An Introduction to Genetic Algorithms

Kalyanmoy Deb
Kanpur Genetic Algorithms Laboratory (KanGAL)
Department of Mechanical Engineering
Indian Institute of Technology Kanpur
Kanpur, PIN 208 016, India
E-mail: deb@iitk.ernet.in

Abstract

Genetic algorithms (GAs) are search and optimization tools, which work differently compared to classical search and optimization methods. Because of their broad applicability, ease of use, and global perspective, GAs have been increasingly applied to various search and optimization problems in the recent past. In this paper, a brief description of a simple GA is presented. Thereafter, GAs to handle constrained optimization problems are described. Because of their population approach, they have also been extended to solve other search and optimization problems efficiently, including multimodal problems, multiobjective problems, scheduling problems, and fuzzy-GA and neuro-GA implementations. The purpose of this paper is to familiarize readers to the concept of GAs and their scope of application.

Keywords: Genetic algorithms, Optimization, Optimal design, Nonlinear programming.

1 Introduction

Over the last decade, genetic algorithms (GAs) have been extensively used as search and optimization tools in various problem domains, including sciences, commerce, and engineering. The primary reasons for their success are their broad applicability, ease of use, and global perspective [19].

The concept of a genetic algorithm was first conceived by John Holland of the University of Michigan, Ann Arbor. Thereafter, he and his students have contributed much to the development of the field. Most of the initial research works can be found in several conference proceedings. However, now there exist several text books on GAs [19, 25, 29, 31, 18]. A more comprehensive description of GAs along with other evolutionary algorithms can be found in the recently compiled 'Handbook on Evolutionary Computation' published by Oxford University Press ([1]). Two journals entitled 'Evolutionary Computation' published by MIT Press and IEEE are now dedicated to promote the research in the field. Besides, most GA applications can also be found in domain-specific journals.

In this paper, we describe the working principle of GAs. A number of extensions to the simple GA to solve various other search and optimization problems are also described. Particularly, real-coded GAs, GAs to solve multimodal problems, GAs to solve multiobjective problems, GAs to solve scheduling problems, and Fuzzy-Neuro-GA implementations are described.

2 Classical Search and Optimization Methods

Traditional optimization methods can be classified into two distinct groups: direct and gradient-based methods [8]. In direct search methods, only objective function ($f(x)$) and constraint values ($g_j(x)$, $h_k(x)$) 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 also 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:

- The 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 optimization problem may not be efficient in solving a different optimization 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 engineering design problems, the search space may have more than one optimal solutions, of which most are undesired 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.

Every traditional optimization algorithm is designed to solve a specific type of problems. For example, the geometric programming method is designed to solve only posynomial-type objective function and constraints. Geometric programming is efficient in solving such problems but can not be applied suitably to solve other types of functions. The conjugate direction or conjugate gradient methods have convergence proofs for solving quadratic objective functions having one optimal solution, but they are not expected to work well in problems having multiple optimal solutions. The Frank-Wolfe's successive linear programming method [32] works efficiently on linear-like function and constraints, but for solving nonlinear problems its performance largely depends on the chosen initial conditions. Thus, one algorithm may be best suited for one problem and may not even be applicable to a different problem. This requires designers to know a number of optimization algorithms to solve different design problems.

In most engineering designs, some problem variables are restricted to take discrete values only. This requirement often arises to meet the market conditions. For example, if the diameter of a mechanical component is a design variable and the component is likely to be procured off-the-self, the optimization algorithm cannot use any arbitrary diameter. An usual practice to tackle such problems is to assume that all variables are continuous during the optimization process. Thereafter, an available size closer to the obtained solution is recommended. But, there are major difficulties with this approach. Firstly, since infeasible values of a design variable are allowed in the optimization process, the optimization algorithm spends enormous time in computing infeasible solutions (in some cases, it may not be possible to compute an infeasible solution). This makes the search effort inefficient. Secondly, as post-optimization calculations, the nearest lower and upper available sizes are to be checked for each infeasible discrete variable. For N such discrete variables, a total of 2^N additional solutions need to be evaluated. Thirdly, two options checked for each variable may not guarantee to form the optimal combination with respect to other variables. All these difficulties can be eliminated if only feasible values of the variables are allowed during the optimization process.

Many optimization problems require the use of a simulation software involving finite element method, computational fluid mechanics approach, nonlinear equation solving, or other computationally extensive methods to compute the objective function and constraints. Because of the affordability and availability of parallel computing machines, it becomes now convenient to use parallel machines in solving complex engineering design 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 as efficient optimization algorithms for engineering design. In the following section, we describe GA technique which can alleviate some of the above difficulties and may constitute an efficient optimization tool.

3 Genetic Algorithms

As the name suggests, genetic algorithms (GAs) borrow its working principle from natural genetics. In this section, we describe the principle of GA's operation. To illustrate the working of GAs better, we also show a hand-simulation of one iteration of GAs. Theoretical underpinnings describing why GAs qualify as robust search and optimization methods are discussed next.

3.1 Working principles

GAs are search and optimization procedures that are motivated by the principles of natural genetics and natural selection. Some fundamental ideas of genetics are borrowed and used artificially to construct search algorithms that are robust and require minimal problem information.

The working principle of GAs are very different from that of most of classical optimization techniques. We describe the working of a GA by illustrating on a simple can design problem. A cylindrical can is considered to have only two[1] design parameters—diameter d and height h. Let us consider that the can needs to have a volume of at least 300 ml and the objective of the design is to minimize the cost of can material. With these constraint and objective, we first write the corresponding nonlinear

[1]It is important to note that many other parameters such as thickness of can, material properties, shape can also be considered, but it will suffice to have two parameters to illustrate the working of a GA.

programming problem (NLP):

$$
\begin{array}{rl}
\text{Minimize} & f(d,h) = c\left(\frac{\pi d^2}{2} + \pi d h\right), \\
\text{Subject to} & g_1(d,h) \equiv \frac{\pi d^2 h}{4} \geq 300, \\
\text{Variable bounds} & d_{\min} \leq d \leq d_{\max}, \\
 & h_{\min} \leq h \leq h_{\max}.
\end{array}
\tag{1}
$$

The parameter c is the cost of can material per squared cm, and diameter d and height h are allowed to vary in [$d_{\min}$, $d_{\max}$] and [$h_{\min}$, $h_{\max}$] cm, respectively.

