them and by eliminating bad solutions. Although there exist a number of reproduction operators to perform this task (such as proportionate selection, tournament selection, ranking selection, and others), they all are characterized by a parameter called *selection pressure*. This parameter accounts for the number copies of the best solution gets after reproduction operation. Table 2 shows the selection pressure of some of the commonly-used reproduction operators. It is clear that for a reproduction operator with a large selection pressure, the best solution gets many copies after reproduction and the population loses diversity. Thus to avoid premature convergence a destructive recombination operator is needed.

Table 2: Selection pressure of some commonly-used reproduction operators

Operator	Selection pressure
Proportionate	$\frac{f_{\max}}{f}$
Binary tournament	2
s -ary tournament	s

5.3 Crossover Parameters

Crossover operator introduces new solutions in the population by recombining partial solutions of parent solutions chosen from a mating pool obtained by the reproduction operator. Like the reproduction operator, there also exist a number of crossover operators, such as single-point crossover, two-point crossover, uniform crossover, and others. Given two parent solutions, each operator has a different *search power*. We define search power as the measure of a proportion of the search space which can be reached by applying the crossover operator once to two complimentary parent solutions. Table 3 shows the search power of three commonly-used crossover operators applied on strings of length ℓ . The

Table 3: Search power in different crossover operators

Operator	Search power
Single-point	$\frac{\ell-1}{2^{\ell-1}}$
Two-point	$\frac{\binom{\ell-1}{2}}{2^{\ell-1}}$
Uniform	1.00

table shows that the search power is maximum in the uniform crossover. For the same reason, uniform crossover is also very destructive and has low probability in preserving already-found building blocks. The effect of crossover's search power is also controlled using a parameter called the crossover probability, p_c . If p_c is close to one, crossover is performed on the whole population and the search effect is maximum. It is a usual practice to set p_c to a large value (in the range 0.7 to 1.0) (Goldberg, 1989; Schaffer et al., 1989)

It is important to note (and as hinted in the previous subsection) that a balance in the search effort of crossover operator and the selection pressure of reproduction operator is a must for successful working of a GA. Goldberg, Deb, and Thierens (1991) have found the following relationship between selection pressure, s and p_c for successful application of a GA to bit-wise linear problems:

$$O(\ln s) \leq p_c \leq 1. \quad (9)$$

Thierens and Goldberg (1993) have found a similar relationship for nonlinear problems.

5.4 Mutation Parameters

Mutation operator maintains diversity in the population and hence used with a small probability, p_m . The usual practice is to set the mutation probability such that on an average there is only one mutation per string. Thus, a p_m is set in the range $0.1/\ell$ to $1/\ell$.

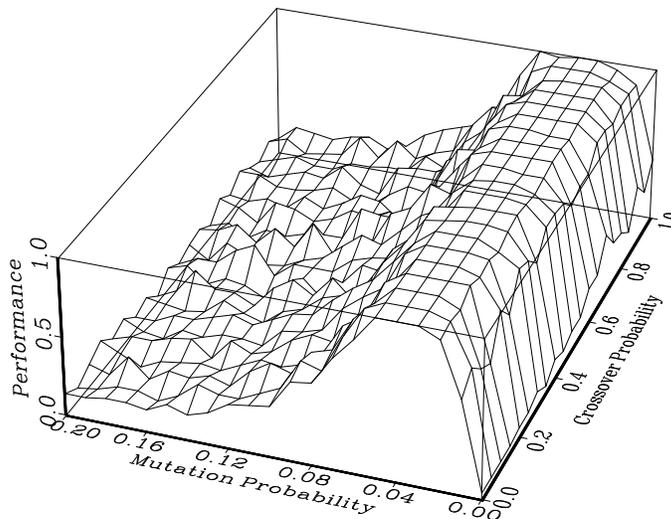


Figure 5: GA’s performance for different crossover and mutation probability on the unimodal function f_1 (population size is 50)

However, to investigate the effect of p_c and p_m on the performance of GAs, we apply GAs with different p_c and p_m values on the f_1 function given in Equation 7. A fixed population of size of 50 and $S = 4,500$ is used. Although the function is unimodal in the search space, Figure 5 shows that the GA does not work well for any combination of p_c and p_m . The figure shows that GA’s performance is sensitive to mutation probability more than the crossover probability. For a p_m in the range between 0.02 to 0.06, GAs are almost insensitive to the crossover probability p_c . These values agree with the suggested setting: $p_m \approx 1/\ell$.

5.5 Constraint Handling

Like classical search and optimization methods, GAs also face difficulty in handling constraints. Although a number of sophisticated methods have been suggested to handle constraints uniquely in GAs, the most commonly-used strategy is the penalty function method (Deb, 1995). In the penalty function method, constraints are first normalized and then a bracket-operator penalty term is used to add an extra value (in the case of minimization problems) in the objective function, if a constraint is violated:

$$\text{Penalized Objective} = f(x) + R \left(\sum_{j=1}^J \langle \bar{g}_j(x) \rangle^2 + \sum_{k=1}^K [\bar{h}_k(x)]^2 \right), \quad (10)$$

where $\langle \alpha \rangle$ is the bracket operator and it is equal to α if α is negative and is equal to zero, otherwise. Notice that the inequality constraints \bar{g}_j and equality constraints \bar{h}_k are normalized. Reproduction operation is performed with the penalized objective, instead of the original objective function $f(x)$. This way, the reproduction operator discourages the propagation of the infeasible solutions to future generations. One difference in the implementation of penalty function method in a GA and in a classical method is that in the former a constant penalty parameter is used throughout. Since GAs do not use

the gradients, sequential update of penalty parameter in a GA is not required. However, studies on innovative use of penalty function method (which are only applicable to population-based approaches) are now being available and are found to be superior to standard penalty function method commonly-used in GAs (Michalewicz and Schoenauer, 1996). However, more studies of these new approaches are needed to replace the current practice.

