Multi-Objective Optimization Using Evolutionary Algorithms

First Edition

Kalyanmoy Deb
Professor, Department of Mechanical Engineering
Indian Institute of Technology Kanpur, India

JOHN WILEY & SONS
Chichester · New York · Brisbane · Toronto · Singapore

4

Evolutionary Algorithms

Evolutionary algorithms (EAs) mimic natural evolutionary principles to constitute search and optimization procedures. EAs are different from classical search and optimization procedures in a variety of ways. In this chapter, we will present a number of popular evolutionary algorithms by discussing their differences with classical methods.

Although this book is about multi-objective optimization, most multi-objective evolutionary algorithms modify single-objective evolutionary algorithms in special ways. Thus, an understanding of single-objective evolutionary algorithms will be essential in understanding the working principles of multi-objective evolutionary algorithms. Among different evolutionary algorithms, we will discuss in detail binary-coded and real-parameter genetic algorithms. Thereafter, we will present various procedures of evolution strategy. Finally, brief descriptions of evolutionary programming and genetic programming are presented. However, there exist other evolutionary and nature-inspired algorithms, such as ant-colony optimization, simulated evolution, DNA computing, and cultural algorithms, descriptions of which can all be found in the EA literature.

As discussed in the previous chapter, one of the two tasks in the ideal multi-objective optimization is to find widely spread solutions in the obtained non-dominated front. Finding and maintaining multiple solutions in one single simulation run is a unique feature of evolutionary optimization techniques. At the end of this chapter, we will also describe a number of methods to find multiple optimal solutions in multi-modal function optimization using evolutionary techniques. Some of these methods are directly used in the multi-objective evolutionary methods described in the subsequent chapters.

4.1 Difficulties with Classical Optimization Algorithms

In any growing field of research and application, it becomes difficult and dangerous to call any study 'classical'. In this book, we have loosely called all search and optimization algorithms that use a single solution update in every iteration and that mainly use a deterministic transition rule as classical methods. Many such optimization algorithms can be found in standard textbooks (Arora, 1989; Deb, 1995;

Fox, 1971; Haug and Arora, 1989; Himmelblau, 1972; Reklaitis et al., 1983). Interested readers can refer to these texts for a good understanding of some of those algorithms.

Most classical point-by-point algorithms use a deterministic procedure for approaching the optimum solution. Such algorithms start from a random guess solution. Thereafter, based on a pre-specified transition rule, the algorithm suggests a search direction, which is often arrived at by considering local information. A uni-directional search is then performed along the search direction to find the best solution. This best solution becomes the new solution and the above procedure is continued for a number of times. Figure 39 illustrates this procedure. Algorithms vary mostly in the way the search directions are defined at each intermediate solution.

Classical optimization methods can be classified into two distinct groups: direct methods and gradient-based methods (Deb, 1995). In direct search methods, only the objective function ($f(\mathbf{x})$) and the constraint values ($g_j(\mathbf{x})$, $h_k(\mathbf{x})$) 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 in the algorithm. On the other hand, gradient-based methods quickly converge near an optimal solution, but are not efficient in non-differentiable or discontinuous problems. In addition, there are some common difficulties with most classical direct and gradient-based techniques, as follows:

- 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 a discrete search space.

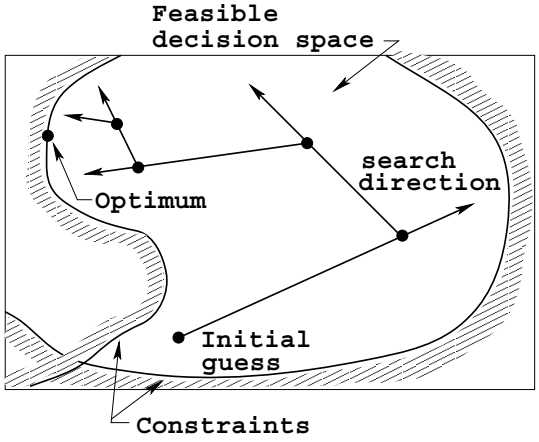


Figure 39 Most classical methods use a point-by-point approach.

- Algorithms cannot be efficiently used on a parallel machine.

Since nonlinearities and complex interactions among problem variables often exist in real-world optimization problems, the search space usually contains more than one optimal solution, of which most are undesired locally optimal solutions having inferior objective function values. While solving these problems, when classical methods get attracted to any of these locally optimal solutions, there is no escape.

Every classical optimization algorithm is designed to solve a specific type of problem. For example, the geometric programming method (Reklaitis et al., 1983) is designed to solve only *posynomial-type* objective function and constraints. Geometric programming is efficient in solving such problems but cannot be applied easily 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 (Reklaitis et al., 1983) works efficiently on linear 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 users to know a number of optimization algorithms in order to solve different optimization problems.

In most practical optimization problems, some decision 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 decision variable and the component is likely to be procured 'off-the-shelf', the optimization algorithm cannot use any arbitrary diameter. A 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. However, there are major difficulties with this approach. First, since infeasible values of a decision variable are allowed in the optimization process, the optimization algorithm spends enormous time in computing infeasible solutions (in some cases, it may not even be possible to evaluate an infeasible solution). This makes the search effort inefficient. Secondly, as post-optimization calculations, the nearest lower and upper available sizes need 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 formation of the optimal combination with respect to other variables. All of these difficulties can be eliminated if only feasible values of the variables are allowed during the optimization process.

Many real-world optimization problems require the use of simulation software involving the finite element method, the 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 systems, it has now become convenient to use them in solving complex real-world optimization problems. Since most classical methods use the point-by-point approach, where one solution gets updated to a new solution in one iteration,

the advantages of parallel systems cannot be fully exploited.

The above discussion suggests that classical methods without major fix-ups may face difficulties in solving practical optimization problems. In this chapter, we will describe a few evolutionary algorithms which may alleviate some of the above difficulties and which are increasingly replacing classical methods in practical problem solving.

4.2 Genetic Algorithms

Over the last decade, genetic algorithms (GAs) have been extensively used as search and optimization tools in various problem domains, including the sciences, commerce and engineering. The primary reasons for their success are their broad applicability, ease of use and global perspective (Goldberg, 1989).

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 this field. Most of the initial research work can be found in various International Conference Proceedings. However, there now exist several textbooks on GAs (Goldberg, 1989; Gen and Cheng, 1997; Holland, 1975; Michalewicz, 1992; Mitchell, 1996; Vose, 1999). A more comprehensive description of GAs, along with other evolutionary algorithms, can be found in the recently compiled 'Handbook on Evolutionary Computation' (Bäck et al., 1997). Three journals ('Evolutionary Computation Journal' published by MIT Press, 'Transactions on Evolutionary Computation' published by IEEE and 'Genetic Programming and Evolvable Machines' published by Kluwer Academic Publishers) are now dedicated to promote research in this field. In addition, most GA applications can also be found in various domain-specific journals.

In this section, we will first describe the working principle of a binary-coded GA. Thereafter, we will describe a real-parameter GA, which is ideally suited to handle problems with a continuous search space. Because of the population-approach of GAs, constraints can be handled in a much better way than the way in which they are handled in classical search and optimization algorithms. We will also describe a number of approaches for handling constraints in GAs.

4.2.1 Binary Genetic Algorithms

As the name suggests, genetic algorithms (GAs) borrow their working principle from natural genetics. In this section, we will describe the principles of a GA's operation. To illustrate the working of GAs better, we will also show a hand-simulation of one iteration of GAs on a two-variable problem. A theoretical description of GA parameter interactions and other salient issues are then presented.

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 is very different from that of most classical optimization techniques. We describe the working of a GA by illustrating a simple can design problem. A cylindrical can is considered to have only two parameters – the 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 the can material. With this constraint and the objective, we first write the corresponding nonlinear programming problem (NLP problem) (Deb, 1999b):

$$\left.\begin{array}{rl} \text{Minimize} & f(d,h) = c\left(\frac{\pi d^2}{2} + \pi dh\right), \\ \text{subject to} & g_1(d,h) \equiv \frac{\pi d^2 h}{4} \geq 300, \\ \text{Variable bounds} & d_{\min} \leq d \leq d_{\max}, \\ & h_{\min} \leq h \leq h_{\max}. \end{array}\right\} \tag{4.1}$$

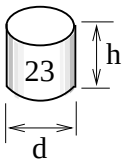
The parameter c is the cost of the can material per square cm, and the decision variables d and h are allowed to vary in $[d_{\min}, d_{\max}]$ and $[h_{\min}, h_{\max}]$ cm, respectively.

Representing a Solution

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

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

This string and corresponding decision variables are shown in Figure 40. In this representation, the lower and upper bounds of both decision variables are considered to be zero and 31, respectively. With five bits to represent a decision variable, there are exactly 2^5 or 32 different solutions possible. Choosing the lower and upper bounds as described above allows the GA to consider only integer values in the range [0, 31]. However, GAs are not restricted to use only integer values; in fact, GAs can be assigned to use any other integer or non-integer values just by changing the string



(d, h) = (8, 10) cm
(Chromosome) = 0 1 0 0 0 0 1 0 1 0

Figure 40 A typical can and its chromosomal representation are shown. The cost of the can is marked as 23 units. This figure is taken from Sādhanā (Deb, 1999b). Reproduced with permission from Indian Academy of Sciences, Bangalore.

length and lower and upper bounds:

$$x_i = x_i^{min} + \frac{x_i^{max} - x_i^{min}}{2^{\ell_i} - 1} \mathrm{DV}(s_i), \tag{4.2}$$

where ℓ_i is the string length used to code the i-th variable and $\mathrm{DV}(s_i)$ is the decoded value of the string s_i (where the complete string is $s = \cup_{i=1}^{n} s_i$). In the above example, $\ell_i = 5$, $x_i^{min} = 0$ and $x_i^{max} = 31$ for both variables, such that $x_i = \mathrm{DV}(s_i)$. The above mapping function allows the following features to exist in the decision variables:

1. Any arbitrary (albeit finite) precision can be achieved in the decision variables by using a long enough string.
2. Different decision variables can have different precisions by simply using different substring lengths.
3. Decision variables are allowed to take positive and negative values.
4. Variable bounds are taken care of by the mapping function given in equation (4.2).

Coding the decision variables in a binary string is primarily used to achieve a pseudo-chromosomal representation of a solution. For example, the 10-bit string illustrated above can be explained to exhibit a biological representation of a can having a diameter of 8 cm and a height of 10 cm. 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 a person's chromosomes may be expressed as black, whereas it could have been blue or some other color). When we see the person, we see the person's phenotypic representation, but each feature of the person is precisely written in his/her chromosomes – the genotypic representation of the person. 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 for dictating the 'slimness' of the can. If the allele value 0 is expressed, the can is slim, while if the value 1 is expressed, the can is 'fat'. Each other bit position or a combination of two or more bit positions can also be explained to support the can's phenotypic appearance, but some of these explanations are interesting and important, and some are not.

After choosing a string representation scheme and creating a population of strings at random, we are ready to apply genetic operations to such strings to hopefully find better populations of solutions. However, before we discuss the genetic operators used in GAs, we shall describe an intermediate step of assigning a 'goodness' measure to each solution represented by a string.

Assigning Fitness to a Solution

It is important to reiterate that binary GAs work with strings representing the decision variables, instead of decision variables themselves. Once a string (or a solution) is

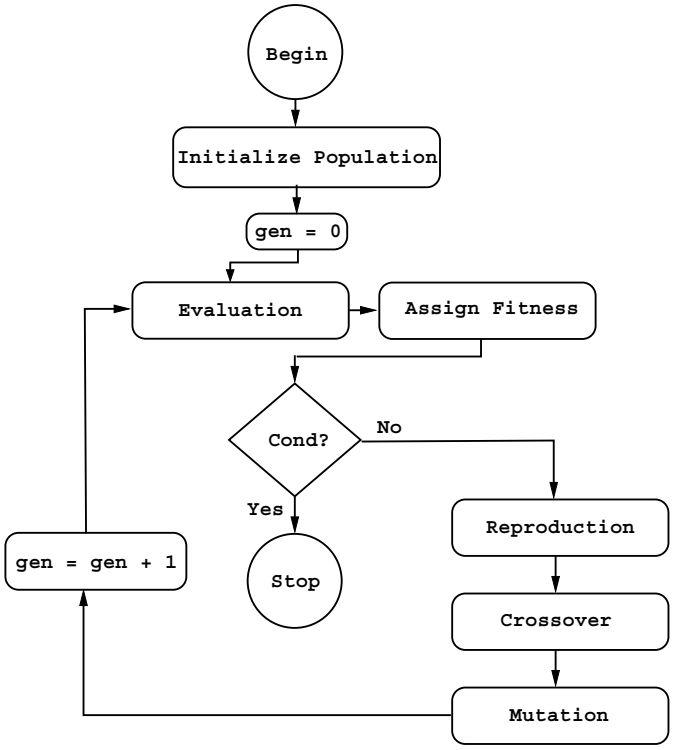


Figure 41 A flowchart of the working principle of a GA.

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 s is:

$$\begin{aligned} F(s) &= 0.065\left[\pi\,(8)^2/2 + \pi\,(8)\,(10)\right], \\ &= 23, \end{aligned}$$

assuming $c = 0.065$. Since the objective of the optimization here 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 which constitute the main part of a GA cycle. Figure 41 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 random population of solutions (in the above example, a random set of binary strings) is created, each is evaluated in the context of the underlying NLP problem (as discussed above) and a fitness is

assigned to each solution. The evaluation of a solution means calculating the objective function value and constraint violations. Thereafter, a metric must be defined by using the objective function value and constraint violations to assign a relative merit to the solution (called the *fitness*). A termination condition is then checked. If the termination criterion is not satisfied, the population of the solutions is 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, one iteration, in the parlance of classical search methods) of the GA is completed. The flowchart shows that the working of a GA is simple and straightforward. We will now discuss the genetic operators, in the light of the can design problem.

Figure 42 shows the phenotypes of a random population of six cans. The fitness (for feasible solutions, this is the cost term, while for infeasible solutions it is the cost plus the penalty in proportion to constraint violation) of each can is marked on the latter. It is interesting to note that two solutions do not have an internal volume of 300 ml and thus are penalized by adding an extra artificial *cost*, a matter which is discussed in Section 4.2.3 later. Currently, it will suffice to note that the extra penalized cost is large enough to cause all infeasible solutions to have a worse fitness value than that of any feasible solution. We are now ready to discuss three genetic operators.

Reproduction or Selection Operator

The primary objective of the reproduction operator is to make duplicates of 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 exists a number of ways to achieve the above tasks. Some common methods are *tournament selection*, *proportionate selection* and *ranking selection* (Goldberg

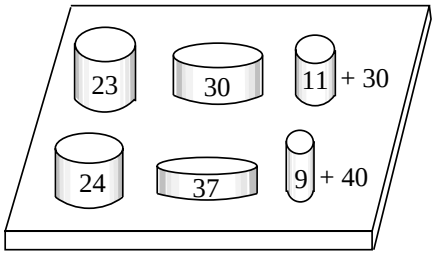


Figure 42 A random population of six cans. This figure is taken from Sādhanā (Deb, 1999b). Reproduced with permission from Indian Academy of Sciences, Bangalore.

and Deb, 1991).

In the tournament selection, tournaments are played between two solutions and the better solution is chosen and placed in the mating pool. Two other solutions are picked again and another slot in the mating pool is filled with the better solution. If carried out 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. Using a similar argument, the worst solution will lose in both tournaments and will be eliminated from the population. In this way, any solution in a population will have zero, one or two copies in the new population. It has been shown elsewhere (Goldberg and Deb, 1991) that the tournament selection has better or equivalent convergence and computational time complexity properties when compared to any other reproduction operator that exists in the literature.

Figure 43 shows six different tournaments played between old population members (each gets exactly two turns). When cans with a cost of 23 units and 30 units are chosen at random for the first tournament, the can costing 23 units wins and a copy of it is placed in the mating pool. The next two cans are chosen for the second tournament and a copy of the better can is then placed in the mating pool. This is how the mating pool (Figure 44) is formed. It is interesting to note how better solutions (having less costs) have made themselves to have multiple copies in the mating pool and worse solutions have been discarded. This is precisely the purpose of a reproduction or

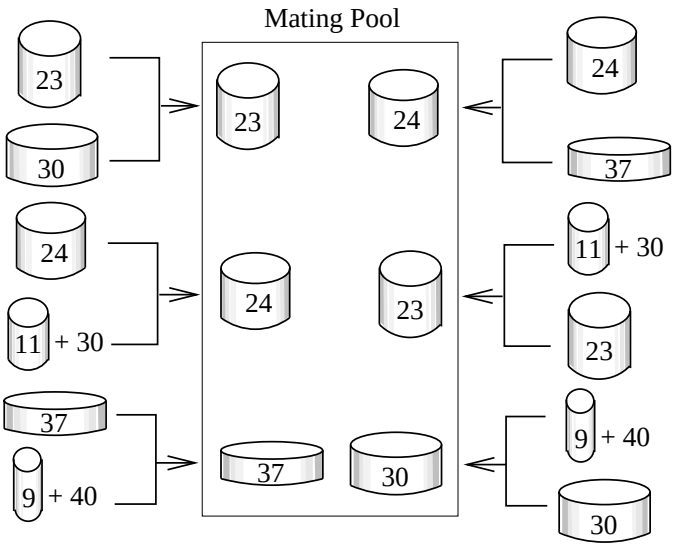


Figure 43 Tournaments are played between the six population members of Figure 42. The population enclosed by the dashed box forms the mating pool. This figure is taken from Sādhanā (Deb, 1999b). Reproduced with permission from Indian Academy of Sciences, Bangalore.

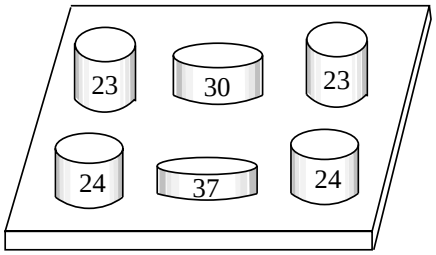


Figure 44 The population after reproduction operation. This figure is taken from Sādhanā (Deb, 1999b). Reproduced with permission from Indian Academy of Sciences, Bangalore.

a selection operator. An interesting aspect of the tournament selection operator is that just by changing the comparison operator, the minimization and maximization problems can be handled easily.