3.1.1 Representing a solution

In order to use GAs to find the optimal parameter values of d and h which satisfies the constraint g_1 and minimizes f, we first need to represent the parameter values in binary strings. Let us assume that we shall use five bits to code each of the two design parameters d and h, thereby making the overall string length equal to 10. The following string represents a can of diameter 8 cm and height 10 cm:

$$\underbrace{\texttt{01000}}_{d}\ \underbrace{\texttt{01010}}_{h}$$

This string and corresponding phenotype are shown in Figure 1. In the above representation, the

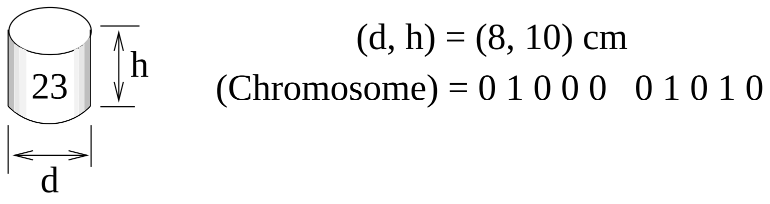

Figure 1: A typical can and its choromosomal representation are shown. The cost of the can is marked as 23 units.

lower and upper bounds of both parameters are considered to be zero and 31, respectively. With five bits to represent a parameter, there are exactly 2^5 or 32 different solutions possible. Choosing the lower and upper bounds like above allows GAs to consider only integer values in the range [0, 31]. However, GAs are not restricted to use only integer values in the above range, in fact GAs can be assigned to use any other integer or non-integer values just by changing the string length and lower and upper bounds:

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

where ℓ_i is the string length used to code i-th parameter and $\text{DV}(s_i)$ is the decoded value of the string s_i. In the above example, $\ell_i = 5$, $x_i^{\min} = 0$ and $x_i^{\max} = 31$ for both variables, such that $x_i = \text{DV}(s_i)$. The above mapping function allows following properties to achieve in the parameter values:

1. Any arbitrary (albeit finite) precision can be achieved in the parameter values by using a long enough string.

2. Different parameters can have different precision by simply using different string lengths.

3. Parameters are allowed to take positive and negative values.

Coding the parameters in a binary string is primarily used to have a pseudo-chromosomal representation of a design solution. For example, the 10-bit string illustrated above can be considered to show a biological representation of the can having 8 cm diameter and 10 cm height. Natural chromosomes are made of many genes, each of which can take one of many different allele values (such as, the gene responsible for the eye color in my chromosome is expressed as black, whereas it could have been blue or some other color). When you see me, you see my phenotypic representation, but each of my features is precisely written in my chromosome—the genotypic representation of me. In the can design problem, the can itself is the phenotypic representation of an artificial chromosome of 10 genes. To see how these 10 genes control the phenotype (the shape) of the can, let us investigate the leftmost bit (gene) of the diameter (d) parameter. A value of `0` at this bit (the most significant bit) allows the can to have diameter values in the range [0, 15] cm, whereas the other value `1` allows the can to have diameter values in the range [16, 31] cm. Clearly, this bit (or gene) is responsible to dictate the slimness of the can. If the allele value `0` is expressed, the can is slim and if the value `1` is expressed the can is fat. Each other bit position or combination of two or more bit positions can also be explained to have some feature of the can, but some are interesting and important and some are not that important. Now that we have achieved a string representation of design solution, we are ready to apply some genetic operations to such strings to hopefully find better and better solutions. But before we do that, we shall describe another important step of assigning a 'goodness' measure to each solution represented by a string.

3.1.2 Assigning fitness to a solution

It is important to reiterate that GAs work with strings representing design parameters, instead of parameters themselves. Once a string (or a solution) is created by genetic operators, it is necessary to evaluate the solution, particularly in the context of the underlying objective and constraint functions. In the absence of constraints, the fitness of a string is assigned a value which is a function of the solution's objective function value. In most cases, however, the fitness is made equal to the objective function value. For example, the fitness of the above can represented by the 10-bit string is

$$\begin{aligned} F(s) &= 0.0654\left(\pi\ (8)^2/2 + \pi\ (8)\ (10)\right), \\ &= 23, \end{aligned}$$

assuming $c = 0.0654$. Since the objective of the optimization is to minimize the objective function, it is to be noted that a solution with a smaller fitness value compared to another solution is better.

We are now in a position to describe the genetic operators that are the main part of the working of a GA. But before we do that let us look at the steps involved in a genetic algorithm. Figure 2

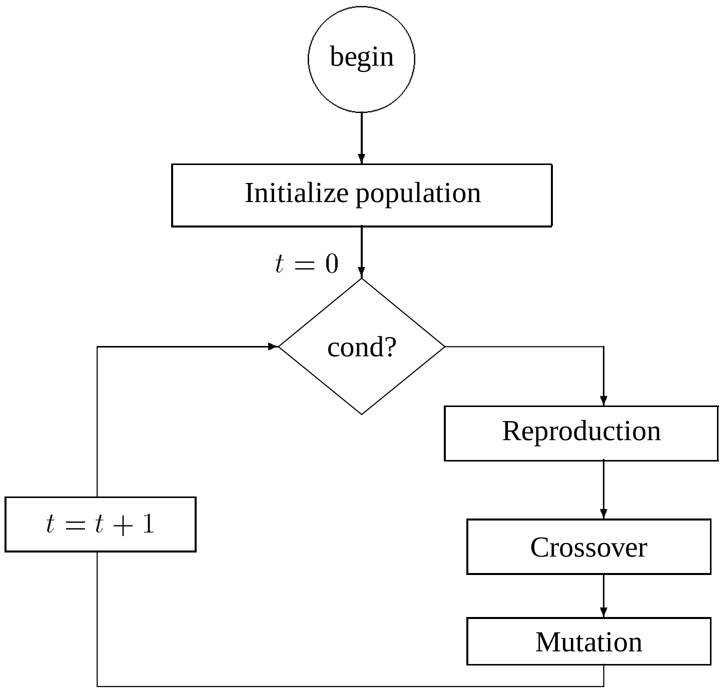

Figure 2: A flowchart of the working principle of a GA.

shows a flowchart of the working of a GA. Unlike classical search and optimization methods, a GA begins its search with a random set of solutions, instead of just one solution. Once a population of solutions (in the above example, a random set of binary strings) are created at random, each is evaluated in the context of the underlying NLP problem, as discussed above. A termination criterion is then checked. If the termination criterion is not satisfied, the population of solutions are modified by three main operators and a new (and hopefully better) population is created. The generation counter is incremented to indicate that one generation (or, iteration in parlance of classical search methods) of GA is completed. The flowchart shows that the working of a GA is simple and straightforward. We now discuss the genetic operators, in the light of the can design problem.

Figure 3 shows phenotypes of a random population of six cans. The fitness (penalized cost) of each can is marked on the can. It is interesting to note that two solutions do not have 400 ml volume inside and thus have been penalized by adding an extra artificial *cost*, a matter which is discussed a little later. Currently, it will suffice to note that the extra penalty cost is large enough to make all infeasible solutions to have worse fitness value than that of any feasible solution. We are now ready to apply three genetic operators, as follows.