After a rather long discussion on the working philosophies of a GA, let us now show at least one applicable of a GA to a real-world engineering design problem.

6 Genetic Algorithm: Applications

GAs have been applied to various search and optimization problems (Back, Fogel, and Michalewicz, 1997; Bramlette and Bouchard, 1991; Callahan and Weeks, 1992; Gen and Cheng, 1997; Lucasius and Kateman, 1989; Rajeev and Krishnamoorthy, 1992). In this section, we show one application of GAs to an engineering design problem. Thereafter, we discuss how GAs can be applied in other soft computing techniques.

6.1 Car Suspension Design

Suspension systems are primarily used in a car to isolate the road excitations from being transmitted directly to the passengers. In a two-dimensional model of a car suspension system, only two wheels (one each at rear and front) are considered. Thus, the sprung mass is considered to have vertical and pitching motions only. The dynamic model of the suspension system is shown in Figure 6. For more information, refer to the detailed study (Deb and Saxena, 1997). The following nomenclature is used in the design formulation.

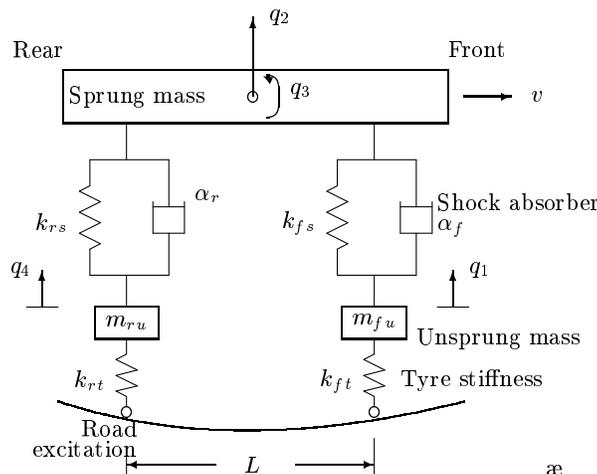


Figure 6: The dynamic model of the car suspension system

Sprung mass m_s ,	Front coil stiffness k_{fs} ,
Front unsprung mass m_{fu} ,	Rear coil stiffness k_{rs} ,
Rear unsprung mass m_{ru} ,	Front tyre stiffness k_{ft} ,
Rear damper coefficient α_r ,	Rear tyre stiffness k_{rt} ,
Front damper coefficient α_f ,	Axle-to-axle distance L ,
Polar moment of inertia of the car J .	

Since a suspension designer is interested in choosing the optimal dampers and suspension coils, we consider only four of the above parameters—front coil stiffness k_{fs} , rear coil stiffness k_{rs} , front damper coefficient α_f , and rear damper coefficient α_r —as design variables. Considering the forces acting on the sprung mass and on the front and rear unsprung mass, we can write the differential equations governing the vertical motion of the unsprung mass at the front axle (q_1), the sprung mass (q_2), and the unsprung mass at the rear axle (q_4), and the angular motion of the sprung mass (q_3) as follows (Deb, 1995). These coupled differential equations can be solved using a numerical integration technique to obtain the pitching and bouncing dynamics of the sprung mass m_s .

We choose to find the optimal suspension system which would minimize the bouncing transmissibility—the ratio of the bouncing amplitude $|q_2(t)|$ of the sprung mass to the maximum road excitation amplitude, A . A number of practical guidelines (such as the natural frequency considerations, limitations in the vertical jerk experienced by passengers, piece-wise variation of spring and damper characteristics), often used in automobile industries, are also considered in the optimal problem formulation.

The following parameters of the car suspension system are used in all simulations:

$$\begin{aligned} m_s &= 730 \text{ kg}, & m_{fu} &= 50 \text{ kg}, & m_{ru} &= 115 \text{ kg}, \\ k_{ft} &= 15 \text{ kg/mm}, & k_{rt} &= 17 \text{ kg/mm}, \\ \ell_1 &= 1.50 \text{ m}, & \ell_2 &= 1.35 \text{ m}, & L &= 2.85 \text{ m}, \\ v &= 5 \text{ Kmph}, & J &= 2.89(10^4) \text{ kg-m}^2. \end{aligned}$$

The car motion is simulated over a sinusoidal bump having 500 mm width and 70 mm height. In all solutions, the spring rates are expressed in Kg/mm and damping coefficients are in Kg-s/mm.

In order to investigate the complexity of the search space, we plot the feasible region along with the contours of the objective function in Figure 7 for fixed values of two of the four variables. The figure shows that the search space has a number of *islands* of

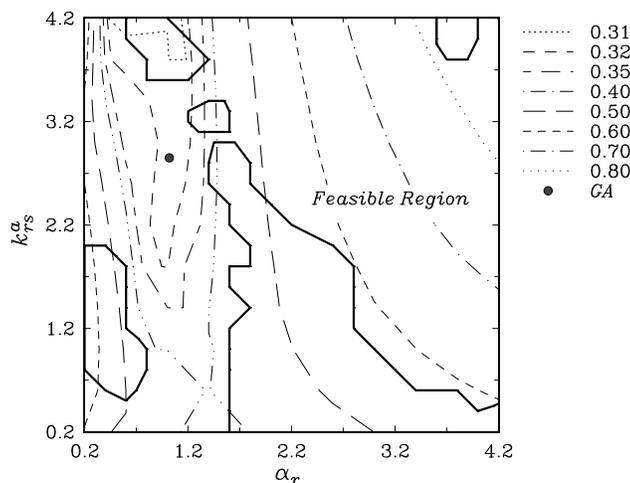


Figure 7: Feasible search space and the GA-optimized solution

infeasible regions, which forces most of the classical search algorithms to converge on a constraint boundary. When sequential quadratic programming (SQP) method is used to solve the above problem, the obtained solutions are found to be largely dependent on the initial solution (Table 4). The final solution gets stuck to a constraint boundary closer to