There exist a number of other selection operators. Before we discuss other genetic operators, let us describe a number of other popular selection operators. In the *proportionate selection* method, solutions are assigned copies, the number of which is proportional to their fitness values. If the average fitness of all population members is f_{avg}, a solution with a fitness f_i gets an expected f_i/f_{avg} number of copies. The implementation of this selection operator can be thought of as a roulette-wheel mechanism, where the wheel is divided into N (population size) divisions, where the size of each is marked in proportion to the fitness of each population member. Thereafter, the wheel is spun N times, each time choosing the solution indicated by the pointer, as shown in Figure 45. This figure shows a roulette wheel for five individuals having different fitness values. Since the third individual has a higher fitness value than any other, it is expected that the roulette wheel selection (RWS) will choose the third solution more often than any other solution. This RWS scheme can also be easily simulated on a computer. Using the fitness value F_i of all strings, the probability of selecting the i-th string is $p_i = F_i / \sum_{j=1}^{N} F_j$. Thereafter, the cumulative probability ($P_i = \sum_{j=1}^{i} p_j$) of each string can be calculated by adding the individual

Solution, i	F_i	p_i	P_i	$p_i N$
1	25.0	0.25	0.25	1.25
2	5.0	0.05	0.30	0.25
3	40.0	0.40	0.70	2.00
4	10.0	0.10	0.80	0.50
5	20.0	0.20	1.00	1.00

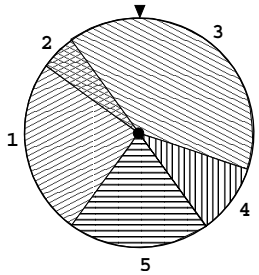


Figure 45 A roulette wheel is marked for five individuals according to their fitness values. The third individual has a higher probability of selection than any other.

probabilities from the top of the list. Thus, the bottom-most string in the population has a cumulative probability (P_N) equal to 1. The roulette-wheel concept can be simulated by realizing that the i-th string in the population represents the cumulative probability values in the range $[P_{i-1}, P_i]$. The first string represents the cumulative values from zero to P_1. In order to choose N strings, N random numbers between zero to one are created. Thus, a string that represents the chosen random number in the cumulative probability range (calculated from the fitness values) for the string is copied to the mating pool. Figure 46 shows the cumulative probability line (ranging from zero to one) and the corresponding ranges of each of the five solutions for the example problem outlined in Figure 45. A random number r_i shown by the pointer marks solution 3. Hence, solution 3 is selected as a member of the mating pool. In this way, the string with a higher fitness value represents a larger range of cumulative probability values and therefore has a higher probability of being copied into the mating pool. Since the computation of the average fitness requires the fitness of all population members, this selection operator is slow compared to the tournament selection method (Goldberg and Deb, 1991). Moreover, the roulette-wheel selection operator inherently maximizes the fitness function.

The above implementation of the proportionate selection is *noisy* in the sense of introducing a large variance in its realizations. The variance may be reduced by using a somewhat deterministic version of the above roulette-wheel selection operator. In the *stochastic remainder roulette-wheel* selection (SRWS) operator, the probabilities p_i are multiplied by the population size and the expected number of copies is calculated for all solutions. Thereafter, each solution is first assigned a number of copies equal to the integer part of the expected number. Thereafter, the usual roulette-wheel selection (RWS) operator is applied with the fractional part of the expected number of all solutions to assign further copies. Since a part of the assignment is deterministic, this operator is less noisy. For the example problem shown in Figure 45, the expected number of copies ($p_i N$) is also computed. Under the SRWS scheme, we first assign one copy each to solutions 1 and 5, two copies to solution 3, and no copies to solutions 2 and 4. The remaining one slot in the mating pool is filled by using a RWS operation on the entire population with the fractional values as fitness: (0.25, 0.25, 0.00, 0.50, 0.00).

Whether the basic roulette-wheel selection or the stochastic remainder roulette-wheel selection operator is used, $O(N)$ random numbers have to be created to complete the selection process. This computation can be reduced if a different version

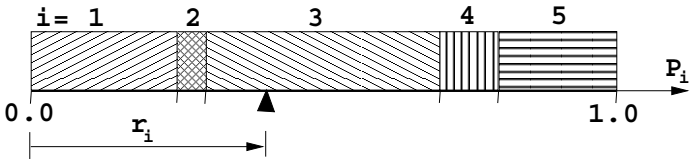


Figure 46 An implementation of the roulette-wheel selection operator.

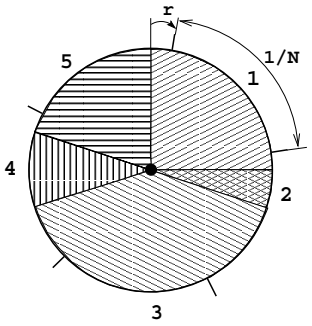


Figure 47 The stochastic universal sampling selection procedure.

– *stochastic universal sampling* (SUS) – is used (Baker, 1985). In this version, only one random number r is chosen for the whole selection process. Since N different solutions have to be chosen, a set of N equi-spaced numbers is created:

$$R = \{r, r + 1/N, r + 2/N, \ldots, r + (N-1)/N\} \mod 1.$$

Thereafter, a solution corresponding to each member of R is chosen from the cumulative probability values as before. Figure 47 shows the selection procedure for the same five solutions using the SUS method. Assuming that the periphery of the roulette wheel is of one unit, the first solution is chosen according to the location of the random number on the periphery. Other solutions are chosen by traversing identical distances (1/N) along the periphery of the roulette wheel. In the above case, solutions 1 and 3 get two copies each and solution 5 gets one copy after selection operation. Solutions 2 and 4 do not get any copies in the mating pool.

Besides having a large computational complexity, proportionate selection methods have a *scaling* problem. The outcome of this selection operator is dependent on the true value of the fitness, instead of the relative fitness values of the population members. For example, if in a population, one solution has a large fitness value compared to the rest of the population members, the probability of choosing this *super-solution* would be close to one, thereby dominating the mating pool with its copies. On the other hand, if in a population, all solutions have more or less the same fitness value, every solution has a similar probability of selection (p_i). This will lead to a single assignment of each solution in the mating pool. This phenomenon is equivalent to not performing the selection operation at all. In order to circumvent both of these difficulties, raw fitness values are mapped within a predefined range of scaled fitness values (Goldberg, 1989). However, the tournament selection does not have this scaling problem.

The scaling difficulty can also be avoided by using a *ranking* selection method. First, the solutions are sorted according to their fitness, from the worst (rank 1) to the best (rank N). Each member in the sorted list is assigned a fitness equal to the rank of the solution in the list. Thereafter, the proportionate selection operator is

applied with the ranked fitness values, and N solutions are chosen for the mating pool.

Crossover Operator

A 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 makes more copies of good solutions at the expense of not-so-good solutions. The creation of new solutions is performed by crossover and mutation operators. Like the reproduction operator, there exists a number of crossover operators in the GA literature (Spears, 1998), 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 to create two new 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 the reproduction operator. The cans and their genotype (strings) are shown in Figure 48. The third site along the string length is chosen at random and the contents of the right side of this cross site are exchanged between the two strings. The process creates two new strings (called offspring). Their phenotypes (the cans) are also shown in this 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 the above crossover operator created a solution (having a cost of 22 units) which is better in cost than both of the parent solutions. One may wonder that if a different cross site were chosen or two other strings were chosen for crossover, whether we would have found a better offspring every time. It is true that every crossover between any two solutions from the new population is not likely to find offspring better than both parent 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 the parent strings being crossed are not any two arbitrary random strings. These strings have survived tournaments played with other solutions during the earlier 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

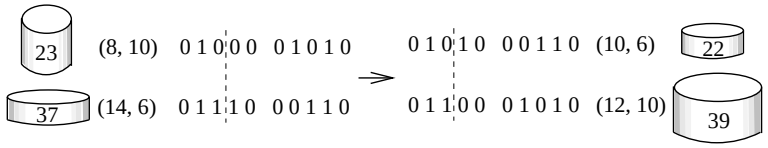


Figure 48 The single-point crossover operator. This figure is taken from Sādhanā (Deb, 1999b). Reproduced with permission from Indian Academy of Sciences, Bangalore.

can only create ℓ different string pairs (instead of all $2^{\ell-1}$ possible string-pairs) with bit combinations from either strings, the created offspring are also *likely* to be good strings. Moreover, every crossover may not create better solutions, but we will not worry too much about this here. If bad solutions are created, they will get eliminated in the next reproduction operation and hence will have a short life. However, let us think about the other possibility, that is, a good solution is created by the crossover operator. Since the offspring is good, it is likely to get more copies in the next reproduction operation and is also likely to get more chances to achieve a crossover with other good solutions in subsequent generations. Thus, more and more solutions in the population are likely to have similar chromosomes. This is exactly how biologists and evolutionists have explained how complex life forms have formed from simple ones (Dawkins, 1976, 1986; Eldredge, 1989).

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

The above concept of exchanging partial information between two strings can also be achieved with more than one cross sites. In a two-point crossover operator, two different cross sites are chosen at random. This will divide the string into three substrings. The crossover operation is completed by exchanging the middle substring between the strings, as shown in the following example:

Parents											Offspring								
0	1	0	0	0	0	1	0	1	0	$\rightarrow$	0	1	1	1	0	0	1	0	1	0
0	1	1	1	0	0	0	1	1	0		0	1	0	0	0	0	0	1	1	0

Extending this idea, one can also implement a n-point crossover, where n cross sites (odd or even) can be chosen. If an odd number of cross sites is chosen, the end of each string can be considered as an additional cross site, thereby making the total number of cross sites an even number. The crossover can then proceed by exchanging alternate substrings. The extreme of this process is the *uniform* crossover, where one offspring is constructed by choosing every bit with a probability p (usually $p = 0.5$ is used) from either parent, as shown in the following example:

Parents											Offspring									
0	1	0	0	0	0	1	0	1	0	$\rightarrow$	0	1	0	1	0	0	0	0	1	0
0	1	1	1	0	0	0	1	1	0		0	1	1	0	0	0	1	1	1	0

Usually, $p = 0.5$ is used. In the example problem, the second, fourth, fifth, seventh and ninth bits are exchanged between the parents. In terms of the extent of exploration (or search) power of a crossover operator, a single-point crossover preserves the structure of the parent strings to the maximum extent in the offspring. The extent of string-preservation reduces with the increase of cross sites in the crossover operator and is a minimum in the case of a uniform crossover operator. Spears (1998) has analyzed a number of these crossover operators and has found several interesting properties for many of them.

Mutation Operator

The crossover operator is mainly responsible for the search aspect of genetic algorithms, even though the mutation operator is also used for this purpose. The bit-wise mutation operator changes a 1 to a 0, and vice versa, with a mutation probability of p_m. The need for mutation is to keep diversity in the population. Figure 49 shows how a string obtained after the use of reproduction and crossover operators has been mutated to another string, thus representing a slightly different can. Once again, the solution obtained in the illustration is better than the original solution. Although it may not happen in all the instances of a mutation operation, mutating a string with a small probability is not a random operation since the process has a bias for creating only a few solutions in the search space.

The bit-wise mutation procedure described above requires the creation of a random number for every bit. In order to reduce the computational complexity, Goldberg (1989) suggested a *mutation clock* operator, where after a bit is mutated, the location of the next mutated bit is determined by an exponential distribution. The mean of the distribution is assumed to be $\mu = 1/p_m$. The procedure is simple. First, create a random number $r \in [0, 1]$. Then, estimate the next bit to be mutated by skipping $\eta = -p_m \ln(1 - r)$ bits from the current bit. In this way, on an average, $O(1/p_m)$ times less random numbers are to be generated. This mutation clock operator has been used in Deb and Agrawal (1999b).

The three operators – selection, crossover, and mutation operators – are simple and straightforward. The reproduction operator selects good strings, while the crossover operator recombines together good substrings from two good strings to hopefully form a better substring. The mutation operator alters a string locally to hopefully create a better string. Since none of these operations are performed deterministically, these claims are not guaranteed, nor explicitly tested, during a GA generation. However, it is expected that if bad strings are created they will be eliminated by the reproduction operator in subsequent generations and if good strings are created, they will be emphasized. Later, we shall see some intuitive reasoning as to why a GA with these simple operators constitutes a potential search and optimization algorithm.

Fundamental Differences

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

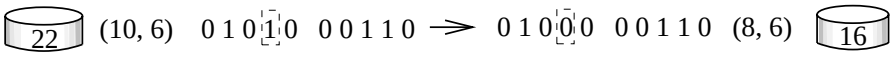


Figure 49 The bit-wise mutation operator. The fourth bit is mutated to create a new string. This figure is taken from Sādhanā (Deb, 1999b). Reproduced with permission from Indian Academy of Sciences, Bangalore.

Binary GAs work with a coding of decision variables, instead of the variables themselves. They 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 a discontinuous function may be handled by using GAs. This allows GAs to be applied to a wide variety of problem domains. Another 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 beforehand what coding is suitable for a problem, a number of experimental studies (Bäck, 1996; Radcliffe, 1991) suggest that a coding which *respects* the underlying *building block* processing must be used.

The more striking difference between GAs and most classical optimization methods is that GAs work with a population of solutions instead of a single solution. Because there is more than one string being processed simultaneously and used to update every string in the population, it is likely that the expected GA solution may be a global solution. Even though some classical algorithms are population-based, such as Box's algorithm (Box, 1965), they do not use the obtained information efficiently. Moreover, since a population is what is updated at every generation, a set of solutions (in the case of multi-modal optimization, multi-objective optimization and others) can be obtained simultaneously.

In the above discussion of GA operators and their working principles, 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, although in some of these methods search directions are found by using the objective function values, which 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 unnecessary, but careful thinking may provide some interesting properties for this type of search. The basic problem with most of the classical methods is that they use fixed transition rules to move from one solution to another solution. Since the destiny of these methods is pre-determined, they can only be applied to a special class of problems, in particular those for which the transition rules lead most search points towards the correct optimum. Thus, these methods are not robust and cannot be applied to a wide variety of problems. GAs, on the other hand, use probabilistic transition rules and an initial random population. These two features allow GAs to recover from early mistakes, if any, and enable them to handle a wide class of problems.

Another difference to most classical methods is that GAs can be easily and conveniently used in parallel systems. By using the tournament selection operator,

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 implementations. There is another advantage. Since in real-world design optimization problems, most computational time is spent in evaluating solutions, 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 information obtained up until the current generation through recombination and mutation operators with the extent of exploitation through the selection operator. If the solutions obtained are exploited too much, premature convergence is expected. On the other hand, if too much stress is given on a search, the information obtained thus far has not been used properly. Therefore, the solution time may be enormous and the search exhibits a similar behavior to that of a random search. Most classical methods have fixed transition rules and hence have fixed degrees of exploration and exploitation. Since these issues can be controlled in a GA by varying the parameters involved in the genetic operators, GAs provide an ideal platform for performing a flexible search.

Understanding How GAs Work

The working principle described above is simple, with GA operators involving string copying and substring exchange, plus the occasional alteration of bits. Indeed, it is surprising that with such simple operators and mechanisms, a potential search is possible. We will try to give an intuitive answer to such doubts and also remind the reader that a number of studies have attempted to find a rigorous mathematical convergence proof for GAs (Rudolph, 1994; Vose, 1999; Whitley, 1992). Even though the operations are simple, GAs are highly nonlinear, massively multi-faceted, stochastic and complex. There exist studies using Markov chain analysis which 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 the high-speed computers available today, other analytical techniques (statistical mechanics approaches and diffusion models) have also been used to analyze the convergence properties of GAs.

In order to investigate why GAs work, let us apply the GA for only one-cycle to a numerical maximization problem:

$$\left.\begin{array}{rl} \text{Maximize} & \sin(x), \\ \text{Variable bound} & 0 \leq x \leq \pi. \end{array}\right\} \tag{4.3}$$

We will use five-bit strings to represent the variable x in the range $[0, \pi]$, so that the string (00000) represents the $x = 0$ solution and the string (11111) represents the $x = \pi$ solution. The other 30 strings are mapped in the range $[0, \pi]$ uniformly. Let

us also assume that we use a population of size four, the proportionate selection, the single-point crossover operator with $p_c = 1$, and no mutation (or, $p_m = 0$). To start the GA simulation, we create a random initial population, evaluate each string, and then use three GA operators, as shown in Table 4. 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, the other three stings are also evaluated. Since the proportionate reproduction scheme assigns a number of copies according to a string's fitness, the expected number of copies for each string is calculated in column 5. When the proportionate selection operator is actually implemented, the number of copies allocated to the strings is shown in column 6. Column 7 shows the mating pool. It is noteworthy that the third string in the initial population has a fitness which is very small compared to the average fitness of the population and is eliminated by the selection operator. On the other hand, the second string, being a good string, made two copies in the mating pool. The crossover sites are chosen at random and the four new strings created after crossover is shown in column 3 of the bottom table. Since no mutation is used, none of the bits are altered. Thus, column 3 of the bottom table represents the population at the end of one cycle of a GA. Thereafter, each of these stings is then decoded, mapped and evaluated. This completes one generation of a GA simulation. The average fitness of the new population is found to be 0.710, i.e. an improvement from that in the initial population. It is interesting to note that even though all operators used random numbers, a GA with all three operators produces a directed search, which usually results in an increase in the average quality of solutions from one generation to the next.

The string copying and substring exchange are all interesting and seem to improve the average performance of a population, but let us now ask the question: 'What has been processed in one cycle of a GA?' If we investigate carefully, we observe that among the strings of the two populations there are some similarities in the string positions among the strings. By the application of three GA operators, the number of strings with similarities at certain string positions has been increased from the initial population to the new population. These similarities are called *schema* in the GA literature. More specifically, a schema represents a set of strings with certain similarities at certain string positions. To represent a schema for binary codings, a triplet (1, 0 and $*$) is used; a $*$ represents both 1 or 0. It is interesting to note that a string is also a schema representing only one string – the string itself.

Two definitions are associated with a schema. The *order* of a schema H is defined as the number of defined positions in the schema and is represented as $o(H)$. A schema with full order $o(H) = \ell$ represents a string. The *defining length* of a schema H is defined as the distance between the outermost defined positions. For example, the schema $H = (* \ 1 \ 0 \ * \ * \ 0 \ * \ * \ *)$ has an order $o(H) = 3$ (there are three defined positions: 1 at the second gene, 0 at the third gene, and 0 at the sixth gene) and a defining length $\delta(H) = 6 - 2 = 4$.