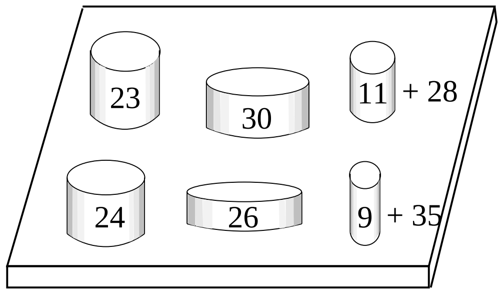

Figure 3: A random population of six cans is created.

3.1.3 Reproduction operator

The primary objective of the reproduction operator is to emphasize good solutions and eliminate bad solutions in a population, while keeping the population size constant. This is achieved by performing the following tasks:

1. Identify good (usually above-average) solutions in a population.
2. Make multiple copies of good solutions.
3. Eliminate bad solutions from the population so that multiple copies of good solutions can be placed in the population.

There exist a number of ways to achieve the above tasks. Some common methods are *tournament selection*, *proportionate selection*, *ranking selection*, and others [20]. In the following, we illustrate the binary tournament selection.

As the name suggests, tournaments are played between two solutions and the better solution is chosen and placed in a population slot. Two other solutions are picked again and another population slot is filled up with the better solution. If done systematically, each solution can be made to participate in exactly two tournaments. The best solution in a population will win both times, thereby making two copies of it in the new population. With a similar argument, the worst solution will loose in both tournaments and will be eliminated from the population. This, way any solution in a population will have zero, one, or two copies in the new population. It has been shown elsewhere [20] that the tournament selection has better convergence and computational time complexity properties compared to any other reproduction operator that exist in the literature, when used in isolation.

Figure 4 shows the six different tournaments played between old population members (each gets exactly two turns) shown in Figure 3. When cans with a cost of 23 units and 30 units are chosen at random for tournament, the can costing 23 unit is chosen and placed in the new population. Both cans are replaced in the old population and two cans are chosen for other tournaments in the next round. This is how the mating pool is formed and the new population shown in Figure 5 is created. It is interesting to note how better solutions (having lesser costs) have made themselves to have more than one copies in the new population and worse solutions have been eliminated from the population. This is precisely the purpose of a reproduction operator.

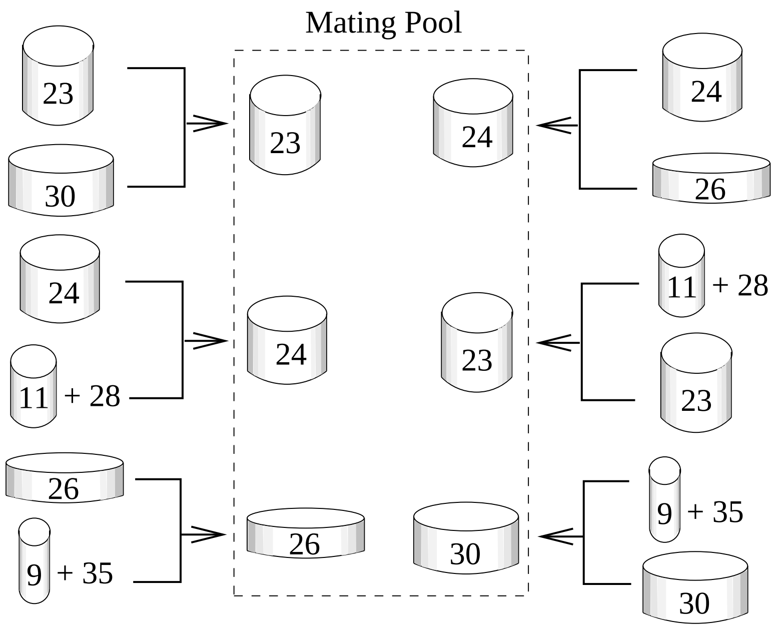

Figure 4: Tournaments played between six population members are shown. This forms the mating pool.

3.1.4 Crossover Operator

Crossover operator is applied next to the strings of the mating pool. A little thought will indicate that the reproduction operator cannot create any new solutions in the population. It only made more copies of good solutions at the expense of not-so-good solutions. Creation of new solutions is performed in crossover and mutation operators. Like reproduction operator, there exists a number of crossover operators in the GA literature [35], 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, this is performed by randomly choosing a crossing site along the string and by exchanging all bits on the right side of the crossing site.

Let us illustrate the crossover operator by picking two solutions (called parent solutions) from the new population created after reproduction operator. The cans and their genotype (strings) are shown in Figure 6. The third site along the string length is chosen at random and contents of the right side of this cross site are swapped between the two strings. The process creates two new strings (called children solutions). Their phenotypes (the cans) are also shown in the figure. Since a single cross site is chosen here, this crossover operator is called the *single-point* crossover operator.

It is important to note that in the above crossover operation we have been lucky and created a solution (22 units) which is better in cost than both parent solutions. One may wonder that if another cross site were chosen or two other strings were chosen for crossover, whether we had found a better child solution every time. A good point to ponder. It is true that every crossover between any two solutions from the new population is not likely to find children solutions better than parent solutions,

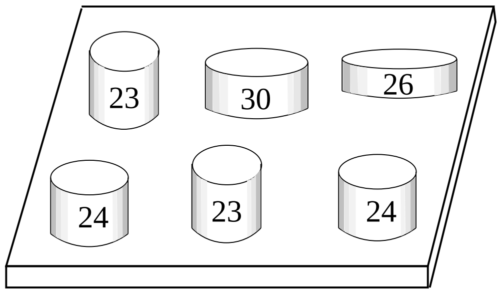

Figure 5: The population after reproduction operation. Good solutions are retained with multiple copies and bad solutions are eliminated.

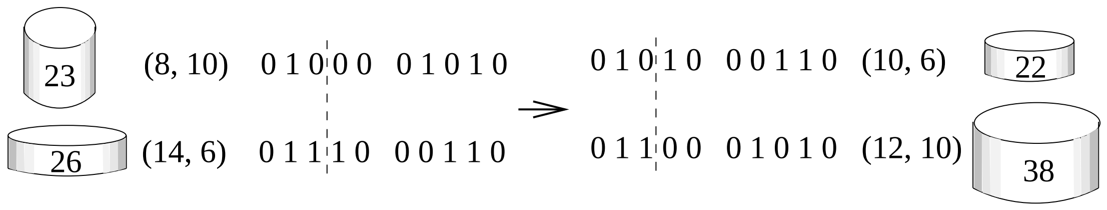

Figure 6: An illustration of the single-point crossover operator. Two parent solutions chosen from mating pool to create two new children solutions.

but it will be clear in a while that the chance of creating better solutions is far better than random. This is true because parent strings being crossed are not any two arbitrary random solutions, they have survived tournaments played with other solutions during the reproduction phase. Thus, they are expected to have some good bit combinations in their string representations. Since, a single-point crossover on a pair of parent strings can only create ℓ different string pairs with bit combinations from either strings, the created children solutions are also *likely* to be good strings. Moreover, every crossover may not create better solutions, but we do not worry about it too much. If bad solutions are created, they will get eliminated in the next reproduction operator and hence will have a short life. But let us think about the other possibility—when a good solution is created (and which has a better chance of happening). Since it is better, it is likely to get more copies in the next reproduction operator and it is likely to get more chances to do crossover with other good solutions. Thus, more solutions are likely to have similar chromosomes like it. This is exactly how biologists and evolutionists explain the formation of complex life forms from simple ones [15, 6, 5].