Table 4: Sequential quadratic programming results

	k_{fs}	α_f^a	k_{rs}^a	α_r^a	Trans.
Initial	1.00	1.00	1.00	1.00	0.96
Final	4.31	0.23	0.83	4.99	0.49
Initial	1.50	3.00	1.00	0.75	0.82
Final	1.85	2.97	0.10	0.34	0.57
Initial	1.00	1.00	1.50	0.90	0.89
Final	3.76	2.89	3.10	2.19	0.48

the initial solution. However, when GAs are used, the optimal solution (as shown in the figure) is always obtained.

The following GA parameters are used for all four variables.

Overall string length : 40 (10 for each variable)

Population size : 30

Crossover probability : 0.8

Mutation probability : 0.01

A fixed penalty parameter of $R = 100$ is used. With above parameters, the binary-coded GA finds the following solution:

$$k_{fs} = 4.53, \quad \alpha_f^a = 1.72, \quad k_{rs}^a = 2.86, \quad \alpha_r^a = 1.01.$$

The bouncing transmissibility for this solution is 0.315. The existing design for a car having identical data (as used in a renowned Indian automobile industry) is as follows:

$$k_{fs} = 1.56, \quad \alpha_f^a = 3.30, \quad k_{rs}^a = 1.45, \quad \alpha_r^a = 1.00.$$

The bouncing transmissibility for this existing design is 0.82. Comparing these two solutions, we notice that GA-optimized design has 64% lesser transmissibility than that in the existing design. The maximum jerk for this suspension is found to be 5.094 m/s^3 , whereas the allowable limit is 18.0 m/s^3 . To better appreciate the GA-optimized design, we plot the bouncing amplitude of the sprung mass, as the car moves over the bump in Figure 8. The first peak arises as the front tyre moves over the bump and the second significant peak at 2.1 sec arises when the rear tyre moves over the same bump. The figure clearly shows that the car having the GA-optimized suspension is better than the existing car suspension.

We now consider the three-dimensional model and introduce jerk and a set of frequency constraints to make the design procedure more realistic. Automobile industries design cars having front natural frequencies smaller than the rear natural frequencies (Guest, 1925). This is achieved to make the pitching oscillations die down faster. In this model, all four wheels are considered. Thus, the rolling motion of the sprung mass can also be studied. The sprung mass can have three motions—vertical bouncing, pitching, and rolling. Besides, each of the four unsprung masses will have a vertical motion. Thus, there are a total of seven second-order differential equations governing the motion of the sprung and unsprung masses, which can be derived using similar methods adopted in two-dimensional model. Since both left and right rear (or front) wheels have the same suspension system, the number of variables in three-dimensional optimal design model is also four. However, the dynamics of the car will be different than that in the two-dimensional case. Here, there are 14 nonlinear differential equations which are solved using a numerical integration procedure and all motions can be computed for a specified road profile.

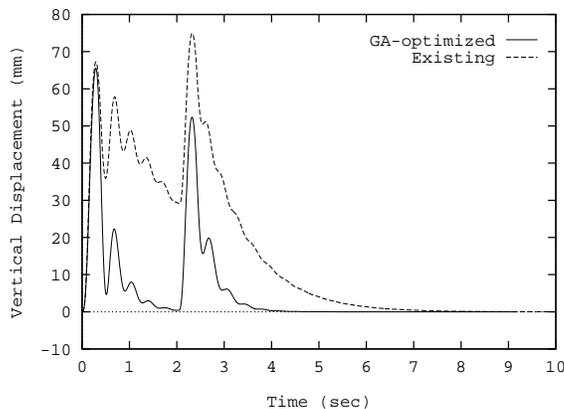


Figure 8: Bouncing amplitude of the sprung mass as the car moves over a bump for existing and GA-optimized suspension

In this model, $m_s = 1,460$ Kg and the wheel-to-wheel distance is 1.462 m are used. To make the problem more realistic, a polyharmonic road profile having wavelength varying between 100 mm to 5000 mm is chosen. The velocity of the car is assumed to be 50 Kmph. The passengers riding over such a road will be exposed to vibrations of varying frequency and amplitude. ISO 2631 (1985) limits the extent of vibrations on different frequency levels which are allowed for different levels of comfort. Specifically, the limits are presented in the form of a chart with frequency and acceleration for different levels of allowable exposure time. A suspension which introduces smaller vertical acceleration in the sprung mass allows a longer exposure time, thereby increasing the ride comfort of the passengers (Markine et al., 1996). Therefore, we use the following procedure to calculate the objective of the optimal suspension design.

The vertical motion (q_2) of the sprung mass is simulated by solving the governing equations of motion for the above mentioned realistic road. Thereafter, the vertical acceleration (\ddot{q}_2) is calculated by numerically differentiating the vertical motion of the sprung mass. The time-acceleration data is then Fourier-transformed to calculate the vertical acceleration as a function of the forcing frequency. The total area under the acceleration-frequency plot is used as the objective. This area is then minimized to obtain the optimal suspension which will give maximum comfort to the passengers.