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 4, we observe that there is only one string

Table 4 One generation of a GA hand-simulation on the function $\sin(x)$.

Initial population						
String	DV[a]	x	f(x)	f_i/f_{avg}	AC[b]	Mating pool
01001	9	0.912	0.791	1.39	1	01001
10100	20	2.027	0.898	1.58	2	10100
00001	1	0.101	0.101	0.18	0	10100
11010	26	2.635	0.485	0.85	1	11010
	Average, f_{avg}		0.569			

Mating Pool	CS[c]	New population			
		String	DV[a]	x	f(x)
01001	3	01000	8	0.811	0.725
10100	3	10101	21	2.128	0.849
10100	2	10010	18	1.824	0.968
11010	2	11100	28	2.838	0.299
			Average, f_{avg}		0.710

[a] DV, decoded value of the string.

[b] AC, actual count of strings in the population.

[c] CS, cross site.

representing this schema H_1 in the initial population and that there are two strings representing 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 not one 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.

The so-called schema theorem provides an estimate of the growth of a schema H under the action of one cycle of the above tripartite GA. Holland (1975) and later Goldberg (1989) calculated the growth of the schema under a selection operator and then calculated the survival probability of the schema under crossover and mutation operators, but did not calculate the probability of constructing a schema from recombination and mutation operations in a generic sense. For a single-point crossover operator with a probability p_c, a mutation operator with a probability p_m, and the proportionate selection operator, Goldberg (1989) calculated the following lower bound on the schema growth in one iteration of a GA:

$$m(H, t+1) \geq m(H, t)\frac{f(H)}{f_{avg}}\left[1 - p_c\frac{\delta(H)}{\ell - 1} - p_m o(H)\right], \tag{4.4}$$

where $m(H, t)$ is the number of copies of the schema H in the population at generation

t, $f(H)$ is the fitness of the schema (defined as the average fitness of all strings representing the schema in the population), and f_{avg} is the average fitness of the population. The above inequality leads to the schema theorem (Holland, 1975), as follows.

Theorem 4.2.1. *Short, low-order, and above-average schemata receive exponentially increasing number of trials in subsequent generations.*

A schema represents a number of similar strings. Thus, a schema can be thought of as representing a certain region in the search space. For the above function, the schema $H_1 = (1\ 0\ *\ *\ *)$ represents strings with x values varying from 1.621 to 2.330 with function values varying from 0.999 to 0.725. On the other hand, the schema $H_2 = (0\ 0\ *\ *\ *)$ 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 4 without having to count all of these competing schema and without the knowledge of the complete search space, but by manipulating only a few instances of the search space. Let us use the inequality shown in equation (4.4) to estimate the growth of H_1 and H_2. We observe that there is only one string (the second string) representing this schema, or $m(H_1,0) = 1$. Since all strings are used in the crossover operation and no mutation is used, $p_c = 1.0$ and $p_m = 0$. For the schema H_1, the fitness $f(H_1) = 0.898$, the order $o(H_1) = 2$, and the defining length $\delta(H_1) = 1$. In addition, the average fitness of the population is $f_{avg} = 0.569$. Thus, we obtain from equation (4.4):

$$\begin{aligned} m(H_1,1) &\geq (1)\cdot\frac{0.898}{0.569}\left[1-(1.0)\frac{1}{5-1}-(0.0)(2)\right], \\ &= 1.184. \end{aligned}$$

The above calculation suggests that the number of strings representing the schema H_1 must increase. We have two representations (the second and third strings) of this schema in the next generation. For the schema H_2, the estimated number of copies using equation (4.4) is $m(H_2,1) \geq 0.133$. Table 4 shows that no representative string of this schema exists in the new population.

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 contain the possible near-optimal solution (the string (1 0 0 0 0) is the optimal string in this problem). The schemata that are short, low-order, 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, such as H_1 (which has the first two bits in common with the true optimal string) in the above example. Note that although H_2 is short and low-order, it is not an above-average schema. Thus, H_2 is not a building block. This is how GAs can emphasize different short, low-order and above-average schemata in the population. Once adequate number of such building blocks are present in a GA population, they

get combined together due to the action of the GA operators to form bigger and better building blocks. This process finally leads a GA to find the optimal solution. This hypothesis is known as the *Building Block Hypothesis* (Goldberg, 1989).

Significance of Population Size

In a genetic algorithm, a decision about whether to have a '1' or a '0' has to be made at every bit position. The mechanism of using selection, crossover and mutation is one of the many ways of arriving at a decision. Holland viewed this decision-making in each bit independently and compared the process to a two-armed bandit game-playing problem. With one arm being fixed to either a 1 or a 0, the task is to find the arm which makes the maximum payoff (or fitness). With respect to the above definition of schema, every one-bit information is an order-one schema. Since the rest $(\ell - 1)$ bits are not considered, the fitness of one instantiation of any order-one schema is dependent on the exact value of bits at other locations.

In order to demonstrate the significance of an adequate population size in a GA, let us make use of a simple bimodal single-variable objective function for maximization:

$$f(x) = c_1 N(x, a_1, b_1) + c_2 N(x, a_2, b_2), \tag{4.5}$$

where the function $N(x, a_i, b_i)$ is a Gaussian function with a mean at a_i and a standard deviation of b_i. The search space is spanned over $[0, 1]$. By varying these six parameters, one can obtain functions of differing complexity. Let us choose the location of two maxima: $a_1 = 0.25$, and $a_2 = 0.80$. First, we use the following setting:

$$b_1 = 0.05, \quad b_2 = 0.05, \quad c_1 = 0.5, \quad c_2 = 1.0.$$

This makes the second maximum (at $x = 0.80$) the global maximum solution. Figure 50 shows the function. Let us consider two competing schemata ($H_0 \equiv 0 * \ldots *$) and ($H_1 \equiv 1 * \ldots *$) and analyze their growth under genetic operators. Figure 50 also shows that the left region ($x \in [0, 0.5]$) is represented by H_0 and the right region ($x \in [0.5, 1.0]$) is represented by the schema H_1. The average fitness of these two schemata are also shown with dashed lines. It is clear that the schema H_1 has a better fitness than H_0. Since the schema order and defining length are identical for both these schemata and H_1 is above-average, it becomes a building block. Thus, it is expected that a GA will make the right decision of choosing more solutions of H_1 than H_0. Since a GA can decide the correct direction of search by processing order-one schemata alone, a population of size two is essentially enough. However, note that the average fitness values shown by dashed lines in Figure 50 for any of the schemata do not adequately represent the fitness of all the strings which are represented by the schema. There is a large variation in the fitness values among the strings representing a schema. Because of this variability in fitness values, an appropriate number of copies in each region must be present to adequately represent the fitness variations. If we assume that λ (> 1) number of solutions are needed to statistically represent a region, an overall population of size 2λ is enough to solve the above problem by using a binary-coded GA.

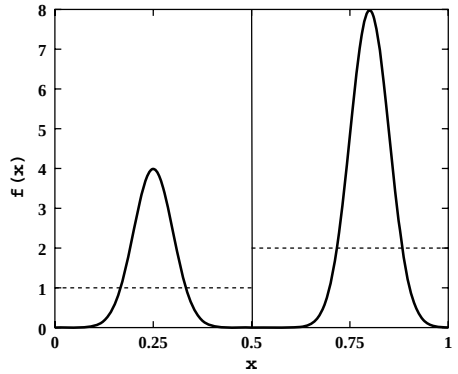


Figure 50 The simple bimodal function and average fitness values of two order-one schemata.

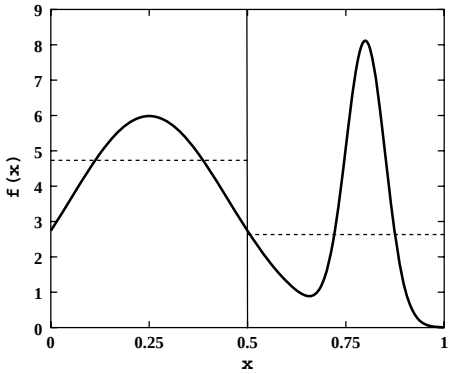


Figure 51 The modified bimodal function and average fitness values of two order-one schemata.

Now let us change the problem by choosing another parameter setting:

$$b_1 = 0.20, \quad b_2 = 0.05, \quad c_1 = 3, \quad c_2 = 1.$$

Figure 51 shows the function having two maxima. Now, the local maximum solution occupies a larger basin of attraction than the global maximum solution. When the fitnesses of H_0 and H_1 are computed (Table 5), it is observed that H_0 has a better fitness than H_1. Thus, if a population size of 2λ is used to solve the modified problem, a GA will be misled and emphasize more solutions of H_0 in a random initial population. Although this does not mean that a GA will not be able to recover from such early mistakes, a reliable application of the GA would be to make sure that it is started in the correct direction from the very first generation. In order to investigate the higher-order schema competitions, we have computed the fitness of four order-two competing schemata in Table 5, where the order of H_{00}, H_{01}, H_{10} and H_{11} is vertically downwards. These values are also shown pictorially in Figure 52. Here also,

Table 5 Schema fitness values for the bimodal problem.

Schema fitness					
Order-one		Order-two		Order-three	
(0.000–0.500)	4.732	(0.000–0.250)	4.732	(0.000–0.125)	3.848
(0.500–1.000)	2.633	(0.250–0.500)	4.732	(0.125–0.250)	5.616
		(0.500–0.750)	1.828	(0.250–0.375)	5.616
		(0.750–1.000)	3.439	(0.375–0.500)	3.848
				(0.500–0.625)	1.808
				(0.625–0.750)	1.848
				(0.750–0.875)	6.324
				(0.875–1.000)	0.553

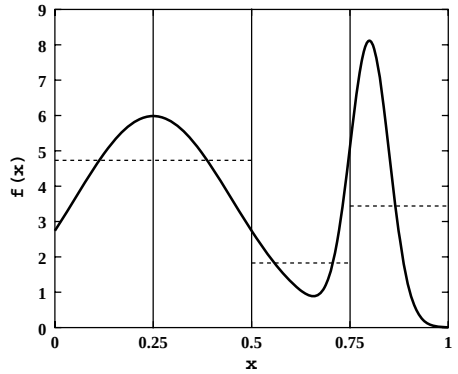


Figure 52 Average fitness values of different order-two schemata shown for the modified bimodal function.

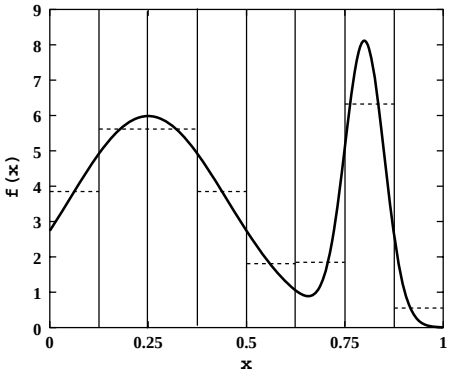


Figure 53 Average fitness values of different order-three schemata shown for the modified bimodal function.

we observe that the schema (representing the region $0.750 \leq x \leq 0.875$) containing the global maximum solution does not win the competition. When we compute the order-three schema competitions (see Table 5), we observe that the schema containing the global maximum wins the competition. Figure 53 clearly shows that the schema containing the global maximum has the maximum fitness value. However, the overall function is favored for the local maximum. In order to recognize the importance of the global maximum solution, a schema with a narrower region must be emphasized. In fact, any higher-order (> 3) schema competition will favor the global maximum solution. Thus, if a GA has to find the global maximum in a reliable way, the GA has to start comparing order-three schemata with the variability in string fitness in mind. Any competition lower than order three will favor the local maximum solution. One of the ways to ensure that a GA will start processing from order-three schemata is to have a population large enough to house strings representing all 2^3 competing schemata. Let us recall that the former bimodal problem (see Figure 50) required us to start processing from order-one schemata (having 2^1 or two competing schemata) in order to proceed in the correct direction. Since here, eight schemata are competing, a population of size 8λ would be adequate to represent a sufficient number of strings from each of the eight competing schemata. Thus, the modified bimodal problem needs four times more population size in order to reliably find the global maximum solution compared to the original bimodal problem.

Thus, we observe that the population size is related to the complexity of the problem. The parameter λ depends on the variability in fitness values of strings representing a schema. For example, when keeping the location of two maxima, if the function is not as smooth (Figure 54), the required sample size λ needed to detect a *signal* from the *noise* would be higher. Based on these considerations, Goldberg et al. (1992) calculated a population sizing estimate for a binary-coded GA. Following this study, Harik et al. (1999) tightened the sizing expression by considering the schema competition from a gambler's ruin model.

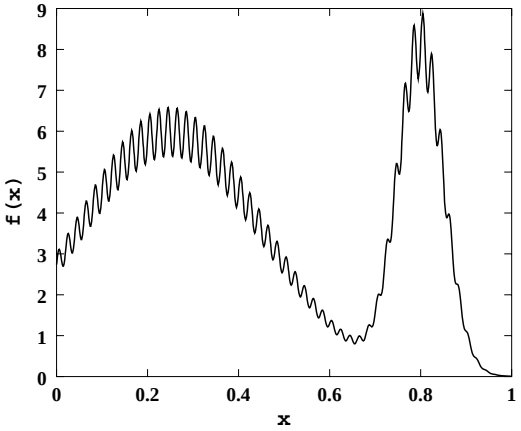


Figure 54 A function with a large variability in function values demands a large population size to find the global optimum.

GA Operator Interactions

Besides choosing an appropriate population size, another important matter is the balance between exploitation caused by the selection operator and the exploration introduced by the chosen recombination and mutation operators. If the selection operator uses too much selection pressure, meaning that it emphasizes the population-best solution too much by assigning many copies of it, the population loses its diversity very quickly. In order to bring back the diversity, the exploration power of the recombination and mutation operators must be large, meaning that these operators must be able to create solutions which are fairly different from the parent solutions. Otherwise, the population can become the victim of excessive selection pressure and eventually converge to a sub-optimal solution. On the other hand, if the selection pressure is very low, meaning that not much emphasis is given to the population-best solutions, the GA's search procedure behaves like a random search process. Although a qualitative argument of the balance between these two issues can be made and understood, a quantitative relationship between them is difficult to achieve.

Goldberg et al. (1993b) balanced the extent of exploitation and exploration issues by calculating the characteristic times of a selection and a crossover operator. In their earlier work (Goldberg and Deb, 1991), the *take-over times* t_s of a number of selection operators were calculated. The take-over time was defined as the number of generations required for the population-best solution to occupy all but one of the population slots by repetitive application of the selection operator alone. This characteristic time of a selection operator provides information about the speed with which the best solution in a population is emphasized. It was observed that binary tournament selection and linear ranking selection (with an assignment of two copies to the best solution) have the same take-over times. Proportionate selection is much slower than tournament selection.

For the uniform crossover operator, investigators have calculated the *mixing time* t_c, which refers to the number of generations required before repetitive application of the crossover operator alone can find a desired solution. This time gives information about how long a GA would have to wait before an adequate mixing of population members can produce the desired solutions.

By comparing the order of magnitudes of these two characteristic times, investigators argued that a GA will work successfully if the following relationship holds:

$$p_c \geq A \ln s, \tag{4.6}$$

where A is a constant which relates to the string length and population size and s is the selection pressure. On the one-max test problem, where the objective is to maximize the number of 1s in a string, Figure 55 shows the theoretical *control map* for successful GAs. A similar control map is obtained using GA simulation results (Figure 56). What is interesting to note from these figures is that a GA with any arbitrary parameter setting is not expected to work well even on a simple problem. A GA with a selection pressure s and a crossover probability p_c falling inside the control map finds the desired optimum. The above problem is a bit-wise linear problem, where a decision can be made in each bit, independent of the decisions taken at other bits. Even for this problem, a tournament selection with a large tournament size (such as $s = 20$ or so) and the uniform crossover operator with a small crossover probability (such as $p_c = 0.2$ or so) and no mutation, is not expected to find the true optimum solution. Although these parameter settings are probably extreme ones to choose in any application, they provide an understanding of the importance of interactions among GA operators and their parameter settings in performing a successful GA run.

For a selection pressure lower than a critical value, GAs can drift to any arbitrary

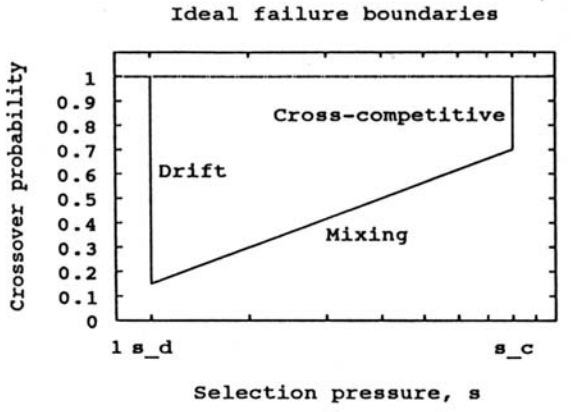


Figure 55 A theoretical control map for the working of a GA on the one-max problem (Goldberg et al., 1993b). Reprinted with permission from the Journal of the Society of Instruments and Control Engineers (SICE).

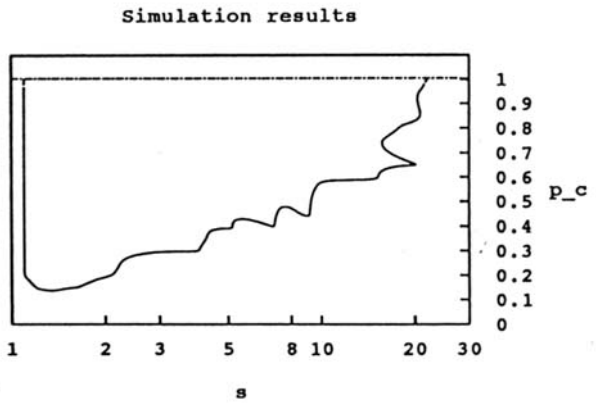


Figure 56 Simulation results show the working region for a GA on the one-max problem (Goldberg et al., 1993b). Reprinted with permission from the Journal of the Society of Instruments and Control Engineers (SICE).

solution and for a very large selection pressure, GAs cause important building blocks to compete among themselves, thereby not finding the desired optimum. These two cases become the two extreme bounds on the selection pressure in the above control map.