In order to preserve some good strings selected during the reproduction operator, not all strings in the population are used in crossover. 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[2].

[2]Even though top $(1-p_c)100\%$ of the current population can be copied deterministically to the new population, this is usually performed at random.

3.1.5 Mutation Operator

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. The need for mutation is to keep diversity in the population. Figure 7 shows how a string obtained after reproduction and crossover operators has been mutated to another string, representing a slightly different can. Once again, the solution obtained is

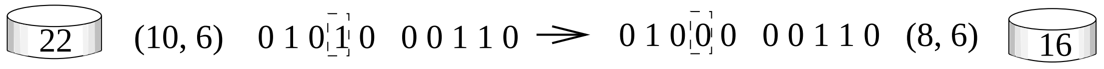

Figure 7: An illustration of the mutation operation. The fourth bit is mutated to create a new string.

better than that the original solution. Although, it may not happen all the times, mutating a string with a small probability is not a random operation since the process has a bias for creating a few solutions in the neighbourhood of the original solution.

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 during a GA generation, 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 see some intuitive reasoning to why GAs with these simple operators may constitute potential search algorithms.

3.2 Fundamental differences

As seen from the above description of GA's working principles, GAs are very different from most of the traditional optimization methods. The fundamental differences are described in the following paragraphs.

GAs work with a coding of variables instead of the variables themselves. Binary GAs work with a discrete search space, even though the function may be continuous. On the other hand, since function values at various discrete solutions are required, a discrete or discontinuous function may be tackled using GAs. This allows GAs to be applied to a wide variety of problem domains. Other advantage is that GA operators exploit the similarities in string-structures to make an effective search. We shall discuss more about this matter a little later. One of the drawbacks of using a coding is that a suitable coding must be chosen for proper working of a GA. Although it is difficult to know before hand what coding is suitable for a problem, a plethora of experimental studies [1] suggest that a coding which respects the underlying *building block* processing must be used.

The more striking difference between GAs and most of the traditional optimization method is that GAs work with a population of solutions instead of a single solution. Because there are more than one strings that are processed simultaneously and used to update any one string in the population, it is very likely that the expected GA solution may be a global solution. Even though some traditional algorithms are population-based like Box's algorithm [2], those methods do not use the obtained information efficiently. Moreover, since a population is what is updated at every generation a set of solutions (in case of multimodal optimization, multiobjective Pareto optimization, and others) can be obtained simultaneously, a matter we discuss in Section 5.

In discussing about GA operators or their working principles above, no word has been mentioned about the gradient or any other auxiliary problem information. In fact, GAs do not require any auxiliary information except the objective function values, although problem information can be used to speed up the GA's search process. The direct search methods used in traditional optimization also do not require gradient information explicitly, but in some of those methods search directions are found using the objective function values that are similar in concept to the gradient of the function. Moreover, some classical direct search methods work under the assumption that the function to be optimized is unimodal. GAs do not impose any such restrictions.

The other difference is that GAs use probabilistic rules to guide their search. On the face of it, this may look ad hoc, but a careful thinking may provide some interesting properties of this type of search. The basic problem with most of the traditional methods is that there are fixed transition rules to move from one solution to another solution. That is why those methods, in general, can only be applied to a special class of problems, where any solution in the search space leads to the desired optimum. Thus, these methods are not robust and simply can not be applied to a wide variety of problems. In trying to solve any other problem, if a mistake is made early on, since fixed rules are used it is very hard to recover from that mistake. GAs, on the other hand, use probabilistic rules and an initial random population. Thus, early on, the search may proceed in any direction and no major decision is made in the beginning. Later on, when population has converged in some locations the search direction narrows and a near-optimal solution is found. This nature of narrowing the search space as generation progresses is adaptive and is a unique characteristic of GAs. This characteristic of GAs also permits them to be applied to a wide class of problems giving them the robustness that is very useful in very sparse nature of optimization problems.

Another difference with most of the traditional methods is that GAs can be easily and conveniently used in parallel machines. By using tournament selection, where two strings are picked at random and the better string is copied in the mating pool, only two processors are involved at a time. Since any crossover operator requires interaction between only two strings, and since mutation requires alteration in only one string at a time, GAs are suitable for parallel machines. There is another advantage. Since in real-world design optimization problems, most computational time is spent in evaluating a solution, with multiple processors all solutions in a population can be evaluated in a distributed manner. This will reduce the overall computational time substantially.

Every good optimization method needs to balance the extent of exploration of the information obtained till the current time with the extent of exploitation of the search space required to obtain new and better solution(s). If the solutions obtained are exploited too much, premature convergence is expected. On the other hand, if too much stress is given on search, the information obtained thus far have 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 amount of exploration and exploitational consideration. For example, pattern search algorithms has a local exploratory search (the extent of which is fixed before hand) followed by a pattern search. The exploitation aspect comes only in the determination of search directions. Box's method [2] has almost no exploration consideration and hence is not very effective. In contrast, the exploitation and exploration aspects of GAs can be controlled almost independently. This provides a lot of flexibility in designing a GA.

3.3 Theory of GAs

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, a 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 [33, 39, 41]. Even though the operations are simple, GAs are highly nonlinear, massively multi-faceted, stochastic, and complex. There have been some studies using Markov chain analysis that involve deriving transition probabilities from one state to another and manipulating them to find the convergence time and solution. Since the number of possible states for a reasonable string length and population size become unmanageable even with high-speed computers available today, other analytical techniques may be used to predict the convergence of GAs. This is not to say that no such proof is possible for GAs nor to discourage the reader to pursue studies relating convergence of GAs, rather this is all the more mentioned here to highlight that more emphasis needs to be put in the study of GA convergence proofs.