The same GA parameters are used and the following suspension has been obtained:

$$k_{fs} = 1.45, \quad \alpha_f^a = 0.14, \quad k_{rs}^a = 1.28, \quad \alpha_r^a = 0.11.$$

The front and rear natural frequencies of the suspension system are found to be 1.35 and 1.37 Hz. It is clear that the front natural frequency is smaller than that of the rear. This solution is close to the optimal solution since this design makes the front natural frequency almost equal to rear natural frequency, thereby making the constraint active. When the peak acceleration at significant frequencies are plotted on the ISO 2631 chart (Figure 9), it is observed that the comfortable exposure time for the existing design is only about 25 minutes, whereas that for the GA-optimized design is about 4 hours, thus giving a much longer comfortable ride.

6.2 Genetic Algorithms and Soft Computing

Soft computing techniques are increasingly getting popular in various applications of science and engineering primarily due to their simplicity and computational advantages

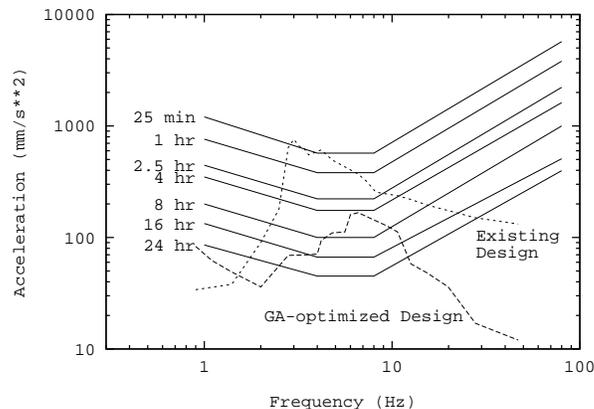


Figure 9: Exposure times for existing and GA-optimized designs using ISO 2631 chart

over classical methods (Herrera and Verdegay, 1996; Jain and Jain, 1997; Jang, Sun, and Mizutani, 1997; Sanchez, Shibata, and Zadeh, 1997). GAs are also a part of soft computing and can be used in conjunction with other soft computing techniques such as fuzzy logic techniques and neural networks. In fact, GAs must be used in these methods, not because they are interesting and new, but because they work better in unison. They all complement each other's performance when used in an appropriate manner. In the following subsections, we discuss some possibilities of their use.

6.2.1 Fuzzy Logic Technique and Genetic Algorithm

Fuzzy logic technique is primarily applied in optimal control problems where a quick control strategy is needed and imprecise and qualitative definition of action plans are available. There are primarily two activities in designing an optimal fuzzy controller:

1. Find optimal membership functions for control and action variables, and
2. Find an optimal set of rules between control and action variables.

In both these cases, GAs can be suitably used. Figure 10 shows typical membership

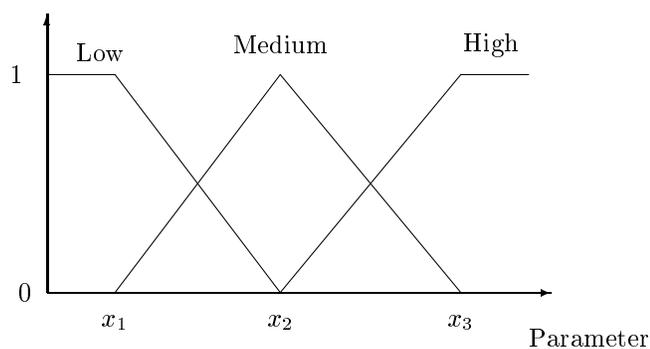


Figure 10: Fuzzy membership functions and typical variables used for optimal design functions for a variable (control or action) having three choices—low, medium, and high.

Since the maximum membership function value of these choices is always one, the abscissas marked x_i are usually chosen by the user. GAs can treat these abscissas as variables and an optimization problem can be posed to find these variables for minimizing or maximizing a control strategy (such as time of overall operation, product quality, and others). A number of such applications exist in the literature (Herrera and Verdegay, 1996; Karr, 1991).

The second proposition of finding an optimal rule base using GAs is more interesting and is unique. Let us take an example to illustrate how GAs can be uniquely applied to this problem. Let us assume that there are two control variables (temperature and humidity) and there are three options for each—low, medium, and high. There is one action variable (water jet flow rate) which also takes one of three choices—low, medium, and high. With these options, there are a total of 3×3 or 9 combinations of control variables possible. In fact, considering the individual effect of control variable separately, there are a total of $((4 \times 4 - 1)$ or 15 total combinations of control variables possible. Thus, finding an optimal rule base is equivalent to finding one of four options (fourth option is no action) of the action variable for each combination of the control variables. A GA with a string length of 15 and with a ternary-coding can be used to represent the rule base for this problem. Following is a typical string:

3 1 2 4 2 4 3 4 4 2 4 3 2 2 4

Each position in the string signifies a combination of action variables. In the above coding, a 1 represents *low*, a 2 represents *medium*, a 3 represents *high* value of the action variable, and a 4 means no action, thereby signifying the absence of the corresponding combination of action variables in the rule base. Thus, the above string represents a rule base having 9 rules (with non-4 values). The rule base does not contain 6 combinations of action variables (namely, 4-th, 6-th, 8-th, 9-th, 11-th, and 15th combinations). Table 5 shows the corresponding rule base. Although this rule base may not be the optimal one,

Table 5: Action variable for a string representing a fuzzy rule base shown in slanted fonts