GA Operator Interactions under a Fixed Number of Trials

It is clear from the description of the working principles of a GA that such operators are tunable with the parameters associated with each of them. Since there exist flexibilities for changing the importance of one operator over another or flexibilities in using a different representation scheme to suit a problem, GAs are widely applicable to various types of problems. As we have seen in the previous section, along with this flexibility a burden is placed on the part of the user to choose an appropriate combination of GA operators and their parameters. While facing a real-world problem, the overall time to solve the problem is often fixed. This means that whatever algorithm is used to solve the problem at hand, we are only allowed to use a fixed number of trials (or function evaluations). For the sake of our discussion here, let us say that we are allowed to use a total of S trials to solve a problem. Since in a generational GA we create N new offspring at every generation, this means that we are allowed to run a GA having a population size N for a maximum of $t_{max} = S/N$ generations. If N is small, a large number of generations are allowed. On the other hand, if N is large, a small number of generations will be allowed. Then, an important point to ponder is what population size is appropriate to choose when the overall number of trials is fixed. A related question to ask is: 'Given the adequate population size, what combination of GA operators is appropriate?' It is important to realize that the answer to these

questions will be different if the maximum number of allowed trials does not have a bound. Several researchers have attempted to find such GA operators and their parameter interactions for fixed number of trials, simply because such considerations are pragmatic in solving real-world problems.

Deb and Agrawal (1999b) conducted a series of experiments with different GA operators and parameter settings and applied these to different problems of varying difficulties. The outcome of this study is important and is outlined in the following:

1. For simpler problems (such as well-behaved unimodal or linear problems), a GA with a selection operator and a crossover or a mutation operator, or a combination of crossover and mutation operators, can all work satisfactorily. However, there is a distinct difference in the required population size in each case. For a selecto-mutation GA, a small population size (such as 3–6) provides the optimum performance. Since a selecto-mutation GA is similar to a local search approach, instead of more population members it requires a larger number of iterations to navigate its search towards the optimum. On the other hand, for a selecto-recombinative GA (with no mutation operator), the population size requirement is rather high. A GA with a crossover requires an adequate population size (described in Section 4.2.1) to steer the search in the right direction. Since the search relies on combining salient building blocks, the building block discovery and their emphasis requires an adequate number of population members. However, once the salient building blocks are found, not many generations are needed to combine them together. Thus, although a larger population size is in order, the number of generations required may be comparatively smaller. This is in contradiction to the common belief that since GAs use a population of solutions in each iteration, they are computationally more expensive than a local search algorithm.
2. For difficult problems (where difficulty can come from multi-modality, dimensionality of the search space, ruggedness of the fitness function, etc.), selecto-mutation GAs do not work successfully in finding the correct optimum solution. However, selecto-recombinative GAs can find the correct optimum with an adequate population size. Oates et al. (1999a, 1999b) observed similar results on a number of other problems.

Elite-Preserving Operator

In order to preserve and use previously found best solutions in subsequent generations, an elite-preserving operator is often recommended. In addition to an overall increase in performance, there is another advantage of using such elitism. In an elitist GA, the statistics of the population-best solutions cannot degrade with generations. There exists a number of ways to introduce elitism. We will discuss here a couple of popular approaches.

In a simple implementation, the best $\epsilon\%$ of the population from the current population is directly copied to the next generation. The rest $((100-\epsilon)\%)$ of the new population is created by the usual genetic operations applied on the entire current

population (including the chosen ϵ% elite members). In this way, the best solutions of the current population not only get passed from one generation to another, but they also participate with other members of the population in creating other population members.

The above implementation is a generational one. However, elitism is more commonly introduced in the context of steady-state GAs, where after every offspring is created, it is used to modify the entire population. The elitism is introduced by comparing two offspring with two parent solutions and then keeping the two better solutions. A somewhat generational version of this concept is implemented by first creating an offspring population by using usual genetic operations and then choosing the best N solutions from a combined population (of size 2N) of parent and offspring populations (Eshelman, 1991).

Theoretical Studies

Besides 'back-of-the-envelope' calculations and empirical studies, there exists a number of rigorous mathematical studies where a complete processing of a finite population under GA operators is modeled by using Markov chains (Vose, 1999; Vose and Wright, 1996) and statistical mechanics (Prügel-Bennett and Shapiro, 1994; Rogers and Prügel-Bennett, 1998). On different classes of functions, the recent dynamical systems analysis of GAs by Vose and Rowe (2000) as a random heuristic search reveals the complex dynamical behavior of a binary-coded GA with many metastable attractors. The interesting outcome is the effect of population size on the degree of metastability. For larger population sizes, the duration near a metastable state is more. In the statistical mechanics approach, instead of considering microscopic details of the evolving system, several macroscopic variables describing the system are modeled. Prügel-Bennett et al. (1994) studied the evolution of the first four cumulants of the fitness distribution as macroscopic variables. These are the mean, standard deviation, skewness and kurtosis of the fitness distribution. Analyzing different GA implementations with the help of these cumulants provides interesting insights into the complex interactions among GA operators (Rogers and Prügel-Bennett, 1998).

Rudolph (1994) has shown that a simple GA with an elite-preserving operator and non-zero mutation probability converges to the global optimum solution. However, the absence of elitism does not always allow a simple GA to have this convergence property.

Why Use Binary Alphabets?

Binary alphabets must naturally be used in problems having Boolean decision variables. In practice, many problems involve Boolean decision variables, particularly where the presence or absence of an entity is a decision variable. However, in handling a generic discrete search space problem, one can think of using higher-cardinality alphabets as well. Although it is not entirely intuitive why binary alphabets are better,

there exist a couple of reasons for choosing binary alphabets. There also exists at least one counter-argument to such a proposal.

Holland (1975) and Goldberg (1989) argued that although a GA with a population size N processes N strings in every generation, in the schema space as many as $O(N^3)$ schemata get processed. This leverage provides a GA with its *implicit parallelism* property. Based on a count of the total number of schema being processed, Goldberg (1991) showed that a binary coding offers the maximum number of schemata per bit of information. Since the schemata count is the maximum for binary alphabets, the leverage achievable from it is also a maximum.

Reeves (1993b) estimated the minimum initial population size needed so that every possible solution in the search space can be reached from this population by crossover alone. The initial population must contain a string such that at every gene position every possible alphabet is present. Since mutation operation is ignored and crossover can only exchange information between strings, this condition will ensure that repetitive operation of crossover to this initial population will eventually create any possible solution in the search space. The conclusion of this study is that the higher the cardinality of the alphabet, then the larger is the minimum population size requirement to achieve the above condition. This means that with larger cardinality alphabets, more storage and computational effort will be necessary.

Antonisse (1989) criticized the schema counting procedure of Holland and Goldberg. He argued that for more than two alphabets, the use of one 'don't care' symbol to represent all combinations of alphabets is flawed. He suggested that all $(2^\nu - \nu - 1)$ different 'don't care' symbols must be used, where ν is the cardinality of the chosen alphabet. This makes $(2^\nu - 1)$ different possibilities in each gene position, thereby making the overall number of schemata as $(2^\nu - 1)^\ell$, and not $(\nu + 1)^\ell$. This modified number of total schemata per alphabet turns out to be monotonically increasing with the cardinality ν; in addition, the greater the cardinality of the alphabet, then the larger is the effect of Antonisse's schema processing. Since a GA does not work on the schema space, it is difficult to evaluate the merits of both of these arguments in an absolute sense. However, it is clear that whether it is Holland's schemata or Antonisse's augmented schemata that get processed in a GA with a higher cardinality alphabet, the implicit parallelism argument holds in both cases.

Other GAs

In addition to the simple binary GA described above, there exist a number of advanced GAs, specifically designed to have advanced features. In order to ensure the convergence properties and to accelerate the rate of convergence, elitist GAs are commonly used. Of these, Eshelman's (Eshelman, 1991) CHC[1] is an elitist GA which selects the best N solutions from a combination of 2N population members formed by

[1] The interpretation of C, H and C is derived from three features of the CHC algorithm: cross generation selection, heterogeneous recombination and cataclysmic mutation (Eshelman, 2001).

using the parent and the offspring populations. Moreover, the CHC algorithm uses a restricted mating (with a uniform like crossover operator) so that dissimilar solutions (in terms of bit differences between them) will mate with each other. Because of a high selection pressure introduced by the elite-preserving operation, this GA needs to use a highly disruptive recombination operator.

Genetic Implementor (GENITOR) (Whitley, 1989) is a steady-state elitist GA, where each offspring is introduced into the parent population one at a time. Parents are chosen according to a ranking selection operator and the created offspring replaces the worst individual in the parent population. On a generational framework, Goldberg and Deb (1991) have shown that the GENITOR algorithm has an inherently large selection pressure. Once again, this GA requires a highly disruptive recombination operator to maintain a balance between exploration and exploitation issues.

Messy GAs (Goldberg et al., 1989, 1990) are variable-length GAs where the identification of all salient building blocks and the subsequent mixing of them is temporally separated into two phases. In the primordial phase, important building blocks are identified by comparing different plausible building blocks with one another. In the juxtapositional phase, the necessary building blocks are combined together by the use of a cut-and-splice operator to form optimal solutions in difficult problems. We will describe messy GAs in more detail in Section 6.7. The quest for finding the necessary linkage needed in a problem on the fly, a matter which took shape with the messy GA study, has now been paid adequate attention. The continuing research on gene expression GAs (Kargupta, 1997), linkage learning GAs (Harik and Goldberg, 1996), the Bayesian optimization algorithm (Pelican et al., 1999), the factorial distribution algorithm (Mühlenbein and Mahnig, 1999) and various others represent worthy efforts applied in this direction.

Genetic algorithms have also been extensively used in scheduling and other combinatorial optimization problems. The representation used in such problems is different from the usual binary string representation. This also requires different recombination operators to be devised for creating valid offspring from valid parent solutions. A number of such GA implementations and applications can be found in various books (Goldberg, 1989; Gen and Cheng, 1997; Michalewicz, 1992; Mitchell, 1996), as well as certain journals and conference proceedings.

4.2.2 Real-Parameter Genetic Algorithms

When binary-coded GAs need to be used to handle problems having a continuous search space, a number of difficulties arise. One difficulty is the Hamming cliffs associated with certain strings (such as strings 01111 and 10000) from which a transition to a neighboring solution (in real space) requires the alteration of many bits. Hamming cliffs present in a binary coding cause artificial hindrance to a gradual search in the continuous search space. The other difficulty is the inability to achieve any arbitrary precision in the optimal solution. In binary-coded GAs, the string length must be chosen a priori to enable GAs to achieve a certain precision in the solution.

The more the required precision, then the larger is the string length. For large strings, the population size requirement is also large (Goldberg et al., 1992), thereby increasing the computational complexity of the algorithm. Since a fixed coding scheme is used to code the decision variables, variable bounds must be such that they bracket the optimum variable values. Since in many problems this information is not usually known a priori, this may cause some difficulty in using binary-coded GAs in such problems. Furthermore, a careful thinking of the schema processing in binary strings reveals that not all Holland's schemata are equally important in most problems having a continuous search space. To a continuous search space, the meaningful schemata are those that represent the contiguous regions of the search space. The schema (* * * * 1), for example, represents every other point in the discretized search space. Although this schema may be useful in certain periodic or oscillatory functions, the schema (1 * * * *) signifies a more meaningful schema representing the right-half of the search space in most problems. Thus, the crossover operator used in the binary coding needs to be redesigned in order to increase the propagation of more meaningful schemata pertaining to a continuous search space.

There exists a number of real-parameter GA implementations, where crossover and mutation operators are applied directly to real parameter values. Since real parameters are used directly (without any string coding), solving real-parameter optimization problems is a step easier when compared to the binary-coded GAs. Unlike in the binary-coded GAs, decision variables can be directly used to compute the fitness values. Since the selection operator works with the fitness value, any selection operator used with binary-coded GAs can also be used in real-parameter GAs.

However, the difficulty arises with the search operators. In the binary-coded GAs, decision variables are coded in finite-length strings and exchanging portions of two parent strings is easier to implement and visualize. Simply flipping a bit to perform mutation is also convenient and resembles a natural mutation event. In real-parameter GAs, the main challenge is how to use a pair of real-parameter decision variable vectors to create a new pair of offspring vectors or how to perturb a decision variable vector to a mutated vector in a meaningful manner. As in such cases the term 'crossover' is not that meaningful, they can be best described as *blending* operators (DeJong, 1999). However, most blending operators in real-parameter GAs are known as crossover operators, and we will continue to use the same term here. First, we describe a number of real-parameter crossover operators and then present a few real-parameter mutation operators. Herrara et al. (1998) have provided a good overview of many real-parameter crossover and mutation operators.

Linear Crossover

One of the earliest implementations was reported by Wright (1991), where a linear crossover operator created the three solutions, $0.5(x_i^{(1,t)}+x_i^{(2,t)})$, $(1.5x_i^{(1,t)}-0.5x_i^{(2,t)})$ and $(-0.5x_i^{(1,t)}+1.5x_i^{(2,t)})$ from two parent solutions $x_i^{(1,t)}$ and $x_i^{(2,t)}$ at generation t, with the best two solutions being chosen as offspring. Figure 57 shows the three

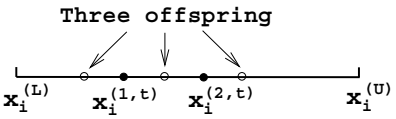


Figure 57 Wright's linear crossover operator. Parents are marked by filled circles.

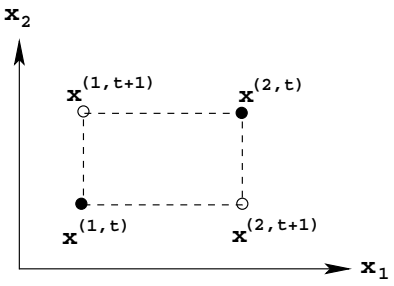


Figure 58 Naive crossover on two decision variables. Parents are marked by filled circles.

offspring created from the two parent solutions.

A Naive Crossover

This crossover operator is similar to the crossover operators used in binary-coded GAs. Cross sites are only allowed to be chosen at the variable boundaries. For example, a single-point crossover at the third site will produce the following offspring:

$$\begin{array}{rl} \text{Parent 1:} & (x_1^{(1,t)} \quad x_2^{(1,t)} \quad x_3^{(1,t)} \quad x_4^{(1,t)} \quad \ldots \quad x_n^{(1,t)}) \\ \text{Parent 2:} & (x_1^{(2,t)} \quad x_2^{(2,t)} \quad x_3^{(2,t)} \quad x_4^{(2,t)} \quad \ldots \quad x_n^{(2,t)}) \\ \text{Offspring 1:} & (x_1^{(1,t)} \quad x_2^{(1,t)} \quad x_3^{(1,t)} \quad x_4^{(2,t)} \quad \ldots \quad x_n^{(2,t)}) \\ \text{Offspring 2:} & (x_1^{(2,t)} \quad x_2^{(2,t)} \quad x_3^{(2,t)} \quad x_4^{(1,t)} \quad \ldots \quad x_n^{(1,t)}) \end{array}$$

For two decision variables and for the case when the cross site falls in the variable boundary, the offspring can be either parents themselves or the other two diagonal solutions, as shown in Figure 58. Like in the single-point crossover operator, two-point, n-point, or uniform crossover operators can also be used in a similar manner. This crossover operator does not have an adequate search power and thus the search within a decision variable has to mainly rely on the mutation operator.

Blend Crossover and Its Variants

Goldberg introduced the concept of virtual alphabets in the context of real-parameter GAs (Goldberg, 1991), a matter which is discussed on page 124. Eshelman and Schaffer (1993) have introduced the notion of *interval* schemata which is similar in principle to the virtual alphabets. They also suggested a blend crossover (BLX-α) operator for real-parameter GAs. For two parent solutions $x_i^{(1,t)}$ and $x_i^{(2,t)}$ (assuming $x_i^{(1,t)} < x_i^{(2,t)}$), the BLX-$\alpha$ randomly picks a solution in the range $[x_i^{(1,t)} - \alpha(x_i^{(2,t)} - x_i^{(1,t)}), x_i^{(2,t)} + \alpha(x_i^{(2,t)} - x_i^{(1,t)})]$. This crossover operator is illustrated in Figure 59.

Thus, if u_i is a random number between 0 and 1, the following is an offspring:

$$x_i^{(1,t+1)} = (1 - \gamma_i)x_i^{(1,t)} + \gamma_i x_i^{(2,t)}, \tag{4.7}$$

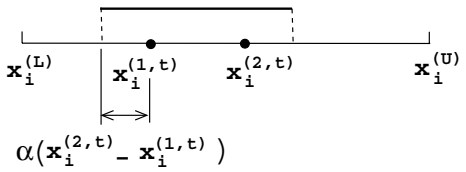


Figure 59 The BLX-α operator. Parents are marked by filled circles.

where $\gamma_i = (1+2\alpha)u_i - \alpha$. If α is zero, this crossover creates a random solution in the range $(x_i^{(1,t)}, x_i^{(2,t)})$. In a number of test problems, the investigators have reported that BLX-0.5 (with $\alpha = 0.5$) performs better than BLX operators with any other α value. However, it is important to note that the factor γ_i is uniformly distributed for a fixed value of α. However, BLX-α has an interesting property: the location of the offspring depends on the difference in parent solutions. This will be clear if we rewrite equation (4.7) as follows:

$$\left(x_i^{(1,t+1)} - x_i^{(1,t)}\right) = \gamma_i \left(x_i^{(2,t)} - x_i^{(1,t)}\right). \tag{4.8}$$

If the difference between the parent solutions is small, the difference between the offspring and parent solutions is also small. This property of a search operator allows us to constitute an adaptive search. If the diversity in the parent population is large, an offspring population with a large diversity is expected, and vice versa. Thus, such an operator will allow the searching of the entire space early on (when a random population over the entire search space is initialized) and also allow us to maintain a focussed search when the population tends to converge in some region in the search space.

It is important to note that there exists a number of other crossover operators which work by using the same principle. The arithmetic crossover (Michalewicz and Janikow, 1991) uses equation (4.7) with a fixed value of γ for all decision variables. However, γ is chosen by carefully calculating its maximum allowed value in all decision variables so that the resulting values do not exceed the lower or upper limits. The extended crossover operator (Voigt et al., 1995) is also similar to BLX-α.