In order to investigate why GAs may work, let us reconsider the one-cycle GA application to a numerical maximization problem.

$$\begin{array}{rl} \text{Maximize} & \sin(x) \\ \text{Variable bound} & 0 \le x \le \pi. \end{array} \tag{3}$$

We use five-bit strings to represent the variable x in the range $[0, \pi]$, so that the string (`00000`) represents $x = 0$ solution and the string (`11111`) represents $x = \pi$ solution. Other 30 strings are mapped in the range $[0, \pi]$. Let us also assume that we shall use a population of size four, proportionate selection[3], 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. The first string has a decoded value equal to 9 and this string corresponds to a solution $x = 0.912$, which has a function value equal to $\sin(0.912) = 0.791$. Similarly, other three stings 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 *proportionate* 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

[3]A string is given a copy in the mating pool proportionate to its fitness value [19]. One way to implement this operator is to mark a roulette-wheel's circumference for each solution in the population in proportion to the solution's fitness. Then, the wheel is spun N times (where N is the population size), each time picking the solution marked by the roulette-wheel pointer. This process makes the expected number of copies of a string having a fitness f_i picked for the mating pool equal to $f_i/\bar{f}$, where $\bar{f}$ is the average fitness of all strings in the population.

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.

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 stings 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.

The string copying and substring exchange are all very interesting and seems to improve the average performance of a population, but let us ask the question: What has been processed in one cycle of GA operators? If we investigate carefully we observe that among the strings of the two populations there are some similarities in string positions among the strings. By the application of three GA operators, the number of strings with similarities at certain string positions have been 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. Form table 1, we observe that there is only one string contained in this schema H_1 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 are 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. This is what we have accomplished in table 1 without having to count all these schema competitions and without the knowledge of the complete search space, but by manipulating only a few instances of the search space. 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 the *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 and finally converge to the optimal solution. 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*.

4 Constrained Optimization Using GAs

The above discussion of the can design problem avoided detailed consideration of constraints, instead a simple penalty term is used to penalize infeasible solutions. Let us discuss the difficulties of such a simple method and present an efficient way of handling constraints.

Typically, an optimal design problem having N variables is written as a nonlinear programming (NLP) problem, as follows:

$$\begin{array}{rll} \text{Minimize} & f(x) & \\ \text{Subject to} & g_j(x) \geq 0, & j = 1, 2, \ldots, J, \\ & h_k(x) = 0, & k = 1, 2, \ldots, K, \\ & x_i^{(l)} \leq x_i \leq x_i^{(u)}, & i = 1, 2, \ldots, N. \end{array} \tag{4}$$

In the above problem, there are J inequality and K equality constraints. The can design problem has two ($N = 2$) variables, one ($J = 1$) inequality constraint and no ($k = 0$) equality constraint. The simple penalty function method converts the above constrained NLP problem to an unconstrained minimization problem by penalizing infeasible solutions:

$$P(x, R, r) = f(x) + \sum_{j=1}^{J} R_j \langle g_j(x) \rangle^2 + \sum_{k=1}^{K} r_k [h_k(x)]^2. \tag{5}$$

The parameters R_j and r_k are the penalty parameters for inequality and equality constraints, respectively. The success of this simple approach lies in the proper choice of these penalty parameters. One thumb rule of choosing the penalty parameters is that they must be so set that all penalty terms are of comparable values with themselves and with the objective function values. This is intuitive because if the penalty corresponding to a particular constraint is very large compared to that of other constraints, the search algorithm emphasizes solutions that do not violate the former constraint. This way other constraints get neglected and search process gets restricted in a particular way. In most cases, most search methods prematurely converge to a suboptimal feasible or infeasible solution.

Since a proper choice of penalty parameters are the key aspect of the working of such a scheme, most researchers *experiment* with different values of penalty parameter values and find a set of reasonable values. In order to reduce the number of parameters, an obvious strategy often used is to normalize the constraints so that only one penalty parameter value can be used [8]. Consider the constraint $g_1(d,h)$ in the can design problem. After normalizing, this constraint can be written as follows:

$$g_1(x) \equiv \frac{\pi d^2 h}{4}/300 - 1 \geq 0. \tag{6}$$

so that the constraint violation is between $[-1, 0]$. If there were another constraint in the can design problem and the constraint were also normalized like g_1, then both constraints would have been emphasized equally. In such cases, an search and optimization method works much better if an appropriate penalty parameter value [12] is used.

Since GAs work with a population of solutions, instead of a single solution, a better penalty approach can be used. The penalty function approach also exploits the ability to have pair-wise comparison in tournament selection operator discussed earlier. During tournament selection, the following criteria are always enforced:

1. Any feasible solution will have a better fitness than any infeasible solution,
2. Two feasible solutions are compared only based on their objective function values.
3. Two infeasible solutions are compared based on the amount of constraint violations.

Figure 8 shows a unconstrained single-variable function $f(x)$ which has a minimum solution in the infeasible region. The fitness $F(x)$ of any infeasible or feasible solution is defined as follows:

$$F(x) = \begin{cases} f(x), & \text{if } g_j(x) \geq 0, \quad \forall j \in J, \\ f_{\max} + \sum_{j=1}^{J} \langle g_j(x) \rangle, & \text{otherwise.} \end{cases} \tag{7}$$

The parameter $f_{\max}$ is the maximum function value of all feasible solutions in the population. The objective function $f(x)$, constraint violation $\langle g(x) \rangle$, and the fitness function $F(x)$ are shown in the figure. It is important to note that $F(x) = f(x)$ in the feasible region. When tournament selection operator is applied to a such a fitness function $F(x)$, all three criteria mentioned above will be satisfied and there will be selective pressure towards the feasible region. The figure also shows how the fitness value of six arbitrary solutions will be calculated. Thus, under this constraint handling scheme, the fitness value of an infeasible solution may change from one generation to another, but the fitness value of a feasible solution will always be the same. Since the above constraint handling method assigns a hierarchy to infeasible solutions and tournament selection does not depend on the exact fitness values, instead their relative difference is important, any arbitrary penalty parameter will work the same. In fact, there is no need of any explicit penalty parameter. This is a major advantage of this constraint handling method. It is important to note that such a constraint handling scheme without the need of a penalty parameter is possible because GAs use a population of solutions and pair-wise comparison

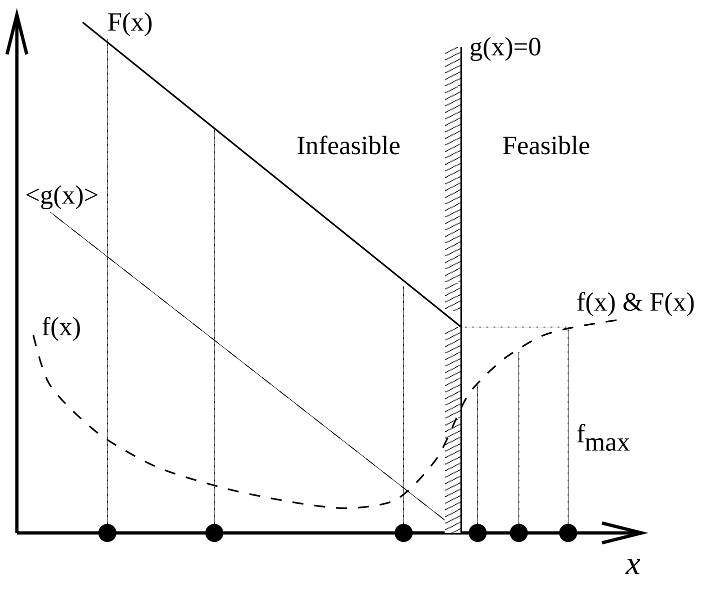

Figure 8: An efficient constraint handling scheme is illustrated. Six solid circles are solutions in a GA population.

of solutions is possible using the tournament selection. For the same reason, such schemes cannot be used with classical point-by-point search and optimization methods.