		Temperature			
		Low	Medium	High	Don't Care
Humidity	Low	<i>High</i>	<i>Medium</i>		<i>Medium</i>
	Medium	<i>Low</i>		<i>Medium</i>	<i>Medium</i>
	High	<i>Medium</i>	<i>High</i>		
	Don't Care			<i>High</i>	

GAs can process a population of such rule bases and finally find the optimal rule base. Once the rules present in the rule base is determined from the string, fixed user-defined membership functions can be used to simulate the underlying process. Thereafter, the objective function value can be computed and the reproduction operator can be used. The usual single-point crossover and a mutation operator (one allele mutating to one of three other alleles) can be used with this coding. Notice that this representation allows GAs to find the optimal number of rules and optimal rules needed to solve the problem simultaneously. In the above problem, binary strings, instead of ternary strings, can also be used. Each of four options in the action variable can now be represented by two bits and a total of 30 bits is necessary to represent a rule base. Since GAs deal with discrete variables and with a string representation of a solution, the above scheme of finding an optimal rule base with optimal number of rules is unique in GAs.

It is interesting to note that both optimal membership function determination and optimal rule base identification tasks can be achieved simultaneously by using a con-

catenation of two codings mentioned above. A part of the overall string will represent the abscissas of the control variables and the rest of the string will represent the rules present in the rule base. The overall fitness of the string is then calculated using both the membership function as well as the rule base obtained from the string.

6.2.2 Neural Networks and Genetic Algorithm

Neural networks have been primarily used in problems where a non-mathematical relationship between a given set of input and output variables is desired. GAs can be used nicely in two major activities in neural network applications:

1. GAs can be used as a learning algorithm (instead of the popular Backpropagation method (McClelland and Rumelhart, 1988)) for a user-defined neural network and
2. GAs can be used to find the optimal connectivity among input, output, and hidden layers (with identification of number of neurons in the hidden layers).

Once the network connectivity is fixed, each connection weight in the network including the biases can be used as a variable in the GA string. Instead of using Backpropagation or other learning rules, GAs can be cranked to find the optimal combination of weights which would minimize the mean-squared error between the desired and obtained outputs. Since the backpropagation algorithm updates the weights based on steepest gradient descent approach, the algorithm has a tendency to get stuck at locally optimal solutions. GA's population approach and inherent parallel processing may allow them not to get stuck at locally optimal solutions and may help proceed near the true optimal solutions. The other advantage of using GAs is that they can be used with a minor change to find an optimal connection weight for a different objective (say, minimizing variance of the difference between desired and obtained output values, and others). To incorporate any such change in the objective of NN technique using the standard practice will require development of a very different learning rule, which may not be tractable for some objectives.

The optimal connectivity of a neural network can also be found using GAs. This problem is similar to finding optimal truss structure optimization problems (Chaturvedi, Deb, and Chakrabarty, 1995; Sandgren and Jensen, 1990) or finding optimal networking problems. The standard search techniques used in those problems can also be used in optimal neural network design problems. However, here we discuss a different strategy adopted by Miller, Todd, and Hegde (1989). A matrix of 1 and 0, specifying whether a connection exists from node i to node j (where i and j varies from 1 to the maximum number of nodes in the network) or not, constitutes the design variables. A neural network (for solving XOR problem) constructed from a typical binary string is shown in Figure 11. The string also carries information about the bias in a neuron. In the figure, the neurons 3, 4, and 5 have the biases. The fitness of this string can be calculated by simulating backpropagation learning on a set of input-output data up to a certain number of epoches and by calculating the mean-squared error between the desired and obtained output values. A string is has a high fitness if the mean-squared error is less. Based on such fitness information, a GA can evolve neural networks which solve the underlying problem optimally (with minimum mean-squared error).

It is important to realize that both problems of finding an optimal network and finding optimal connection weights in the neural network can also be coded simultaneously in a GA. The optimal solution thus found will be the true optimal solution of the overall problem which is likely to be better than that obtained in any of the individual optimization problems. GAs offer an efficient way to solve both the problems simultaneously (Winter et al, 1996).

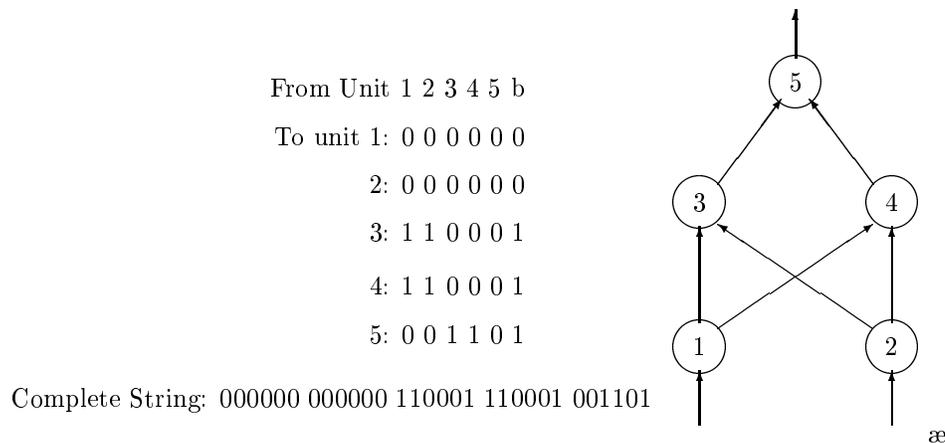


Figure 11: A typical string for neural network connectivity design

7 Summary

In this paper, we have described a new yet potential search and optimization algorithm originally conceived by John Holland about three decades ago. A genetic algorithm (GA) is different from other classical search and optimization methods in a number of ways (Goldberg, 1989): A GA does not use gradient information; it works with a set of solutions instead of one solution in each iteration; it works on a coding of solutions instead of solutions themselves; it is a stochastic search and optimization procedure; and it is highly parallelizable. The GA technique is an abstraction of natural genetics and natural selection processes and aims to solve search and optimization problems. GAs are finding increasing popularity primarily because of their wide spread applicability, global perspective, and inherent parallelism.