Simulated Binary Crossover

In 1995, this author and his students developed the simulated binary crossover (SBX) operator, which works with two parent solutions and creates two offspring (Deb and Agrawal, 1995; Deb and Kumar, 1995). As the name suggests, the SBX operator simulates the working principle of the single-point crossover operator on binary strings. In these studies, we showed that this crossover operator *respects* the *interval* schemata processing, in the sense that common interval schemata between the parents are preserved in the offspring. The procedure of computing the offspring $x_i^{(1,t+1)}$ and $x_i^{(2,t+1)}$ from the parent solutions $x_i^{(1,t)}$ and $x_i^{(2,t)}$ is described as follows. A spread

factor β_i is defined as the ratio of the absolute difference in offspring values to that of the parents:

$$\beta_i = \left| \frac{x_i^{(2,t+1)} - x_i^{(1,t+1)}}{x_i^{(2,t)} - x_i^{(1,t)}} \right|. \tag{4.9}$$

First, a random number u_i between 0 and 1 is created. Thereafter, from a specified probability distribution function, the ordinate β_{q_i} is found so that the area under the probability curve from 0 to β_{q_i} is equal to the chosen random number u_i. The probability distribution used to create an offspring is derived to have a similar *search power* to that in a single-point crossover in binary-coded GAs and is given as follows (Deb and Agrawal, 1995):

$$\mathcal{P}(\beta_i) = \begin{cases} 0.5(\eta_c + 1)\beta_i^{\eta_c}, & \text{if } \beta_i \leq 1; \\ 0.5(\eta_c + 1)\dfrac{1}{\beta_i^{\eta_c+2}}, & \text{otherwise.} \end{cases} \tag{4.10}$$

Figure 60 shows the above probability distribution with $\eta_c = 2$ and 5 for creating offspring from two parent solutions ($x_i^{(1,t)} = 2.0$ and $x_i^{(2,t)} = 5.0$) in the real space. In the above expressions, the distribution index η_c is any non-negative real number. A large value of η_c gives a higher probability for creating 'near-parent' solutions and a small value of η_c allows distant solutions to be selected as offspring. Using equation (4.10), we calculate β_{q_i} by equating the area under the probability curve equal to u_i, as follows:

$$\beta_{q_i} = \begin{cases} (2u_i)^{\frac{1}{\eta_c+1}}, & \text{if } u_i \leq 0.5; \\ \left(\dfrac{1}{2(1-u_i)}\right)^{\frac{1}{\eta_c+1}}, & \text{otherwise.} \end{cases} \tag{4.11}$$

After obtaining β_{q_i} from the above probability distribution, the offspring are calculated as follows:

$$x_i^{(1,t+1)} = 0.5\left[(1+\beta_{q_i})x_i^{(1,t)} + (1-\beta_{q_i})x_i^{(2,t)}\right], \tag{4.12}$$

$$x_i^{(2,t+1)} = 0.5\left[(1-\beta_{q_i})x_i^{(1,t)} + (1+\beta_{q_i})x_i^{(2,t)}\right]. \tag{4.13}$$

Thus, the following step-by-step procedure is followed to create two offspring ($x_i^{(1,t+1)}$ and $x_i^{(2,t+1)}$) from two parent solutions ($x_i^{(1,t)}$ and $x_i^{(2,t)}$):

Step 1: Choose a random number $u_i \in [0, 1)$.

Step 2: Calculate β_{q_i} using equation (4.11).

Step 3: Compute the offspring by using equations (4.12) and (4.13).

Note that two offspring are symmetric about the parent solutions. This is deliberately enforced to avoid a bias towards any particular parent solution in a single crossover operation. Another interesting aspect of this crossover operator is that for

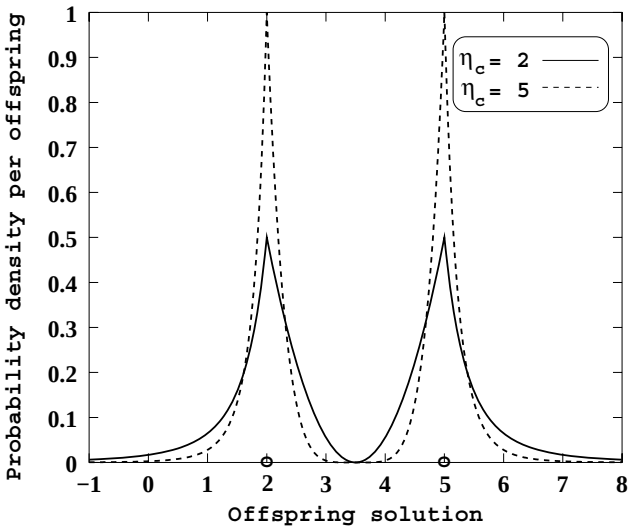


Figure 60 The probability density function for creating offspring under an SBX-η_c operator. Parents are marked with an 'o'. Reproduced from Deb and Beyer (2001) (© 2001 by the Massachusetts Institute of Technology).

a fixed η_c the offspring have a spread which is proportional to that of the parent solutions:

$$\left(x_i^{(2,t+1)} - x_i^{(1,t+1)}\right) = \beta_{q_i}\left(x_i^{(2,t)} - x_i^{(1,t)}\right). \tag{4.14}$$

This has an important implication. Let us consider two scenarios: (i) two parents are far away from each other, and (ii) two parents are closer to each other. For illustration, both of these cases (with parent solutions $x_i^{(1,t)} = 2.0$ and $x_i^{(2,t)} = 5.0$ in the first case and with parent solutions $x_i^{(1,t)} = 2.0$ and $x_i^{(2,t)} = 2.5$ in the second case) and the corresponding probability distributions with $\eta_c = 2$ are shown in Figures 61 and 62, respectively. For an identical random number u_i, the parameter β_{q_i} take the same value in both cases. From equation (4.14) it is clear that in the first case the offspring are likely to be more widely spread than in the second case. Figures 61 and 62 also show the corresponding offspring (marked with a box) for $u_i = 0.9$ or $\beta_{q_i} = 5^{1/3}$. These figures clearly show that if the parent values are far from each other (the first case), it is possible for solutions away from the parents to be created. Compare the offspring $x_i^{(1,t+1)} = 0.935$ with parent $x_i^{(1,t)} = 2.0$. However, if the parent values are close by (the second case), distant offspring are not likely. Compare the offspring $x_i^{(1,t+1)} = 1.822$ with parent $x_i^{(1,t)} = 2.0$ created by using the same random number as in the first case. In initial populations, where the solutions are randomly placed (like the first case), this allows almost any value to be created as an offspring. However, when the solutions tend to converge due to the action of genetic operators (like the second case), distant solutions are not allowed, thereby focusing the search to a narrow region.

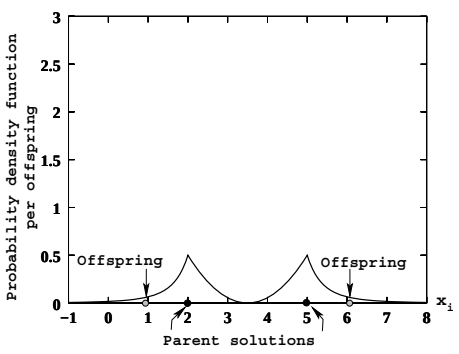


Figure 61 Probability distribution of offspring with distant parents.

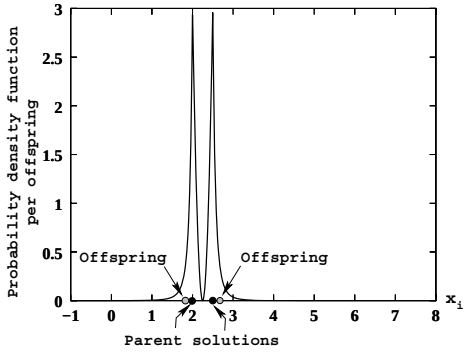


Figure 62 Probability distribution of offspring with closely spaced parents.

It is interesting to note that both equations (4.12) and (4.13) can be written in the form of equation (4.7) with the following relationship: $\gamma_i = 0.5(1 \mp \beta_{q_i})$. However, it is important that, unlike in the BLX-α operator, the equivalent γ_i term in the SBX operator is not uniformly distributed (refer to equation (4.11)). The SBX operator biases solutions near each parent more favorably than solutions away from the parents. Essentially, the SBX operator has two properties:

1. The difference between the offspring is in proportion to the parent solutions.
2. Near-parent solutions are monotonically more likely to be chosen as offspring than solutions distant from parents.

The SBX operator is also extended for bounded variables (Deb, 2000). Using the SBX operator, a mixed-integer programming algorithm with GAs (GeneAS) was introduced and applied to a number of engineering design problems (Deb, 1997a).

Fuzzy Recombination Operator

Voigt et al. (1995) suggested a fuzzy recombination (FR) operator, which is similar to the above SBX operator. In fact, the FR operator can be thought of as a special case of the SBX operator with $\eta_c = 1$. Like other recombination operators, this operator also has a tunable parameter d. First, the difference between two parent solutions is computed. Thereafter, a triangular probability distribution is used around each parent. This distribution has its apex located at the parent solution and the base is proportional to the difference in parents or $d(x_i^{(2,t)} - x_i^{(1,t)})$, as shown in Figure 63. Like the SBX operator, this operator also gives importance to the creation of solutions closer to the parents than away from them. The distribution can be changed with the parameter d. As d is large, solutions away from the parents can get created. However, if d is small, only near-parent solutions are allowed. Like the BLX and SBX operators, this operator also creates distant solutions in proportion to the spread of the parent solutions.

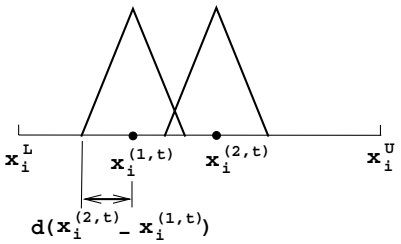


Figure 63 The FR operator. Parents are marked by filled circles.

Unimodal Normally Distributed Crossover

Ono and Kobayashi (1997) suggested a unimodal normally distributed crossover (UNDX) operator, where three or more parent solutions are used to create two or more offspring. Offspring are created from an ellipsoidal probability distribution with one axis formed along the line joining two of the three parent solutions and the extent of the orthogonal direction is decided by the perpendicular distance of the third parent from the axis (Figure 64). Unlike the BLX operator, this operator assigns more probability for creating solutions near the center of the first two parents than near the parents themselves. Recently, a multi-parental UNDX operator with more than three parents has also been suggested (Kita et al., 1998). Like the BLX operator, this operator also assigns offspring proportional to the spread of the parent solutions.

Simplex Crossover

In the simplex crossover (Tsutsui et al., 1999), more than two parents participate in the crossover operation. As many as $(n+1)$ parents (where n is the number of decision variables) are used as a simplex (representing a region enclosed by the points). The centroid of these parents are first calculated. Thereafter, the simplex is enlarged by extending the apex of the simplex at points away from the centroid. Each apex is

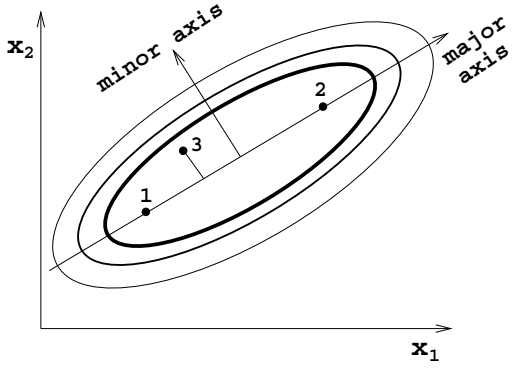


Figure 64 The UNDX operator.

placed on a line joining the centroid and each parent. Thereafter, with a uniform probability distribution, a number of solutions ($H = 200$ is suggested) are created from this extended simplex. Figure 65 illustrates this procedure. Investigators have suggested replacing only two of the parents by two solutions from a set of H new solutions and the two parent solutions. The first offspring is the best solution of H randomly created solutions and the other is chosen by using a rank-based roulette-wheel selection from the set H, excluding the already chosen first offspring. Since, at least n solutions are needed to *span* an n-dimensional search space, the use of $(n+1)$ solutions in the simplex seems like a good idea.

On a number of different correlated problems, investigators have shown that their GA with the simplex crossover can find increased precision in the obtained solution much faster than GAs with the UNDX operator. Moreover, they argue that this crossover operator is invariant in terms of linear coordinate transformation. Since the choice of two offspring from H new solutions is different from the usual implementation of creating two offspring from two parent solutions, a straightforward comparison of this crossover operator with the UNDX operator or any other real-parameter crossover operator (such as SBX or BLX) is difficult.

Fuzzy Connectives Based Crossover

This operator (Herrara et al., 1995) is also applied variable by variable. For each variable x_i, three regions are marked: region I $(x_i^{(L)}, x_i^{(1,t)})$, region II $(x_i^{(1,t)}, x_i^{(2,t)})$ and region III $(x_i^{(2,t)}, x_i^{(U)})$ (here, we assume that $x_i^{(1,t)} < x_i^{(2,t)}$). The fourth region is an overlapping region $(y_i^{(1)}, y_i^{(2)})$ with all of the above three regions. Here, $y_i^{(1)} \leq x_i^{(1,t)}$ and $y_i^{(2)} \geq x_i^{(2,t)}$. All of these four regions are shown in Figure 66. Once these regions are identified, one solution is chosen from each of them by using user-defined fuzzy

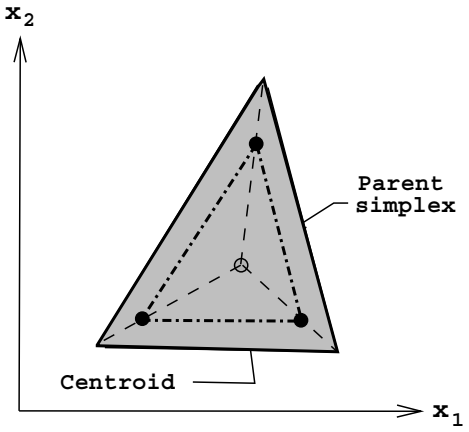


Figure 65 The SPX operator. The parents are marked by filled circles.

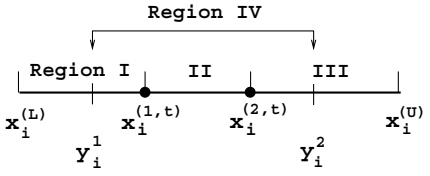


Figure 66 Four regions for the FCB crossover operator. The parents are marked by filled circles.

connective functions, which are defined over two normalized parents:

$$s = \frac{x_i^{(1,t)} - x_i^{(L)}}{x_i^{(2,t)} - x_i^{(1,t)}}, \quad s' = \frac{x_i^{(2,t)} - x_i^{(L)}}{x_i^{(2,t)} - x_i^{(1,t)}}.$$

Four different connectives (T, G, P, and $\hat{C}$) are used in different regions:

$$\text{Region I: } x_i^{(1,t+1)} = x_i^{(L)} + (x_i^{(U)} - x_i^{(L)})T(s,s'), \tag{4.15}$$

$$\text{Region II: } x_i^{(2,t+1)} = x_i^{(L)} + (x_i^{(U)} - x_i^{(L)})G(s,s'), \tag{4.16}$$

$$\text{Region III: } x_i^{(3,t+1)} = x_i^{(L)} + (x_i^{(U)} - x_i^{(L)})P(s,s'), \tag{4.17}$$

$$\text{Region IV: } x_i^{(4,t+1)} = x_i^{(L)} + (x_i^{(U)} - x_i^{(L)})\hat{C}(s,s'). \tag{4.18}$$

Investigators have suggested different forms for these connectives. By applying this approach to a number of test problems, they suggested that the following connectives give the best performance:

$$T(s,s') = \min(s,s'), \qquad G(s,s') = \max(s,s'),$$
$$P(s,s') = (1-\lambda)s + \lambda s' \quad \hat{C}(s,s') = T^{1-\lambda}G^{\lambda}.$$

A list of other fuzzy connectives can be found from the original study (Herrera et al., 1998). Of the four solutions, two of these directly replace the two parent solutions. The other two solutions replace two random solutions taken from the rest of the population members waiting to undergo a crossover. It is clear that the performance of this crossover depends on the chosen fuzzy connective functions. Compared to other crossover operators, it is interesting to ponder whether such complicated arithmetic operators are needed to make the crossover operator efficient.

Unfair Average Crossover

In the unfair average crossover (Nomura and Miyoshi, 1996), two parents are used and two offspring are created. However, unlike the SBX or BLX operators, this crossover operator does not preserve the mean of the parent solutions, but instead will create offspring towards one of the parent solutions. Unless this favored parent solution is the better of the two parent solutions, introduction of such a bias is hard to justify. Specifically, the following equations can be used to create offspring with $\alpha \in [0, 0.5]$:

$$x_i^{(1,t+1)} = \begin{cases} (1+\alpha)x_i^{(1,t)} - \alpha x_i^{(2,t)}, & \text{for } i = 1,\ldots,j, \\ -\alpha x_i^{(1,t)} + (1+\alpha)x_i^{(2,t)}, & \text{for } i = j+1,\ldots,n. \end{cases} \tag{4.19}$$

$$x_i^{(2,t+1)} = \begin{cases} (1-\alpha)x_i^{(1,t)} + \alpha x_i^{(2,t)}, & \text{for } i = 1,\ldots,j, \\ \alpha x_i^{(1,t)} + (1-\alpha)x_i^{(2,t)}, & \text{for } i = j+1,\ldots,n. \end{cases} \tag{4.20}$$

The parameter j is a randomly chosen integer between 1 and n, indicating the cross site. For two variables ($n = 2$) and with a cross site at $j = 2$, the bias in creating offspring is clearly shown in Figure 67. The shaded region marks the possible location of the offspring.

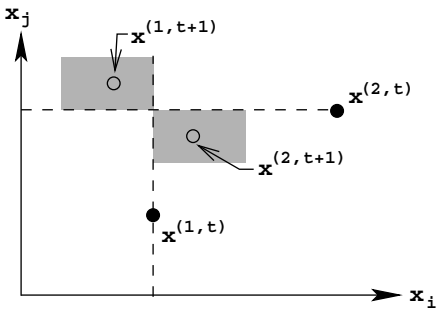


Figure 67 The unfair crossover operator illustrated on a two-dimensional search space. The parents are marked by filled circles.

Similarity in Different Crossovers

The plethora of different real-parameter crossover operators discussed above clearly shows the importance and the need for real-parameter GAs. However, a simple thought also comes to mind: 'Are all these crossover operators different from each other?' There exist a number of theoretical studies involved with trying to understand the properties of different crossover operators (Nomura, 1997; Qi and Palmieri, 1993). A recent study (Beyer and Deb, 2000) has found similarity in all of these operators.