To show the efficacy of this constraint handling method, we apply GAs with this method to solve a two-variable, two-constraint NLP problem:

$$\begin{array}{ll} \text{Minimize} & f_1(x_1,x_2) = (x_1^2+x_2-11)^2 + (x_1+x_2^2-7)^2 \\ \text{Subject to} & g_1(x) \equiv 4.84 - x_1^2 - (x_2-2.5)^2 \geq 0 \\ & g_2(x) \equiv (x_1-0.05)^2 + (x_2-2.5)^2 - 4.84 \geq 0 \\ & 0 \leq x_1 \leq 6, 0 \leq x_2 \leq 6 \end{array} \tag{8}$$

The unconstrained objective function $f_1(x_1,x_2)$ has a minimum solution at (3,2) with a function value equal to zero. However, due to the presence of constraints, this solution is no more feasible and the constrained optimum solution is $x^* = (2.246826, 2.381865)$ with a function value equal to $f_1^* = 13.59085$. The feasible region is a narrow crescent-shaped region (approximately 0.7% of the total search space) with the optimum solution lying on the second constraint, as shown in Figure 9. GAs with a population of size 50 is run for 50 generations. No mutation is used here. Figure 9 shows how a typical GA run distributes solutions around the crescent-shaped feasible region and finally converges to a feasible solution very close to the true optimum solution.

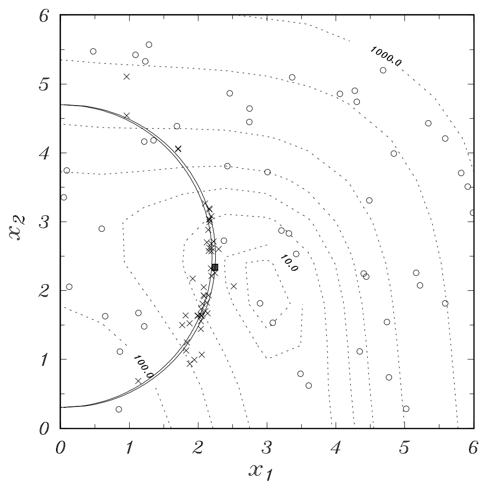

Figure 9: Population history at initial generation (marked with open circles), at generation 10 (marked with 'x') and at generation 50 (marked with open boxes) using the proposed scheme. The population converges to a solution very close to the true constrained optimum solution on a constraint boundary.

5 Advanced GAs

The simple GAs described above has been extended to solve different search and optimization problems. Some of them are listed and discussed in brief in the following:

Real-coded GAs: In Section 3 we have discussed binary GAs where variables are represented by a binary string. However, this is not always necessary and variables taking real values can be used directly. Although the same reproduction operator described here can be used, the trick lies in using efficient crossover and mutation operators [10, 13, 16]. The real-coded GAs eliminate the difficulties of achieving arbitrary precision in decision variables and the Hamming cliff problem [19] associated with binary string representation of real numbers.

The crossover is performed variable by variable. For crossing i-th variable between two parent solution vectors (having x_i^1 and x_i^2 values), the following procedure is used to create two new values (y_i^1 and y_i^2) using a probability distribution ($\mathcal{P}(\beta)$) described as follows [10]:

$$\mathcal{P}(\beta) = \begin{cases} 0.5(\eta+1)\beta^\eta, & \text{if } \beta \leq 1, \\ 0.5(\eta+1)\frac{1}{\beta^{\eta+2}}, & \text{otherwise,} \end{cases} \tag{9}$$

where the parameter β is the ratio of the difference in the children and parent solutions:

$$\beta = \left| \frac{y_i^1 - y_i^2}{x_i^1 - x_i^2} \right| . \tag{10}$$

The procedure of calculating children solutions are as follows:

1. Create a random number u between 0 and 1.
2. Find a $\bar{\beta}$ for which $\int_0^{\bar{\beta}} \mathcal{P}(\beta) d\beta = u$.
3. The children are calculated as

$$\begin{aligned} y_i^1 &= 0.5 \left[(x_i^1 + x_i^2) - \bar{\beta} |x_i^2 - x_i^1| \right], \\ y_i^2 &= 0.5 \left[(x_i^1 + x_i^2) + \bar{\beta} |x_i^2 - x_i^1| \right]. \end{aligned}$$

The probability distribution is chosen to produce near-parent solutions with a larger probability than solutions far away from parent solutions. The parameter η determines how close children solutions can become with respect to parent solutions. A small value of η makes children solutions to be away from parents and a large value allows children solutions to be close to parents. It has been observed that a value of $\eta = 2$ works well in most cases [10].

Since the probability term is defined with a non-dimensionalized parameter β, this crossover operator is adaptive in nature. Initially, when the population is randomly created, parents are likely to be far away from each other and this crossover operator has almost uniform probability to create any solution in the search space. On the other hand, after sufficient number of generations when population is converging to a narrow region, parents are closer to each other. This crossover will then not allow distant solutions to be created, instead will find solutions closer to parents, thereby allowing GAs to have arbitrary precision in solutions. The above distributions can also be extended for bounded variables. A discrete version of this probability distribution is also designed to tackle discrete search space problems [12]. Along this line, a real-coded mutation operator is also developed to find a perturbed child solution from a parent solution using a probability distribution [12].

Multimodal optimization: Many real-world problems contain multiple solutions that are optimal or near-optimal. The knowledge of multiple optimal solutions in a problem provides flexibility in choosing alternate yet equally good solutions as and when required. In trying to find more than one optimal solutions using traditional point-by-point methods, repeated application of the optimization algorithm with different initial points is required. This requires some knowledge of the *basin of attraction* of desired optima in the problem, otherwise many restarts may converge to the same optimum. Since, GAs work with a population points, a number of optimal solutions may be made to coexist in the population, thereby allowing to solve multiple optimal solutions simultaneously.