There exists a number of extensions to the simple GA described in this paper. Interested readers may refer to the GA literature for details:

Real-coded GA: Variables taking real values are used directly. Although the same reproduction operator described here can be used, the trick lies in developing an efficient crossover and mutation operators (Deb and Agrawal, 1995; Deb and Kumar, 1995; Eshelman and Schaffer, 1992). The real-coded GAs eliminate the arbitrary precision and Hamming cliff problem that binary GAs may have.

Micro GA: A small population size (of the order of 4 or 5) is used (Krishnakumar, 1989). This GA largely depends on the mutation operator, since such a small population cannot take advantage of the discovery of good partial solutions by a selecto-recombination GA. However, for unimodal and simple problems, micro-GAs are good candidates.

Knowledge-augmented GA: GA operators and/or the initial population is assisted with problem knowledge, if available. In most problems, some problem information is available and generic GA operators mentioned in this paper can be modified to make the search process faster (Davidor, 1991; Deb, 1993)

Hybrid GA: A classical greedy search operator is used starting from a solution obtained by a GA. Since a GA can find a good regions in the search space quickly, using a greedy approach from a solution in the global basin may make the overall search effort efficient (Powell and Skolnick, 1989; Kelly and Davis, 1991).

Multimodal GA: Due to the population approach, GAs can be used to capture multiple optimal solutions in one simulation of a GA run. The reproduction operator needs to be performed with *shared* fitness value, computed by degrading a string's original fitness by the number of strings closer to it (Deb, 1989; Deb and Goldberg, 1989; Goldberg and Richardson, 1987).

Multi-objective GA: Multiple Pareto-optimal solutions are found simultaneously in a population. A GA is unique optimization algorithm in solving multi-objective optimization problems in this respect. In one implementation, non-domination concept is used with all objective functions to determine a fitness measure for each solution. Thereafter, the GA operators described here are used as usual. On a number of multi-objective optimization problems, this non-dominated sorting GA has been able to find multiple Pareto-optimal solutions in one single run (Fonseca and Fleming, 1993; Horn and Nafploitis, 1993; Srinivas and Deb, 1994).

Nonstationary GA: The concept of diploidy and dominance can be implemented in a GA to solve nonstationary optimization problems. Information about earlier good solutions can be stored in recessive alleles and when needed can be expressed by suitable genetic operators (Goldberg and Smith, 1987).

Scheduling GA: Job-shop scheduling, time tabling, traveling salesman problems are solved using GAs. A solution in these problems is a permutation of N objects (name of machines or cities). Although reproduction operator similar to one described here can be used, the crossover and mutation operators must be different. These operators are designed in order to produce offsprings which are valid and yet have certain properties of both parents (Davis, 1991; Goldberg, 1989; Starkweather, 1991).

Acknowledgments

The author acknowledges the help of Vikas Saxena in performing the simulation results of the car suspension design problem. This work is funded by the Department of Science and Technology, Government of India, under Project No. DST/PAC(6)/96-ET.

References

1. Back, T., Fogel, D., and Michalewicz, Z. (Eds.) (1997). *Handbook of Evolutionary Computation*. New York: Institute of Physics Publishing and Oxford University Press.
2. Bagley, J. D. (1967). The behavior of adaptive systems which employ genetic and correlation algorithms (Doctoral dissertation, University of Michigan). *Dissertation Abstracts International*, 28(12), 5106B. (University Microfilms No. 68-7556).
3. Bethke, A. D. (1981). Genetic algorithms as function optimizers (Doctoral dissertation, University of Michigan). *Dissertation Abstracts International*, 41(9), 3503B. (University Microfilms No. 8106101).
4. Bramlette, M. F. and Bouchard, E. E. (1991). Genetic algorithms in parametric design of aircraft. In L. Davis (Ed.), *Handbook of genetic algorithms* (pp. 109–123). New York: Van Nostrand Reinhold.
5. Callahan, K. J. and Weeks, G. E. (1992). Optimum design of composite laminates using genetic algorithms. *Composites Engineering*, Vol. 2, No. 3, pp. 149–160.