The issue of 'this crossover operator' versus 'that crossover operator' is really context-dependent. The crossover cannot be viewed as an independent operator. The issue of exploitation and exploration makes a crossover operator dependent on the chosen selection operator for a successful GA run. Viewed in this way, Beyer and Deb argued that in most situations the selection operator reduces the population diversity, because selection eliminates some population members by making duplicates of a few others. Figure 68 shows a population, one instance of the effect of a selection operator on this population, and one instance of the effect of a recombination operator on the mating pool. The extent of reduction in diversity due to the selection operator can be related to the exploitation power of the selection operator. Thereafter, when a crossover operator is acted on the population found after the selection operator, it should enhance the population diversity in general. Such a balance between the reduction and enhancement of population diversity will allow a GA to have an adequate search property. It is important to highlight that a balance between exploitation and exploration is also achieved from the characteristic time of each operator for binary GAs (refer to page 104). Based on these arguments, two postulations have been recently suggested (Beyer and Deb, 2000). Under a crossover operator:

1. the population mean should not change;
2. the population diversity should increase, in general.

Although not intuitive, the first postulation is desired. Since most crossover operators

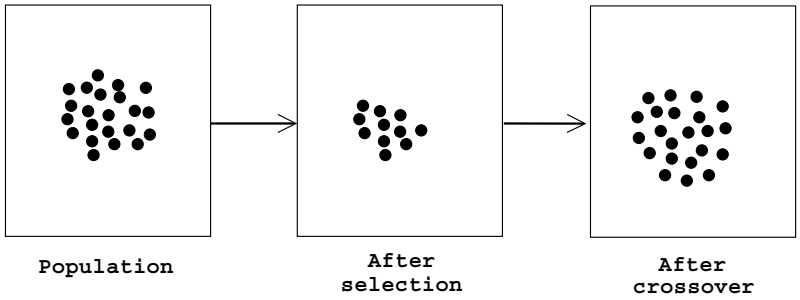


Figure 68 When selection reduces the diversity, the recombination operator must increase the diversity.

do not use any fitness information, there is no reason for a crossover operator to steer the search in any particular direction. The best that a crossover operator can do is to keep the mean of the offspring population the same as that of the parent population and alter the population variance. The second postulation also makes sense. Since a selection operator reduces the population variance in general, if a crossover operator also reduces the population variance, the combined action may lead to a premature convergence. To counterbalance the reduction in population variance due to the selection operator, the crossover operator should increase the population variance in general. Beyer and Deb (2000) gave arguments for an increase in population diversity for several types of objective functions. In the light of the above two postulates, it will suffice to characterize a crossover operator with two factors: the shift in the population mean and the growth of population variance with generations. The study calculated the population mean and variance under three real-parameter crossover operators in terms of their characteristic parameters. By equating the variance growth of the SBX, BLX and FR operators, the investigators obtained four sets of characteristic parameter values (shown in Table 6) producing identical variance growth in a flat fitness landscape. This means that on a flat fitness landscape, a GA with SBX-2, BLX-0.662, and FR-1.095 will have similar performances. Although in other fitness

Table 6 Parameter values for an identical population variance growth of three crossover operators on a flat fitness landscape.

SBX	BLX	FR
η_c	α	d
2	0.662	1.095
3	0.500	0.707
5	0.419	0.433
10	0.381	0.226

functions, these parameters values may not produce exactly the same performance, the above argument suggests that there exist characteristic parameter values which will make these crossover operators equivalent. In other words, all of the mean-preserving crossover operators can be expected to show similar behavior for certain characteristic parameter settings of each operator. However, the generic FCB crossover operator and the unfair average crossover operator discussed above do not preserve the population mean. There may be a difference in performance of GAs with these crossover operators with GAs with mean-preserving crossover operators. It is mentioned earlier that the motivation for using a crossover operator which does not preserve the mean of the parent population in the created offspring population is weak.

Since real-parameter crossover operators directly manipulate two or more real numbers to create one or more real numbers as offspring, one may wonder whether there is a special need for using another mutation operator. The confusion arises because both operators seem to be doing the same task, i.e. perturb every solution in the parent population to create a new population. The distinction among operators lies in the extent of perturbation allowed in each operator. Although debatable, this author believes the distinction between a crossover and a mutation operator lies mainly in the number of parent solutions used in the perturbation process. If only one parent solution is used, it should be called a mutation operation. With only one parent, a range of perturbations must be predefined. However, if more than one parent solution is used in the perturbation process, the range of perturbation may be adaptive and can be determined from the diversity of the parents on the fly. If the diversity in participating parents is large, the crossover can create offspring which are also diverse with respect to each other. Thus, with a crossover operator, it is possible to achieve adaptively large or small perturbations without any predefined setting of the range of perturbation. As in the binary-coded GA a mutation is meant to have a local perturbation, in real-parameter GAs a local perturbation in a predefined manner can also be useful in maintaining diversity in a population. In the following, we will mention some of the most commonly used real-parameter mutation operators.

Random Mutation

The simplest mutation scheme would be to create a solution randomly from the entire search space (Michalewicz, 1992):

$$y_i^{(1,t+1)} = r_i(x_i^{(U)} - x_i^{(L)}), \tag{4.21}$$

where r_i is a random number in $[0, 1]$. Figure 69 shows this probability distribution with a continuous line. This operator is independent of the parent solution and is equivalent to a random initialization. Instead of creating a solution from the entire search space, a solution in the vicinity of the parent solution with a uniform probability distribution (shown with a dashed line in Figure 69) can also be chosen:

$$y_i^{(1,t+1)} = x_i^{(1,t)} + (r_i - 0.5)\Delta_i, \tag{4.22}$$

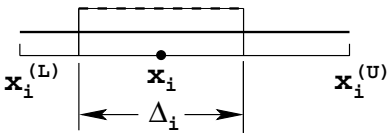


Figure 69 The random mutation operator.

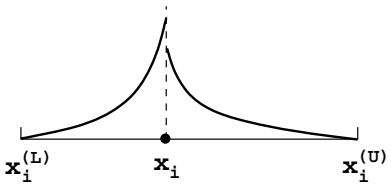


Figure 70 The non-uniform mutation operator.

where Δ_i is the user-defined maximum perturbation allowed in the i-th decision variable. Care should be taken to check if the above calculation takes $y_i^{(1,t+1)}$ outside of the specified lower and upper limits. Instead of using a uniform probability distribution, non-uniform probability distributions have also been suggested. In the following subsections, we will present a few such mutation operators.

Non-Uniform Mutation

Here, the probability of creating a solution closer to the parent is more than the probability of creating one away from it (Figure 70). However, as the generations (t) proceed, this probability of creating solutions closer to the parent gets higher and higher (Michalewicz, 1992):

$$y_i^{(1,t+1)} = x_i^{(1,t+1)} + \tau(x_i^{(U)} - x_i^{(L)}))\left(1 - r_i^{(1-t/t_{max})^b}\right). \tag{4.23}$$

Here, τ takes a boolean value, -1 or 1, each with a probability of 0.5. The parameter t_{max} is the maximum number of allowed generations, while b is a user-defined parameter. In this way, from early on the above mutation scheme acts like a uniform distribution, while in later generations it acts like Dirac's function, thus allowing a focussed search.

Normally Distributed Mutation

A simple and popular method is to use a zero-mean Gaussian probability distribution:

$$y_i^{(1,t+1)} = x_i^{(1,t+1)} + N(0, \sigma_i). \tag{4.24}$$

Here, the parameter σ_i is a fixed, user-defined parameter. This parameter is important and must be correctly set in a problem. Such a parameter can also be adaptively changed in every generation by some predefined rule. This operator is identical to the mutation operator used in evolution strategy (ES). However, the latter also has a self-adaptive mutation operator, where the mutation strength is evolved with the decision variables (Fogel et al., 1995; Hansen and Ostermeier, 1996; Herdy, 1992; Sarvanan et al., 1995; Schwefel, 1987b). For handling bounded decision variables, special care

must be taken to make sure that no solution is created outside of the specified lower and upper limits.

Polynomial Mutation

Like in the SBX operator, the probability distribution can also be a polynomial function, instead of a normal distribution (Deb and Goyal, 1996):

$$y_i^{(1,t+1)} = x_i^{(1,t+1)} + (x_i^{(U)} - x_i^{(L)})\bar{\delta}_i, \tag{4.25}$$

where the parameter $\bar{\delta}_i$ is calculated from the polynomial probability distribution $\mathcal{P}(\delta) = 0.5(\eta_m + 1)(1 - |\delta|)^{\eta_m}$:

$$\bar{\delta}_i = \begin{cases} (2r_i)^{1/(\eta_m+1)} - 1, & \text{if } r_i < 0.5, \\ 1 - [2(1 - r_i)]^{1/(\eta_m+1)}, & \text{if } r_i \geq 0.5. \end{cases} \tag{4.26}$$

This distribution is similar to the non-uniform mutation operator, although a fixed value of the parameter η_m is suggested here. It is observed that a value of η_m produces a perturbation of the order $O(1/\eta_m)$ in the normalized decision variable. For handling bounded decision variables (Deb and Agrawal, 1999a), the mutation operator is modified for two regions, i.e. $[x_i^{(L)}, x_i]$ and $[x_i, x_i^{(U)}]$, very similar to the non-uniform mutation described above. The difference is that in this mutation operator the shape of the probability distribution is directly controlled by an external parameter η_m and the distribution is not dynamically changed with generations.

Theory of Real-Parameter Genetic Algorithms

Although real-parameter GAs deal with real parameter values and bring the GA technique a step closer to classical optimization algorithms, there does not exist many theoretical studies on real-parameter GAs. In the following paragraphs, we will discuss a couple of salient studies.

Goldberg (1991) argued that in a real-parameter GA the action of a selection operator causes the above-average solutions to survive in the population. In each decision variable, these sub-regions of above-average solutions constitute *virtual characters* of a virtual alphabet. For example, Figure 71 shows that the first and second decision variables have virtual alphabets with a cardinality of three and two, respectively. Thus, after a few generations, a real-parameter GA treats a continuous search space problem as a discrete search space problem and the search is then similar to a binary-coded GA with low cardinal alphabets. In such an event, the search gets restricted to certain islands in the search space. Although mutation can allow some diffusion, the mutation strength must be large enough to create solutions outside of the width of the virtual characters. Based on this virtual alphabet theory, Goldberg defined a number of *blocking* functions, where the global optimum lies in a region not marked by the virtual characters. Since a real-parameter GA will soon converge to intersections of virtual characters of different decision variables, the global optimum will be difficult to find.

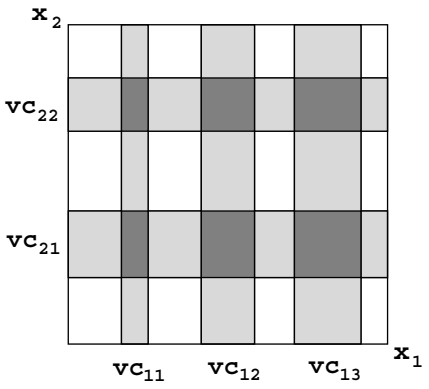


Figure 71 Virtual characters are shown in a two-variable space. The dark-shaded regions are intersections of virtual hyperplanes.

Qi and Palmieri (1993) made an extensive study on different combinations of real-parameter GA operators by using a discrete-time stochastic process. The use of a large population assumption allowed the investigators to find the probability density functions $\pi_t(\cdot)$, characterizing the distribution of the population at any generation t. Thereafter, by letting the population size N go to infinity, the limiting behavior of the GA operator interactions was found. The salient features of their study are as follows:

1. For a GA with selection and mutation operators, the selection operator tends to reduce the variance of $\pi_t(\cdot)$, while mutation spreads the distribution. However, the combined effect is a shift towards the optimum in a unimodal problem. With a large number of generations, the distribution $\pi_t(\cdot)$ approaches a Dirac function, $\delta(\mathbf{x} - \mathbf{x}^*)$[3].
2. The second observation is that a GA with a selection operator alone monotonically increases the density of solutions in the regions with above-average fitness.
3. The third observation is that a repetitive application of a crossover operator alone decorrelates the decision variables, leading to:

$$\lim_{t\to\infty} \pi_t(x_1, x_2, \ldots, x_n) = \Pi_{i=1}^{n} \pi_0(x_i), \tag{4.27}$$

   where $\pi_0(x_i)$ is the marginal density function of x_i.
4. The fourth observation is that a combined effect of selection and crossover allows hyperplanes of higher average fitness value to be sampled more often. The consequence of this is that intersection of these hyperplanes allows better solutions to be sampled more often. This can be considered as the real-coded version of the building-block hypothesis. A hyperplane representing a larger region is equivalent to a low-order schema in binary-coded GAs. Intersection of hyperplanes (low-order schemata) signifies smaller regions (higher-order schemata) in the context of real-parameter GAs.

4.2.3 Constraint-Handling in Genetic Algorithms

In the example problem illustrated in Section 4.2.1, we have handled the volume constraint by adding a penalty proportional to the constraint violation to the objective function value. In minimization problems, this is a common way of handling constraints in evolutionary algorithms. However, in this section we will describe a number of other efficient methods for handling constraints. As outlined elsewhere (Michalewicz and Schoenauer, 1996; Michalewicz et al., 2000), most constraint handling methods which exist in the literature can be classified into five categories, as follows:

1. Methods based on preserving feasibility of solutions.
2. Methods based on penalty functions.
3. Methods biasing feasible over infeasible solutions.
4. Methods based on decoders.
5. Hybrid methods.

We will describe these methods briefly in the following subsections.

Methods Based on Preserving Feasibility of Solutions

In this approach, two feasible solutions, after recombination and mutation operations, will create two feasible offspring. This can be achieved in a number of different ways.

First, one decision variable can be eliminated by using an equality constraint either implicitly or explicitly. In this way, the equality constraint is always satisfied. For example, for the following equality constraint:

$$h(\mathbf{x}) \equiv 2x_1 - x_2^2 x_3 = 0,$$

the variable x_1 can be eliminated by the following equation: $x_1 = 0.5x_2^2 x_3$ in the objective function and in all other constraints. In this way, an EA has to use one decision variable less. This also allows the above equality constraint to be satisfied automatically in all solutions used in the optimization process. Similarly, for K linear simultaneous equalities, K decision variables can be eliminated (only $(n-K)$ decision variables need to be used in an EA), thereby automatically satisfying all K linear equality constraints. In the implicit method of handling an equality constraint $h(\mathbf{x})$, a variable x_k can be calculated from the constraint by finding the root of:

$$h\left(x_k, (\mathbf{x}\backslash x_k)^{(i)}\right) = 0, \tag{4.28}$$

where $(\mathbf{x}\backslash x_k)^{(i)}$ is the $(n-1)$ real-parameter vector (without x_k) corresponding to the i-th solution. Here, an EA uses all n decision variables, but in finding the root of the above equation, the value of x_k is not taken from the individual vector; instead, the estimated root of the above equation is used as a value of x_k. There are two ways to handle this new value of x_k. In the Lamarckian method, the new value is substituted in the i-th individual vector, while in the Baldwinian method, the new value is not

substituted. However, both methods use the new value to compute objective function and other constraints. Although the Lamarckian method of repairing an infeasible solution seems desirable (Deb and Goel, 2001b), the Baldwinian method is found to be more efficient in some problems (Liu et al., 2000).

Since in most interesting optimization problems, the optimal solution lies on the intersection of constraint boundaries, operators that search only in the boundary area between feasible and infeasible region are found to be efficient in certain problems (Michalewicz and Schoenauer, 1996).

Methods Based on Penalty Functions

In most applications, an exterior penalty term (Deb, 1995; Reklaitis et al., 1983), which penalizes infeasible solutions, is used. Based on the constraint violation $g_j(\mathbf{x})$ or $h_k(\mathbf{x})$, a bracket-operator penalty term is added to the objective function and a penalized function is formed:

$$F(\mathbf{x}) = f(\mathbf{x}) + \sum_{j=1}^{J} R_j \langle g_j(\mathbf{x}) \rangle + \sum_{k=1}^{K} r_k |h_k(\mathbf{x})|, \tag{4.29}$$

where R_j and r_k are user-defined penalty parameters. The bracket-operator $\langle\ \rangle$ denotes the absolute value of the operand, if the operand is negative. Otherwise, if the operand is non-negative, it returns a value of zero. Since different constraints may take different orders of magnitude, it is essential to normalize all constraints before using the above equation. A constraint $\underline{g}_j(\mathbf{x}) \geq b_j$ can be normalized by using the following transformation:

$$g_j(\mathbf{x}) \equiv \underline{g}_j(\mathbf{x})/b_j - 1 \geq 0.$$

Equality constraints can also be normalized similarly. Normalizing constraints in the above manner has an additional advantage. Since all normalized constraint violations take more or less the same order of magnitude, they all can be simply added as the overall constraint violation and thus only one penalty parameter R will be needed to make the overall constraint violation of the same order as the objective function:

$$F(\mathbf{x}) = f(\mathbf{x}) + R \left[\sum_{j=1}^{J} \langle g_j(\mathbf{x}) \rangle + \sum_{k=1}^{K} |h_k(\mathbf{x})| \right]. \tag{4.30}$$

When an EA uses a fixed value of R in the entire run, the method is called the *static* penalty method. There are two difficulties associated with this static penalty function approach:

1. The optimal solution of $F(\mathbf{x})$ depends on penalty parameters R. Users usually have to try different values of R to find which value would steer the search towards the feasible region. This requires extensive experimentation to find any reasonable solution. This problem is so severe that some researchers have used different values of R depending on the level of constraint violation (Homaifar et al., 1994), while

some have used a sophisticated temperature-based evolution of penalty parameters through generations (Michalewicz and Attia, 1994) involving a few parameters describing the rate of evolution. We will discuss these *dynamically* changing penalty methods a little later.

2. The inclusion of the penalty term *distorts* the objective function (Deb, 1995). For small values of R, the distortion is small, but the optimum of $F(\mathbf{x})$ may not be near the true constrained optimum. On the other hand, if a large R is used, the optimum of $F(\mathbf{x})$ is closer to the true constrained optimum, but the distortion may be so severe that $F(\mathbf{x})$ may have artificial locally optimal solutions. This primarily happens due to interactions among multiple constraints. EAs are not free from the distortion effect caused due to the addition of the penalty term in the objective function. However, EAs are comparatively less sensitive to distorted function landscapes due to the stochasticity in their operators.