The idea of coexisting a number of optimal solutions in a population requires some change in the simple genetic algorithms described in the previous section. Borrowing the analogy of coexisting multiple niches in nature, we recognize that multiple niches (human and animals, for example) exist by sharing available resources (land and food, for example). Similar sharing concept is introduced artificially in GA population by *sharing functions* [9, 11, 22], which calculate an extent of sharing that needs to be performed between two strings. If the distance (could be some norm of the difference in decoded parameter values) between i-th and j-th strings is d_{ij}, usually a linear sharing function is used:

$$Sh(d_{ij}) = \begin{cases} 1 - \frac{d_{ij}}{\sigma}, & \text{if } d_{ij} < \sigma; \\ 0, & \text{otherwise.} \end{cases} \tag{11}$$

The parameter σ is the maximum distance between two strings for them to be shared and is fixed before hand [9]. The sharing enhancement to the simple GAs is as follows. For every string in the population, the sharing function value is calculated for other strings taken either from a sample of the population or from the whole population. These sharing function values are added together to calculate the niche count, $m_i = \sum_j Sh(d_{ij})$. Finally the shared fitness of the i-th string is calculated as $f_i' = f_i/m_i$ and this shared fitness is used in the reproduction operator instead of the objective function value f_i. Other operators are used as before. This allows coexistence of multiple optimal solutions (both local and global) in the population for the following reason. If in a generation, there exists fewer strings from one optimal solution in the population, the niche count for these strings will be smaller compared to strings from other optima and the shared fitness of these strings will be higher. Since the reproduction operator will now emphasize these strings from the extinct optima, there will suddenly be more strings from this optima in the population. This is how sharing would maintain instances of multiple optima in the population. GAs with this sharing strategy has solved a number of multimodal optimization problems, including a massively multimodal problem having more than five million local optima, of which only 32 are global optima [21].

Multi-objective optimization: In a multiobjective optimization problem, there are more than one objective functions, which are to be optimized simultaneously. Traditionally, the practice is to convert multiple objectives into one objective function (usually an weighted average of the objectives is used) and then treat the problem as a single objective optimization problem. Unfortunately, this technique is subjective to the user, with the optimal solution being dependent on the chosen weight vector. In fact, the solutions of the multiobjective optimization problem can be thought as a collection of optimal solutions obtained by solving different single objective function formed using different weight vectors. These solutions are known as *Pareto-optimal* solutions.

In order to find a number of Pareto-optimal solutions, different extensions of GAs are tried in the recent past [17, 26, 37]. Because of their population approach, GAs are ideal candidates to solve these kinds of problems. In one implementation of GAs, the concept of nondominated sorting of population members is used. We briefly describe that method in the following.

GAs require only one fitness value for an individual solution in the population. Thus, an artificial fitness value must be assigned to each solution in the population depending on the comparative values of each objective function. In order to assign a fitness measure to each solution, Srinivas and Deb [37] have borrowed Goldberg's [19] idea of nondomination among population members. In a population, the nondominated solutions are defined as those solutions which are better in at least one objective than any other solution in the population. In order to implement nondominated sorting concept, the following procedure is adopted:

- The population is sorted to find the nondominated set of solutions. All individual in this subpopulation is assigned a large artificial fitness value.
- Since the objective is to find a number of Pareto-optimal solutions, sharing procedure described earlier is performed among these nondominated solutions and a new shared fitness is calculated for each of these solutions.
- These solutions are temporarily counted out of the population and the next nondominated set is found. These solutions are assigned an artificial fitness value marginally smaller than the least shared fitness value in the previous nondominated set. This is done to impose a higher preference for solutions in the previous (and better) set than for the current set.
- Sharing is performed again among the new nondominated set and this process continues till all population members are ranked in descending order of the nondominated sets.
- Thereafter, the reproduction operation is performed with these artificial fitness values.
- Crossover and mutation operators are applied as usual.

This extension of GAs has been applied to solve a number of test problems [37] and a number of engineering design optimization problems [13, 36].

GAs in fuzzy logic controller design: 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 have been suitably used. Figure 10 shows typical membership func-

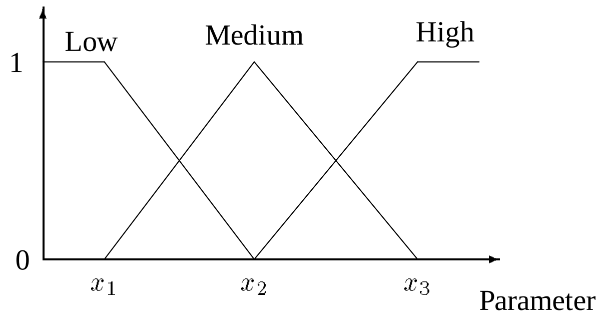

Figure 10: Fuzzy membership functions and typical variables used for optimal design

tions 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 [24, 27].

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 2 shows the corresponding rule base. Although this rule base may not be the optimal one, GAs can process a population of such rule

Table 2: Action variable for a string representing a fuzzy rule base shown in slanted fonts

		Temperature			
		Low	Medium	High	Don't Care
Humidity	Low	*High*	*Medium*		*Medium*
	Medium	*Low*		*Medium*	*Medium*
	High	*Medium*	*High*		
	Don't Care			*High*	

bases and finally find the optimal rule base. Once the rules present in the rule base is determined from the string, user-defined fixed 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 the 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. One such technique has been used to design a fuzzy logic controller for mobile robot navigation among dynamic obstacles [14].

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

GAs with neural networks: 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 [28] 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 back-propagation 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 neural network 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 [3, 34] or finding optimal networking problems. The standard search techniques used in those problems can also be used in optimal neural network design problems. A bit in a string can represent the existence `1` or absence `0` of a connection between two neurons. Thus, in a network having I input neurons, O output neurons, and one hidden layer having H neurons, the overall string length is $I \times H + H \times O + I \times O$. Biases in each neuron can also be considered. Each string, thus, represents one neural network configuration and a fitness can be assigned based on how close an output it finds compared to the desired output with a fixed number of epochs. Evolution of neural networks in this fashion has resulted in networks which were more efficient than what human designers could think of [30] 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 [40].

Searching for optimal schedules: 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 [4, 19, 38].

Non-stationary function optimization: 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 [23].

Acknowledgement

This paper is written while the author has been visiting the University of Dortmund, Germany on Alexander von Humboldt Fellowship. The author greatly acknowledges the support from Alexander von Humboldt Foundation.

References

[1] T. Bäck, D. Fogel and Z. Michalewicz, (Eds.) *Handbook of Evolutionary Computation*. Institute of Physics Publishing and Oxford University Press, New York. 1997.

[2] M. J. Box. A new method of constrained optimization and a comparison with other methods. *Computer Journal*. **8**, 42–52. 1965.

[3] D. Chaturvedi, K. Deb, and S. K. Chakrabarty. Structural optimization using real-coded genetic algorithms. In: *Proceedings of the Symposium on Genetic Algorithms*, Edited by P. K. Roy and S. D. Mehta, March 1995, pp. 73–82.