6. Cavicchio, D. J. (1970). *Adaptive search using simulated evolution*. Unpublished doctoral dissertation, University of Michigan, Ann Arbor, (University Microfilms No. 25-0199).
7. Chaturvedi, D., Deb, K., and Chakrabarty, S. K. (1995). Structural optimization using real-coded genetic algorithms. In P. K. Roy and S. D. Mehta (Eds.), *Proceedings of the Symposium on Genetic Algorithms* (pp. 73–82).
8. Davis, L. (1991). *Handbook of genetic algorithms*. New York: Van Nostrand Reinhold.
9. Davis, T. E. and Principe, J. C. (1991). A simulated annealing-like convergence theory for the simple genetic algorithm. In R. K. Belew, & L. B. Booker (Eds.), *Proceedings of the Fourth International Conference on Genetic Algorithms* (pp. 174–181).
10. Dawkins, R. (1986). *The Blind Watchmaker*. New York: Penguin Books.
11. Dawkins, R. (1976). *The Selfish Gene*. New York: Oxford University Press.
12. De Jong, K. A. (1975). An analysis of the behavior of a class of genetic adaptive systems. (Doctoral dissertation, University of Michigan). *Dissertation Abstracts International* 36(10), 5140B.
13. Deb, K. (1996). Genetic algorithms for function optimization. In F. Herrera and J. L. Verdegay (Eds.) *Genetic Algorithms and Soft Computing*. Heidelberg: Physica-Verlag.
14. Deb, K. (1995). *Optimization for engineering design: Algorithms and examples*. Delhi: Prentice-Hall.
15. Deb, K. (1993). Genetic algorithms in optimal optical filter design. In E. Balagurusamy and B. Sushila (Eds.), *Proceedings of the International Conference on Computing Congress* (pp. 29–36).
16. Deb, K. (1989). Genetic algorithms in multimodal function optimization, *Master's Thesis*, (TCGA Report No. 89002). Tuscaloosa: University of Alabama.
17. Deb, K. and Agrawal, R. B. (1995) Simulated binary crossover for continuous search space. *Complex Systems*, 9 115–148.
18. Deb, K. and Goldberg, D. E. (1989). An investigation of niche and species formation in genetic function optimization, *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 42-50.
19. Deb, K. and Kumar, A. (1995). Real-coded genetic algorithms with simulated binary crossover: Studies on multimodal and multiobjective problems. *Complex Systems*, 9(6), 431–454.
20. Deb, K. and Saxena, V. (1997). Car suspension design for comfort using genetic algorithms. In Thomas Back (Ed.) *Proceedings of the Seventh International Conference on Genetic Algorithms* (pp. 553–560).
21. Duffin, R. J., Peterson, E. L., and Zener, C. (1967). *Geometric Programming*. New York: Wiley.
22. Eldredge, N. (1989). *Macro-evolutionary Dynamics: Species, niches, and adaptive peaks*. New York: McGraw-Hill.
23. Eshelman, L. and Schaffer, J. D. (1993). Real-coded genetic algorithms and interval-schemata. To appear in *Foundations of Genetic Algorithms II*.

24. Fonseca, C. M. and Fleming P. J. (1993). Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In S. Forrest (Ed.), *Proceedings of the Fifth International Conference on Genetic Algorithms* (pp. 416–423).
25. Gen, M. and Cheng, R. (1997). *Genetic Algorithms and Engineering Design*. New York: Wiley.
26. Glover, F. and Laguna, M. (1997). *Tabu search*. New York: Kluwer Academic Publishers.
27. Goldberg, D. E. (1993). A Wright-Brothers theory of genetic-algorithm flight. *Journal of SICE*, 37(8), 450–458.
28. Goldberg, D. E. (1990). Real-coded genetic algorithms, virtual alphabets, and blocking. *Complex Systems*, 5(2), 139–168. (Also IlliGAL Report 90001).
29. Goldberg, D. E. (1989). *Genetic algorithms in search, optimization, and machine learning*. New York: Addison-Wesley.
30. Goldberg, D. E. (1983). Computer-aided gas pipeline operation using genetic algorithms and rule learning (Doctoral dissertation, University of Michigan). *Dissertation Abstracts International*, 44(10), 3174B. (University Microfilms No. 8402282).
31. Goldberg, D. E. (1985). Optimal initial population size for binary-coded genetic algorithms. *TCGA Report No. 850001*. The Clearinghouse for Genetic Algorithms. Tuscaloosa: University of Alabama.
32. Goldberg, D. E. and Deb, K. (1991). A comparison of selection schemes used in genetic algorithms, *Foundations of Genetic Algorithms*, edited by G. J. E. Rawlins, pp. 69–93.
33. Goldberg, D. E., Deb, K., and Clark, J. H. (1992). Genetic algorithms, noise, and the sizing of populations. *Complex Systems*, Vol. 6, pp. 333–362.
34. Goldberg, D. E., Deb, K., and Thierens, D. (1991). Toward a better understanding of mixing in genetic algorithms. *Journal of SICE*, Vol. 32, No. 1.
35. Goldberg, D. E. and Richardson, J. (1987). Genetic algorithms with sharing for multimodal function optimization. *Proceedings of the Second International Conference on Genetic Algorithms*, 41–49.
36. Goldberg, D. E. and Smith, R. (1987). Nonstationary function optimization using genetic algorithms with dominance and diploidy. In J. J. Grefenstette (Ed.) *Proceedings of the Second International Conference on Genetic Algorithms*. New Jersey: Lawrence Erlbaum Associates. (pp. 59–68).
37. Guest, J. J. (1925). the main free vibrations of an autocar. *Proc. Inst. Auto. Engrs. (London)*, 20(505).
38. Harik, G., Cantú-Paz, E., Goldberg, D. E., and Miller, B. (1996). The gambler’s ruin problem, genetic algorithms, and sizing of populations (IlliGAL Report No. 96004). Urbana: University of Illinois at Urbana-Champaign. Illinois Genetic Algorithms Laboratory.
39. Herrera, F. and Verdegay, J. L. (Eds.) (1996). *Genetic Algorithms and Soft Computing*. Heidelberg: Physica-Verlag.
40. Holland, J. H. (1975). *Adaptation in natural and artificial systems*. Ann Arbor: University of Michigan Press.