In order to investigate the effect of the penalty parameter R on the performance of EAs, we consider a well-studied welded beam design problem (Reklaitis et al., 1983). Figure 72 shows the configuration of the welded-beam problem. The resulting optimization problem has four design variables, $\mathbf{x} = (h, \ell, t, b)$, and five inequality constraints, involving different stress, buckling and logical limitations:

$$\left.\begin{array}{rl} \text{Minimize} & f_w(\mathbf{x}) = 1.104\,71 h^2 \ell + 0.048\,11 tb(14.0 + \ell), \\ \text{Subject to} & g_1(\mathbf{x}) \equiv 13\,600 - \tau(\mathbf{x}) \geq 0, \\ & g_2(\mathbf{x}) \equiv 30\,000 - \sigma(\mathbf{x}) \geq 0, \\ & g_3(\mathbf{x}) \equiv b - h \geq 0, \\ & g_4(\mathbf{x}) \equiv P_c(\mathbf{x}) - 6000 \geq 0, \\ & g_5(\mathbf{x}) \equiv 0.25 - \delta(\mathbf{x}) \geq 0, \\ & 0.125 \leq h \leq 10, \\ & 0.1 \leq \ell, t, b \leq 10. \end{array}\right\} \tag{4.31}$$

The terms $\tau(\mathbf{x})$, $\sigma(\mathbf{x})$, $P_c(\mathbf{x})$ and $\delta(\mathbf{x})$ are given below:

$$\begin{aligned} \tau(\mathbf{x}) &= \sqrt{(\tau'(\mathbf{x}))^2 + (\tau''(\mathbf{x}))^2 + \ell\tau'(\mathbf{x})\tau''(\mathbf{x})/\sqrt{0.25[\ell^2 + (h+t)^2]}}, \\ \sigma(\mathbf{x}) &= \frac{504\,000}{t^2 b}, \end{aligned}$$

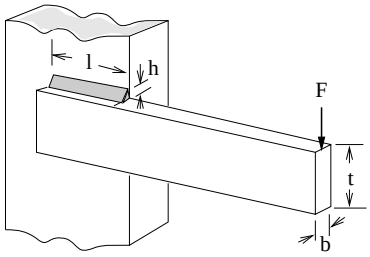


Figure 72 The welded beam design problem.

$$P_c(\mathbf{x}) = 64\,746.022(1 - 0.028\,234\,6t)tb^3,$$
$$\delta(\mathbf{x}) = \frac{2.1952}{t^3 b},$$

where:

$$\tau'(\mathbf{x}) = \frac{6000}{\sqrt{2}h\ell},$$
$$\tau''(\mathbf{x}) = \frac{6000(14 + 0.5\ell)\sqrt{0.25[\ell^2 + (h+t)^2]}}{2\{0.707h\ell[\ell^2/12 + 0.25(h+t)^2]\}}.$$

The optimized solution reported in the literature (Reklaitis et al., 1983) is $h^* = 0.2444$, $\ell^* = 6.2187$, $t^* = 8.2915$, and $b^* = 0.2444$, with a function value equal to $f^* = 2.38116$. Binary GAs are applied to this problem in an earlier study (Deb, 1991) and the solution $\mathbf{x} = (0.2489, 6.1730, 8.1789, 0.2533)$ with $f = 2.43$ (within 2% of the above best solution) was obtained with a population size of 100. However, it was observed that the performance of the GAs largely depended on the chosen penalty parameter values.

In order to obtain further insights into the working of the GAs, we apply binary GAs with the tournament selection and a single-point crossover operator with $p_c = 0.9$ on this problem. Each variable is coded in 10 bits, so that the total string length is 40. A population size of 80 is used and GAs with 50 different initial populations are run until 500 generations have been achieved. All constraints are normalized and a single penalty parameter R is used. Table 7 shows the performance of such binary GAs for different penalty parameter values. For each case, the best, median (the optimized objective function values of 50 runs are arranged in ascending order and the 25th value in the list is called the median optimized function value) and worst values of 50 optimized objective function values are also shown in the table. With $R = 1$, although three out of 50 runs have found a solution within 10% of the best-known solution, 13 GA runs have not been able to find a single feasible solution in 40 080 function evaluations. This happens because with a small R there is not much pressure for the solutions to become feasible. With a large value of R, the pressure

Table 7 The number of runs (out of 50) converged within ϵ% of the best-known solution are shown when using binary GAs with different penalty parameter values on the welded beam design problem.

R	ϵ							Optimized $f_w(\mathbf{x})$		
	$\leq$ 1%	$\leq$ 2%	$\leq$ 10%	$\leq$ 20%	$\leq$ 50%	$>$ 50%	Infeasible	Best	Median	Worst
10^0	0	1	3	3	12	25	13	2.41	7.62	483.50
10^1	0	0	0	0	12	38	0	3.14	4.33	7.45
10^3	0	0	0	0	1	49	0	3.38	5.97	10.66
10^6	0	0	0	0	0	50	0	3.73	5.88	9.42

for solutions to become feasible is more and all 50 runs then found feasible solutions. However, because of the larger emphasis assigned for solutions to become feasible, a feasible solution has a large selective advantage over an infeasible solution. If new and different feasible solutions are not created, GAs would overemphasize this sole feasible solution and soon prematurely converge near this solution. This is exactly what has happened in GA runs with larger R values, where the best solution obtained, in most cases, is more than 50% away (in terms of function values) from the true constrained optimum. Similar experiences have been reported by other researchers in applying GAs with an exterior penalty function approach to other constrained optimization problems. Thus, if the penalty function method is to be used, the user usually have to take many runs or 'adjust' the penalty parameters to get a solution within an acceptable limit.

There exist a number of *dynamic* penalty methods where the penalty parameter is changed with the generation counter t (Joines and Houck, 1994):

$$F(\mathbf{x}) = f(\mathbf{x}) + (C \cdot t)^{\alpha} \left(\sum_{j=1}^{J} \langle g_j(\mathbf{x}) \rangle^{\beta} + \sum_{k=1}^{K} |h_k(\mathbf{x})|^{\beta} \right), \tag{4.32}$$

where C, α and β are user-defined constants. A value of $\beta = 2$ is suggested. The penalty parameter R can be updated from the last k generations of population statistics and updated at every generation. However, both static and dynamic penalty methods require exogenous parameters, which must be tuned for every new problem.

Methods Based on Feasible Over Infeasible Solutions

In these methods, a clear distinction between feasible and infeasible solutions in the search space is made. In the *behavioral memory method* (Schoenauer and Xanthakis, 1993), all constraints are considered in a sequence. When sufficient feasible solutions satisfying the constraints taken from the top of the sequence are found, the next constraint from the sequence is considered. In this way, each constraint gets its turn to divert the population members in the region of its feasibility.

In another approach (Powell and Skolnick, 1993), an additional generation-dependent term is added to the static penalty term as follows:

$$F(\mathbf{x}) = f(\mathbf{x}) + R \left(\sum_{j=1}^{J} \langle g_j(\mathbf{x}) \rangle + \sum_{k=1}^{K} |h_k(\mathbf{x})| \right) + \lambda(t, \mathbf{x}). \tag{4.33}$$

At every generation, the term $\lambda(t, \mathbf{x})$ is the difference between the best static penalized function value among all infeasible solutions and the worst feasible solution. This would make the best infeasible solution in the population to have the same $F()$ value as that of the worst feasible solution in the population. For a hypothetical single-variable function $f(x)$ having one constraint $g(x) \geq 0$, the left plot of Figure 73 shows the construction of $F(x)$. This figure illustrates the procedure with six population

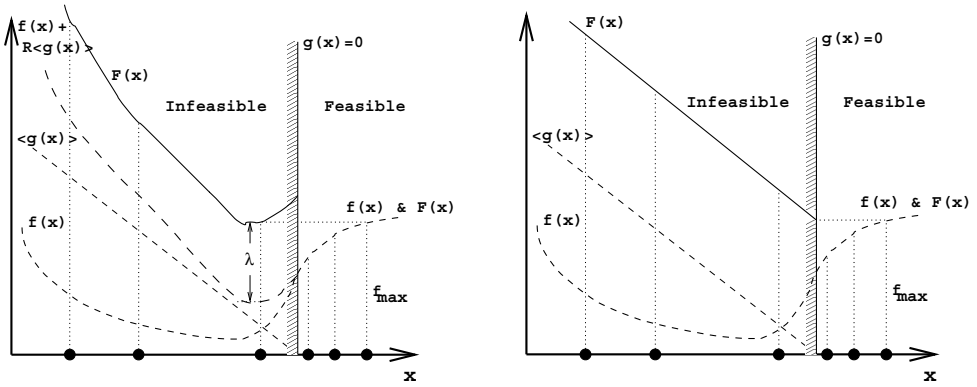


Figure 73 The left figure shows Powell and Skolnick's constraint handling method, while the right figure shows Deb's constraint handling method. Reprinted from Computer Methods in Applied Mechanics and Engineering, Volume 186, K. Deb, 'An efficient constraint handling method for genetic algorithms', Pages 311–338, Copyright 2000, with permission from Elsevier Science.

members, of which three are feasible. Notice that for all feasible solutions, $F(x) = f(x)$. As discussed in an another study (Deb, 2000), this method has at least two difficulties:

1. The need to fix a penalty parameter R still exists.
2. The fitness function $F(\mathbf{x})$, constructed by using the above approach, may have an artificial local optimum.

A recent study (Deb, 2000) suggested a modification, which does not need any penalty parameter:

$$F(\mathbf{x}) = \begin{cases} f(\mathbf{x}), & \text{if } \mathbf{x} \text{ is feasible;} \\ f_{max} + \sum_{j=1}^{J} \langle g_j(\mathbf{x}) \rangle + \sum_{k=1}^{K} |h_k(\mathbf{x})|, & \text{otherwise.} \end{cases} \tag{4.34}$$

Here, f_{max} is the objective function value of the worst feasible solution in the population. The right figure of Figure 73 shows the construction procedure of $F(x)$ from $f(x)$ and $g(x)$ for a single-variable objective function. One fundamental difference between this approach and the previous approach is that the objective function value is *not* computed for any infeasible solution. Since all feasible solutions have zero constraint violation and all infeasible solutions are evaluated according to their constraint violations only, both the objective function value and constraint violation are not combined in any solution in the population. Thus, there is no need to have any penalty parameter R for this approach. Moreover, the approach is also quite pragmatic. Since infeasible solutions are not to be recommended for use, there is no real reason for one to find the objective function value for an infeasible solution. The method uses a binary tournament selection operator, where two solutions are compared at a time, and the following scenarios are always assured (Goldberg, 1992):

1. Any feasible solution is preferred to any infeasible solution.

2. Among two feasible solutions, the one having a better objective function value is preferred.
3. Among two infeasible solutions, the one having a smaller constraint violation is preferred.

To steer the search towards the feasible region and then towards the optimal solution, the method recommends the use of a niched tournament selection (where two solutions are compared in a tournament only if their Euclidean distance is within a pre-specified limit). This ensures that even if a few isolated solutions are found in the feasible space, they will be propagated from one generation to another for maintaining diversity among the feasible solutions.

Using this penalty parameter-less constraint handling strategy, real-parameter GAs (with an SBX-2 operator), using a niched tournament selection and a population size of 80, has found 50 out of 50 runs within a maximum of 0.1% of the optimal objective function value in the welded beam design problem (Deb, 2000). This means that with the proposed approach, one run is enough to find a satisfactory solution close to the true optimal solution. The best, median and mean objective function values of 50 runs are 2.381 45, 2.382 63, and 2.383 55, respectively.

Methods Based on Decoders

In these approaches, a chromosome stores information about how to fix an infeasible solution. For example, the chromosome may keep information about an ordering of decision variables which may be altered to make a solution feasible. In some instances, a decoder imposes a mapping between a feasible solution and a decoded solution. One such example is the use of a homomorphous mapping between an n-dimensional cube and a feasible search space (Koziel and Michalewicz, 1998). The mapping allows every solution in the cube to be mapped into a feasible solution in the search space. Moreover, the entire feasible region is mapped in the cube. In this way, a search in the cube will result in a search in the feasible region only. In such methods, care must be taken in assuring that the computational complexity of the mapping is not too high. Moreover, the continuity in the decoded space (in the cube) must preserve the continuity of actual solutions in the feasible search region.

Hybrid Methods

Since classical constrained optimization methods have been studied and applied for many years, it may be wise to combine such a method with the existing operators of an evolutionary algorithm. In one application, the Hooke–Jeeves pattern search method is combined with evolutionary algorithms (Waagen et al., 1992). In the coevolutionary approach suggested by Paredis (1994), where a population of decision variable vectors is evolved along with a population of constraints, fitter solutions satisfy more constraints and fitter constraints are violated by fewer solutions. A different twist to this coevolutionary approach, suggested elsewhere (Barbosa, 1996),

uses a population of solutions and a population of Lagrange multipliers. Since the constrained optimum solutions are *saddle* points of the corresponding Lagrangian function, the coevolutionary approach helps to find the optimum solution along with optimum Lagrange multipliers. A knowledge of the optimal Lagrange multiplier is extremely helpful in performing post-optimal sensitivity analysis.

4.3 Evolution Strategies

More or less contemporary to the development of GAs, research on a field very similar to such algorithms was in progress in Germany. As a result of this work, P. Bienert, I. Rechenberg and H.-P. Schwefel of the Technical University of Berlin suggested the so-called *evolution strategy* (ES) during the early Sixties. The first applications of ES were experimental and attempted to solve the shape optimization of a bended pipe (Lichtfuss, 1965), the drag minimization of a jointed plate (Rechenberg, 1965), and the shape optimization of a flashing nozzle (Schwefel, 1968), along with various other problems. Since the evaluation of a solution in each of these problems was difficult and time-consuming, a simple two-membered ES was used in all of the early studies. However, Schwefel was the first to simulate a different version of the ES on a computer in 1965. Thereafter, multi-membered ESs, recombinative ESs, and self-adaptive ESs were all suggested. However, the early ES procedure is fundamentally different from binary GAs in mainly two ways: (i) ESs use real parameter values and (ii) early ESs do not use any crossover-like operator. However, an ES's working principle is similar to that of a real-parameter GA used with selection and mutation operators only.

Realizing the similarity between these two surprisingly similar (although practiced in two geographically distant places) procedures, recent ES studies have introduced crossover-like operators. In the following subsections, we will describe these approaches and discuss an important issue which is mostly studied in the context of evolution strategies.

4.3.1 Non-Recombinative Evolution Strategies

In this section, we describe a number of different ESs where no recombination operator is used.

Two-Membered ESs: (1+1)-ES

This is the simplest of all ESs. In each iteration, one parent is used to create one offspring by using a Gaussian mutation operator. The step-by-step procedure is shown below. (The procedure is given for solving minimization problems.)

Two-Membered ES: (1+1)-ES

Step 1 Choose an initial solution $\mathbf{x}$ and a mutation strength σ.

Step 2 Create a mutated solution:

$$\mathbf{y} = \mathbf{x} + \mathbf{N}(0, \sigma), \tag{4.35}$$

where $\mathbf{N}(0,\sigma)$ is a vector of instances created using a zero-mean normal distribution and having a standard deviation σ.

Step 3 If $f(\mathbf{y}) < f(\mathbf{x})$, replace $\mathbf{x}$ with $\mathbf{y}$.

Step 4 If a termination criterion is satisfied, stop. Otherwise, go to Step 2.

Here, all decision variables are mutated with a normal distribution having the same mutation strength (a term used for the 'standard deviation' in ES research). It is intuitive that the success of this algorithm in finding a solution close to the true optimum solution largely depends on the chosen σ value. For a few simple problems, the optimal σ values are calculated for obtaining the maximum rate of convergence towards the optimum solution. In order to find the minimum of a linear function $f(\mathbf{x}) = c_0 + c_1 x_1$ in the range $-b/2 \leq x_i \leq b/2$ for $i = 1, 2, \ldots, n$, Rechenberg (1973) calculated the optimal mutation strength:

$$\sigma^* = \sqrt{\frac{\pi}{2}\frac{b}{n}}. \tag{4.36}$$

For a sphere model $f(\mathbf{x}) = \sum_{i=1}^{n} x_i^2$, the optimal mutation strength is found as follows:

$$\sigma^* = 1.224\frac{r}{n}, \tag{4.37}$$

where r denotes the current Euclidean distance from the true optimum solution. It is interesting to note that for this function the optimal mutation strength must change dynamically. The chosen mutation strength must be inversely proportional to the number of variables and as the solution comes closer and closer to the optimum, the mutation strength should reduce proportionately. This phenomenon is exploited in developing *self-adaptive* ESs, where in addition to the decision variables, mutation strength is also evolved in the ES. We will discuss self-adaptive ESs in Section 4.3.3 below.

Using these two calculations, the probabilities of a successful mutation occurring in both functions are found to be 0.184 and 0.270. Based on these numbers, Rechenberg postulated the famous '1/5 success rule', where the mutation strength σ is modified across generations (Rechenberg, 1973, p. 123):

> The ratio of successful mutations to all mutations should be 1/5. If this ratio is greater than 1/5, increase the mutation strength, while if it is less than 1/5, decrease the mutation strength.

A mutation is defined as successful if the mutated offspring is better than the parent solution. If many successful mutations are observed, this indicates that the solutions are residing in a better region in the search space. Hence, it is time to increase the mutation strength in the hope of finding even better solutions closer to the optimum solution. By counting the number of successful mutations p_s over n trials, Schwefel (1981) suggested a factor $c_d = 0.817$ in the following σ-update rule:

$$\sigma^{t+1} = \begin{cases} c_d\sigma^t, & \text{if } p_s < 1/5, \\ \frac{1}{c_d}\sigma^t, & \text{if } p_s > 1/5, \\ \sigma^t, & \text{if } p_s = 1/5. \end{cases} \tag{4.38}$$

In the presence of constraints, a mutated solution is considered better if it is feasible and is better than the parent in terms of the objective function value. Oyman, Deb, Beyer (1999) suggested a better constraint handling ES along the lines of constraint handling strategies used in genetic algorithms.

Multi-Membered ESs: $(\mu+\lambda)$-ES and (μ,λ)-ES

With the use of optimal mutation strength and the 1/5 success rule, the (1+1)-ES is better than the simple Monte Carlo method, where a solution in the neighborhood is created with an arbitrary user-defined probability distribution. However, the population approach used in multi-membered ESs makes them much better algorithms. There are two ways to introduce multiple members in an ES. We first discuss the 'plus' strategy, where μ (> 1) parent solutions are used to create λ offspring using mutation alone.