[4] L. Davis. *Handbook of genetic algorithms*. New York: Van Nostrand Reinhold. 1991.

[5] R. Dawkins. *The Blind Watchmaker*. New York: Penguin Books. 1986.

[6] R. Dawkins.. *The Selfish Gene*. New York: Oxford University Press. 1976.

[7] K. Deb. Genetic algorithms for function optimization. In: *Genetic Algorithms and Soft Computing*, Edited by F. Herrera and J. L. Verdegay. Heidelberg: Physica-Verlag. 1996.

[8] K. Deb. *Optimization for engineering design: Algorithms and examples*, Delhi: Prentice-Hall, 1995.

[9] K. Deb. Genetic algorithms in multimodal function optimization, *Master's Thesis*, (TCGA Report No. 89002). Tuscaloosa: University of Alabama. 1989.

[10] K. Deb. and R. B. Agrawal. Simulated binary crossover for continuous search space. *Complex Systems*, Volume 9, 1995, pp. 115–148.

[11] K. Deb and D. E. Goldberg. An investigation of niche and species formation in genetic function optimization, *Proceedings of the Third International Conference on Genetic Algorithms*, 1989, pp. 42–50.

[12] K. Deb and M. Goyal. A robust optimization procedure for mechanical component design based on genetic adaptive search, *ASME Journal of Mechanical Design*, (in press).

[13] K. Deb. and A. Kumar. Real-coded genetic algorithms with simulated binary crossover: Studies on multimodal and multiobjective problems. *Complex Systems*. Volume 9, No. 6, 1995, pp. 431–454.

[14] K. Deb, D. K. Pratihar, and A. Ghosh. Learning to avoid moving obstacles optimally for mobile robots using a genetic-fuzzy approach, *Parallel Problem Solving from Nature, V*, September, 1998.

[15] N. Eldredge. *Macro-evolutionary Dynamics: Species, niches, and adaptive peaks*, 1989, New York: McGraw-Hill.

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

[17] C. M. Fonseca and P. J. Fleming. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In S. Forrest (Ed.), *Proceedings of the Fifth International Conference on Genetic Algorithms*, 1993, pp. 416–423.

[18] M. Gen and R. Cheng. *Genetic Algorithms and Engineering Design*. New York: Wiley, 1997.

[19] D. E. Goldberg *Genetic algorithms in search, optimization, and machine learning*, New York: Addison-Wesley, 1989.

[20] D. E. Goldberg and K. Deb. A comparison of selection schemes used in genetic algorithms, In: *Foundations of Genetic Algorithms*, Edited by G. J. E. Rawlins, 1991, pp. 69–93.

[21] D. E. Goldberg, K. Deb, and J. Horn. Massive multimodality, deception, and genetic algorithms. In R. Manner and B. Manderick (Eds.), *Parallel Problem Solving from Nature II*, 1992, pp. 37–46.

[22] D. E. Goldberg and J. Richardson. Genetic algorithms with sharing for multimodal function optimization. *Proceedings of the Second International Conference on Genetic Algorithms*, 1987, pp. 41–49.

[23] D. E. Goldberg and R. Smith. Nonstationary function optimization using genetic algorithms with dominance and diploidy. In: *Proceedings of the Second International Conference on Genetic Algorithms*, Edited by J. J. Grefenstette, New Jersey: Lawrence Erlbaum Associates. 1987, pp. 59–68.

[24] F. Herrera and J. L. Verdegay (Editors). *Genetic Algorithms and Soft Computing*, Heidelberg: Physica-Verlag, 1996.

[25] J. H. Holland. *Adaptation in natural and artificial systems*, Ann Arbor: University of Michigan Press, 1975.

[26] J. Horn and N. Nafpliotis. Multiobjective optimization using niched Pareto genetic algorithms (IlliGAL Report No 93005). Urbana: University of Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory, 1993.

[27] C. Karr. *Design of an adaptive fuzzy logic controller using a genetic algorithm*. In: *Proceedings of the Fourth International Conference on Genetic Algorithms*, Edited by R. K. Belew and L. B. Booker, San Mateo, CA: Morgan Kaufmann, 1991, pp. 450–457.

[28] J. L. McClelland and D. E. Rumelhart. *Parallel Distributed Processing, Vol 1 and 2*, Cambridge: MIT Press, 1988.

[29] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*, Berlin: Springer-Verlag, 1992.

[30] G. F. Miller, P. M. Todd, and S. U. Hegde. Designing neural networks using genetic algorithms, *Proceedings of the Third International Conference on Genetic Algorithms*, San Mateo, CA: Morgan Kaufmann, 1991, pp. 379–384.

[31] M. Mitchell. *Introduction to Genetic Algorithms*. Ann Arbor: MIT Press, 1996 (Also New Delhi: Prentice-Hall).

[32] G. V. Reklaitis, A. Ravindran, and K. M. Ragsdell. *Engineering optimization methods and applications*. New York: John Wiley and Sons, 1983.

[33] G. Rudolph. Convergence analysis of canonical genetic algorithms. *IEEE Transactions on Neural Network*, 1994, pp 96–101.

[34] E. Sandgren and E. Jensen. Topological design of structural components using genetic optimization methods. *Proceedings of the 1990 Winter Annual Meeting of the ASME*, AMD-Vol. 115, 1990.

[35] W. M. Spears and K. A. De Jong. An analysis of multi-point crossover. In G. J. E. Rawlins (Eds.), *Foundations of Genetic Algorithms*, 1991, pp. 310–315. (Also AIC Report No. AIC-90-014).

[36] N. Srinivas. *Multiobjective optimization using nondominated sorting in genetic algorithms.* Masters thesis, Indian Institute of Technology, Kanpur, 1994.

[37] N. Srinivas and K. Deb. Multiobjective function optimization using nondominated sorting genetic algorithms, *Evolutionary Computation*, Volume 2, No. 3, 1995, pp. 221–248.

[38] T. Starkweather, S. McDaniel, K. Mathias, D. Whitley, and C. Whitley. A comparison of genetic scheduling operators. In: *Proceedings of the Fourth International Conference on Genetic Algorithms*,,Edited by R. Belew and L. B. Booker, San Mateo, CA: Morgan Kaufmann. 1991, pp. 69–76.

[39] M. D. Vose. Generalizing the notion of schema in genetic algorithms, *Artificial Intelligence*, 1990.

[40] G. Winter, J. Périaux, M. Galan, and P. Cuesta. (Editors) *Genetic Algorithms in Engineering and Computer Science.* Chichister: Wiley, 1996.

[41] D. Whitley. An executable model of a simple genetic algorithm. In D. Whitley (Ed.), *Foundations of Genetic Algorithms II*, 1992, pp. 45–62.