41. Horn J. and Nafpliotis N. (1993). Multiobjective optimization using niched Pareto genetic algorithms (IlliGAL Report No 93005). Urbana: University of Illinois at Urbana-Champaign. Illinois Genetic Algorithms Laboratory.
42. ISO 2631/1 (1985). Evaluation of human exposure to whole-body vibration – Part 1: General requirements. International Organization for Standardization.
43. Jain, L. C. and Jain, R. K. (Eds.) (1997). *Hybrid Intelligent Engineering Systems*. Advances in Fuzzy Systems–Applications and Theory (Vol 11). Singapore: World Scientific.
44. Jang, J.-S. R., Sun, C.-T., and Mizutani, E. (1997). *Neuro-Fuzzy and Soft Computing: A computational approach to learning and machine intelligence*. New Jersey: Prentice-Hall.
45. Karr, C. (1991). *Design of an adaptive fuzzy logic controller using a genetic algorithm*. In R. K. Belew and L. B. Booker (Eds.) *Proceedings of the Fourth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann. (pp. 450–457).
46. Kelly, J. D. and Davis, L. (1991). Hybridizing the genetic algorithm and the K nearest neighbors. In R. Belew and L. B. Booker (Eds.) *Proceedings of the Fourth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann. (pp. 377–383).
47. Kirpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220(4598). 671–680.
48. Krishnakumar, K. (1989). Microgenetic algorithms for stationary and non-stationary function optimization. *SPIE Proceedings on Intelligent Control and Adaptive Systems*, 1196, 289–296.
49. Lucasius, C. B. and Kateman, G. (1989). Application of genetic algorithms in chemometrics. In J. D. Schaffer (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 170–176).
50. Markine, V. L., Meijers, P. and Meijaard, J. P. (1996). Optimization of the dynamic response of linear mechanical systems using a multipoint approximation technique. In D. Bestle and W. Schiehlen (Eds.) *Proceedings of the IUTAM Symposium on Optimization of Mechanical Systems*. pp. 189–196.
51. McClelland, J. L. and Rumelhart, D. E. (1988) *Parallel Distributed Processing, Vol 1 and 2*. Cambridge: MIT Press.
52. Michalewicz, Z. and Schoenauer, M. (1996). Evolutionary algorithms for constrained parameter optimization problems. *Evolutionary Computation*, 4(1), 1–32.
53. Miller, G. F., Todd, P. M., and Hegde, S. U. (1989). Designing neural networks using genetic algorithms. *Proceedings of the Third International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann. (pp. 379–384).
54. Mitchell, M. (1996). *Introduction to Genetic Algorithms*. Ann Arbor: MIT Press.
55. Powell, D. and Skolnick, M. M. (1993). Using genetic algorithms in engineering design optimization with nonlinear constraints. In S. Forrest (Ed.) *Proceedings of the Fifth International Conference on Genetic Algorithms*, San Mateo, CA: Morgan Kaufmann (pp. 424–430).
56. Rajeev, S. and Krishnamoorthy, C. S. (1992). Discrete optimization of structures using genetic algorithms. *Journal of Structural Engineering*, Vol. 118, No. 5, pp. 1233–1250.

57. Reklaitis, G. V., Ravindran, A., and Ragsdell, K. M. (1983). *Engineering optimization methods and applications*. New York: John Wiley and Sons.
58. Rudolph, G. (1994). Convergence analysis of canonical genetic algorithms. *IEEE Transactions on Neural Network*. (pp 96–101).
59. Sanchez, E., Shibata, T., and Zadeh, L. A. (Eds.) (1997). *Genetic Algorithms and Fuzzy Logic Systems*. Advances in Fuzzy Systems–Applications and Theory (Vol 7). Singapore: World Scientific.
60. Sandgren, E. and Jensen, E. (1990). Topological design of structural components using genetic optimization methods. *Proceedings of the 1990 Winter Annual Meeting of the ASME*, AMD-Vol. 115.
61. Schaffer, J. D., Caruana, J. D., Eshelman, L. J., and Das, R. (1989). A study of control parameters affecting online performance of genetic algorithms for function optimization. *Proceedings of the Third International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann. (pp. 51–60).
62. Spears, W. M. and De Jong, K. A. (1991). An analysis of multi-point crossover. In G. J. E. Rawlins (Eds.), *Foundations of Genetic Algorithms* (pp. 310–315). (Also AIC Report No. AIC-90-014).
63. Srinivas, N. and Deb, K. (1995). Multiobjective function optimization using nondominated sorting genetic algorithms, *Evolutionary Computation*, 2(3), 221–248.
64. Starkweather, T., McDaniel, S., Mathias, K., Whitley, D., and Whitley, C. (1991). A comparison of genetic scheduling operators. In R. Belew and L. B. Booker (Eds.) *Proceedings of the Fourth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann. (pp. 69–76).
65. Syswerda, G. (1989). Uniform crossover in genetic algorithms. In J. D. Schaffer (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 2–9).
66. Thierens, D. and Goldberg, D. E. (1993). Mixing in genetic algorithms. In S. Forrest (Ed.) *Proceedings of the Fifth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann. (pp. 38–45).
67. Vose, M. D. (1990). Generalizing the notion of schema in genetic algorithms. *Artificial Intelligence*.
68. Vose, M. D., and Liepins, G. E. (1991). Punctuated equilibria in genetic search. *Complex Systems*, 5(1). 31–44.
69. Winter, G., Périaux, J., Galan, M., and Cuesta, P. (Eds.) (1996). *Genetic Algorithms in Engineering and Computer Science*. Chichister: Wiley.
70. Whitley, D. (1992). An executable model of a simple genetic algorithm. In D. Whitley (Ed.), *Foundations of Genetic Algorithms II*. (pp. 45–62).