Multi-Membered ES: $(\mu+\lambda)$-ES

Step 1 Choose an initial population P of μ solutions $\mathbf{x}^{(i)}$, $i = 1, 2, \ldots, \mu$ and a mutation strength σ.

Step 2 Create λ mutated solutions:

$$\mathbf{y}^{(j)} = \mathbf{x}^{(i)} + \mathbf{N}(0, \sigma). \tag{4.39}$$

To create the j-th offspring, a parent i is chosen at random from μ parent solutions. $\mathbf{N}(0,\sigma)$ is defined as earlier.

Step 3 Modify the parent population P by combining all parent and offspring:

$$P = \left(\cup_{j=1}^{\lambda}\{\mathbf{y}^{(j)}\}\right) \cup \left(\cup_{i=1}^{\mu}\{\mathbf{x}^{(i)}\}\right).$$

From P, the best μ solutions are chosen and the rest are deleted.

Step 4 If a termination criterion is satisfied, stop. Otherwise, go to Step 2.

For many problems, $\lambda/\mu \approx 5$ is suggested. Figure 74 shows a schematic of this procedure. Since in the selection operation both parent and offspring populations are used, the $(\mu+\lambda)$-ES is an elitist algorithm. Although elite-preservation is desired, this procedure does not control the extent of elitism. In certain problems, where new successful mutations are difficult to achieve, the search in a $(\mu+\lambda)$-ES will stagnate. In order to avoid this problem, an alternate (μ,λ)-ES is suggested. The only difference between this and a $(\mu+\lambda)$-ES is in Step 3. Instead of combining the parent and offspring populations before selection, only the offspring population is now used. Thus, the modified population P before selection is $P = \cup_{j=1}^{\lambda}\{\mathbf{y}^{(j)}\}$. This requires that $\lambda \geq \mu$. In this way, the parent solutions are ignored and the success of the (μ,λ)-ES depends on the ability of the algorithm to find better offspring. Figure 75 shows this procedure. The (μ,λ)-ES is a non-elitist procedure. A strategy, where some limited copies of the better parent solutions are chosen and the rest are filled from

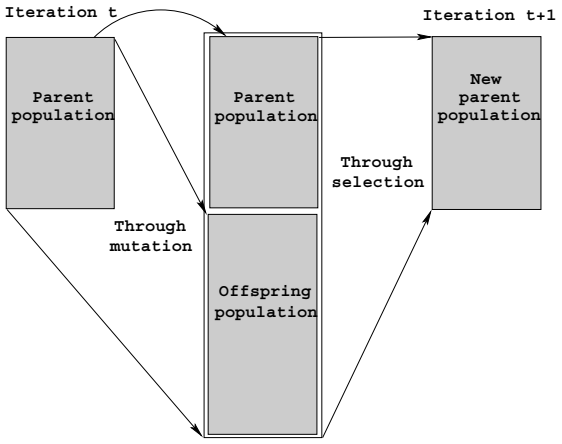


Figure 74 The procedure for a $(\mu + \lambda)$-ES.

the offspring population, would introduce elitism in a controlled manner and may turn out to be a good compromise between a $(\mu + \lambda)$-ES and a (μ, λ)-ES. One way to achieve this is to allow a life span, κ (≥ 1), for each parent (Schwefel and Bäck, 1998). A parent older than κ generations is not considered further in the selection process, thereby allowing new offspring solutions to enter into the new population. Using this notation, the two extreme strategies are (μ, λ)-ES for $\kappa = 1$ and $(\mu + \lambda)$-ES for $\kappa = \infty$.

4.3.2 Recombinative Evolution Strategies

In the recombinative ESs, a set of chosen parents are first recombined to find a new solution. Thereafter, the solution is mutated as before. Instead of choosing two parents or all parents for the recombination, a set of ρ (a value between 1 and μ inclusive) parents are chosen at random. A value of $\rho = 1$ means no recombination. Two types of recombination are mainly in use: (i) intermediate and (ii) discrete. In the intermediate

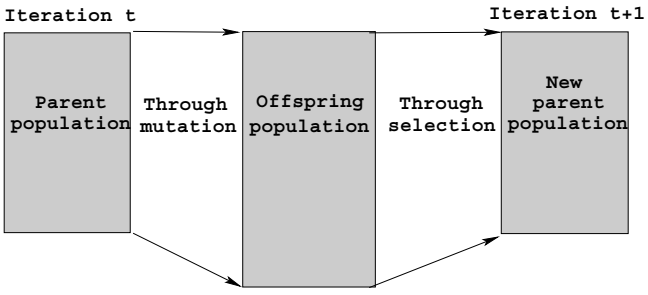


Figure 75 The procedure for a (μ, λ)-ES.

recombination operator, the average ρ chosen solution vectors are calculated as follows:

$$\mathbf{y} = \frac{1}{\rho}\sum_{i=1}^{\rho}\mathbf{x}^{(i)}. \tag{4.40}$$

It is clear that if all parents are used in each recombination, or $\rho = \mu$, the procedure has a tendency to cluster around the centroid of the current population. In problems, where a population brackets the true optimum, this may be a good strategy.

In the discrete recombination, each decision variable is chosen from one of the ρ parents at random. For example, a discretely recombined solution is shown derived from the following four solutions ($\rho = \mu$):

Parent 1:	$x_1^{(1)}$	$x_2^{(1)}$	$x_3^{(1)}$	$x_4^{(1)}$	$x_5^{(1)}$	$x_6^{(1)}$
Parent 2:	$x_1^{(2)}$	$x_2^{(2)}$	$x_3^{(2)}$	$x_4^{(2)}$	$x_5^{(2)}$	$x_6^{(2)}$
Parent 3:	$x_1^{(3)}$	$x_2^{(3)}$	$x_3^{(3)}$	$x_4^{(3)}$	$x_5^{(3)}$	$x_6^{(3)}$
Parent 4:	$x_1^{(4)}$	$x_2^{(4)}$	$x_3^{(4)}$	$x_4^{(4)}$	$x_5^{(4)}$	$x_6^{(4)}$
Recombinant:	$x_1^{(2)}$	$x_2^{(3)}$	$x_3^{(4)}$	$x_4^{(2)}$	$x_5^{(4)}$	$x_6^{(3)}$

This operator is similar to the uniform crossover procedure used in genetic algorithms. Different combinations of decision variables of the existing solutions can be obtained by this procedure. The step-by-step procedure of the entire algorithm is as follows:

Recombinant ES: $(\mu/\rho + \lambda)$-ES

Step 1 Choose an initial population P of μ solutions $\mathbf{x}^{(i)}$, $i = 1, 2, \ldots, \mu$ and a mutation strength vector $\boldsymbol{\sigma}$.

Step 2 Create λ mutated solutions, each using ρ parents randomly chosen from μ parents, as follows:

1. Calculate the recombinant solution $\mathbf{y}$ either by using an intermediate or a discrete recombination from ρ parents.
2. Mutate the recombinant solution:

$$\mathbf{y} = \mathbf{y} + \mathbf{N}(0, \sigma). \tag{4.41}$$

Step 3 Modify the parent population P by combining all parent and offspring:

$$P = \left(\cup_{j=1}^{\lambda}\{\mathbf{y}^{(j)}\}\right) \cup \left(\cup_{i=1}^{\mu}\{\mathbf{x}^{(i)}\}\right).$$

From P the best μ solutions are chosen and the rest are deleted.

Step 4 If a termination criterion is satisfied, stop. Otherwise, go to Step 2.

For the $(\mu/\rho, \lambda)$-ES, only the offspring population would be used to create the new population in Step 3 of the above algorithm.

4.3.3 Self-Adaptive Evolution Strategies

Self-adaptation is a phenomenon which makes evolutionary algorithms flexible and closer to natural evolution. Among the evolutionary methods, self-adaptation properties have been explored with evolution strategies (ESs) (Bäck, 1992, 1997; Beyer, 1995a, 1995b; Hansen and Ostermeier, 1996; Rechenberg, 1973; Sarvanan et al., 1995; Schwefel, 1987a) and evolutionary programming (EP) (Fogel et al., 1995), although there exist some studies of self-adaptation in genetic algorithms (GAs) with the mutation operator (Bäck, 1992). When applied to function optimization, there are a number of reasons why evolutionary algorithmists should pay attention to self-adaptation:

1. Knowledge of lower and upper bounds for the optimal solution may not be known a priori.
2. It may be necessary to know the optimal solution with arbitrary precision.
3. The objective function and the optimal solution may change with time.

There are three different ways where self-adaptation is introduced into ESs:

1. A hierarchically organized population-based meta-ES (Herdy, 1992).
2. Adaptation of the covariance matrix (CMA) determining the probability distribution for mutation (Hansen and Ostermeier, 1996).
3. Explicit use of self-adaptive control parameters (Rechenberg, 1973; Schwefel, 1987a).

The meta-ES method of self-adaptation uses two levels of ESs: the top level optimizes the strategy parameters (such as mutation strengths), a solution of which is used to optimize the true objective function in the lower-level ES. The second method (CMA) records the population history for a certain number of iterations to calculate covariance and variance information among the object variables. The subsequent search effort is influenced by these variance values. Here, we discuss the third type of self-adaptive ES, where the strategy parameters (variances σ_i^2 and covariances c_{ij}^2) are explicitly coded along with the decision variables and updated by using predefined update rules in each generation. There are basically three different implementations which are in use.

Isotropic Self-Adaptation

In this self-adaptive ES, a single mutation strength σ is used for all variables. In addition to n object variables, the strategy parameter σ is also used in a population member. The logarithmic update rules for the decision variables and the mutation strength are as follows:

$$\sigma^{(t+1)} = \sigma^{(t)} \exp(\tau_0 N(0,1)), \tag{4.42}$$

$$x_i^{(t+1)} = x_i^{(t)} + \sigma^{(t+1)} N_i(0,1), \tag{4.43}$$

where $N(0,1)$ and $N_i(0,1)$ are realizations of a one-dimensional normally distributed random variable with a mean of zero and a standard deviation of one. The parameter

τ_0 is the learning parameter which should be set as $\tau_0 \propto n^{-1/2}$, where n is the dimension of the variable vector (Schwefel, 1987a). Beyer (1995b) has shown that, for the sphere model, the optimal learning parameter for a (1,λ)-ES is $\tau_0 = c_{1,\lambda}/\sqrt{n}$, where $c_{1,\lambda}$ is the progress coefficient. For multi-membered ESs, an optimal learning parameter value is not determined. However, most studies use $c_{\mu,\lambda}$ or $c_{\mu/\mu,\lambda}$ as the constant of proportionality in the corresponding ES. The above update rule for σ requires an initial value, which may be chosen as $\sigma^{(0)} = (x_i^{(U)} - x_i^{(L)})/\sqrt{12}$, assuming a uniform distribution of solutions within the specified range of x_i values.

Besides the logarithmic update rules described above, there also exist symmetric and generalized two-point update rules (Schwefel, 1995).

Non-Isotropic Self-Adaptation

Here, a different mutation strength σ_i is used for each variable. Thus, this type of self-adaptive ES is capable of learning to adapt to problems where each variable has an unequal contribution to the objective function. In addition to n object variables, n other strategy parameters are included in the decision variable vector. The logarithmic update rules for decision variables and mutation strengths are as follows:

$$\sigma_i^{(t+1)} = \sigma_i^{(t)} \exp\left(\tau' N(0,1) + \tau N_i(0,1)\right), \tag{4.44}$$

$$x_i^{(t+1)} = x_i^{(t)} + \sigma_i^{(t+1)} N_i(0,1), \tag{4.45}$$

where $\tau' \propto (2n)^{-1/2}$ and $\tau \propto (2n^{1/2})^{-1/2}$. Due to lack of any theoretical results on this self-adaptive ES, the progress coefficients for the (μ,λ)-ESs or $(\mu/\mu,\lambda)$-ESs are normally used as the constant of proportionality for τ' and τ. Similar initial values for $\sigma_i^{(0)}$, as discussed for isotropic self-adaptive ESs, may be chosen.

Correlated Self-Adaptation

In correlated self-adaptation, in addition to n mutation strengths, $\binom{n}{2}$ covariances, at most, are included in each individual solution. Thus, there are a total of $\left(\binom{n}{2} + n\right)$ exogenous strategy parameters to be updated for each solution. Thus, this type of self-adaptive ES can adapt to problems where decision variables ($\mathbf{x}$) are correlated. Schwefel (1981) suggested that instead of using covariances, one can replace them with a rotation angle corresponding to each pair of coordinates. In a correlated problem (where decision variables are nonlinearly interacting with each other), the task is to find all pair-wise coordinate rotations and the spread of solutions in each rotated coordinate system so that the objective function is completely uncorrelated in the new coordinate system. Thus, the update rules are suggested for n decision variables, n mutation strengths σ_i and $\binom{n}{2}$ rotation angles α_j as follows:

$$\sigma_i^{(t+1)} = \sigma_i^{(t)} \exp\left(\tau' N(0,1) + \tau N_i(0,1)\right), \tag{4.46}$$

$$\alpha_j^{(t+1)} = \alpha_j^{(t)} + \beta_\alpha N_j(0,1), \tag{4.47}$$

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} + \mathbf{N}\left(\mathbf{0}, C(\boldsymbol{\sigma}^{(t+1)}, \boldsymbol{\alpha}^{(t+1)})\right), \tag{4.48}$$

where $\mathbf{N}\left(\mathbf{0}, C(\boldsymbol{\sigma}^{(t+1)}, \boldsymbol{\alpha}^{(t+1)})\right)$ is a realization of a normally distributed correlated mutation vector with a zero mean vector and a covariance matrix C. The parameter β_α is fixed as 0.0873 (or 5^o) (Schwefel, 1987a). Whenever an angle α_j goes outside the range $[-\pi, \pi]$, it is mapped inside the range, that is, if $|\alpha_j| > \pi$, $\alpha_j = \alpha_j - 2\pi \cdot \alpha_j/|\alpha_j|$. The parameters τ', τ and $\sigma_i^{(0)}$ are set as before. The rotation angles are initialized within zero and 180 degrees at random.

4.3.4 Connection Between Real-Parameter GAs and Self-Adaptive ESs

Without loss of generality, we will now discuss the similarity in the working of a real-parameter GA with an SBX operator and the isotropic self-adaptive ES. Under the latter, the difference (say Δ_i) between the offspring ($x_i^{(t+1)}$) and its parent ($x_i^{(t)}$) can be written from equations (4.42) and (4.43) as follows:

$$\Delta_i = \left[\sigma^{(t)} \exp(\tau_0 N(0,1))\right] N(0,1). \tag{4.49}$$

Thus, an instantiation of Δ_i is a normal distribution with zero mean and a variance which depends on $\sigma^{(t)}$, τ_0, and the instantiation of the log-normal distribution. For our argument, there are two aspects of this procedure:

1. For a particular realization of the log-normal distribution, the difference Δ_i is normally distributed with zero mean. That is, offspring closer to the parents are monotonically more likely to be created than offspring away from the parents.
2. The standard deviation of Δ_i is proportional to the mutation strength $\sigma^{(t)}$, which signifies, in some sense, the population diversity.

In Figure 76, we show the effect of a 50% reduction in $\sigma^{(t)}$ on the distribution of Δ_i under a self-adaptive ES.

Under the SBX operator, we write the term Δ_i by using equations (4.12) or (4.13) as follows:

$$\Delta_i = \frac{\delta_{p_i}}{2}(\beta_{q_i} - 1), \tag{4.50}$$

where δ_{p_i} is the absolute difference in the two parent solutions. There is a similarity between equations (4.49) and (4.50). The latter equation suggests that the distribution of Δ_i depends on the distribution of $(\beta_{q_i} - 1)$ for a particular pair of parents. The distribution of β_{q_i} has its mode at $\beta_{q_i} = 1$ (refer to equation (4.11)). Thus, the distribution of $(\beta_{q_i} - 1)$ will have its mode at zero. Although the distribution $(\beta_{q_i} - 1)$ is not a normal distribution, Figure 60 suggests that a small Δ_i has a higher probability to be created than a large Δ_i, and that this distribution is monotonic to the distance from a parent. The variance of this distribution depends on δ_{p_i}, which signifies the population diversity. Figure 77 shows the effect of a 50% reduction in δ_{p_i} on the distribution of Δ_i under the SBX operator. Comparing Figures 76 and 77 we observe that in self-adaptive ES, the diversity in the offspring population is controlled by the explicit mutation strength parameter, while in GAs with SBX operator it is controlled by the parent population diversity. Thus, there is a remarkable similarity in the way

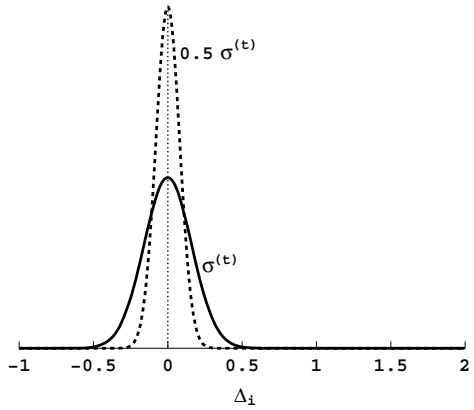


Figure 76 Effect of mutation strength $\sigma^{(t)}$ on the probability distribution of Δ_i under the self-adaptive ES. Reproduced from Deb and Beyer (2001) (© 2001 by the Massachusetts Institute of Technology).

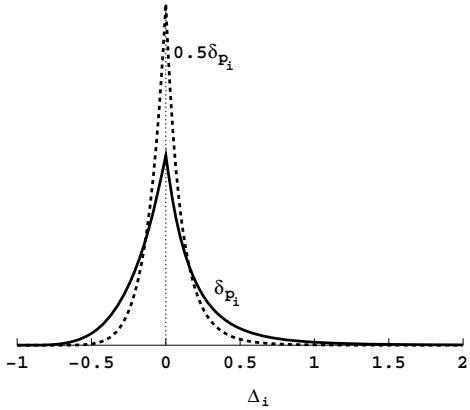


Figure 77 Effect of δ_{p_i} on the probability distribution of Δ_i under the SBX operator (with $\eta_c = 2$). Reproduced from Deb and Beyer (2001) (© 2001 by the Massachusetts Institute of Technology).

that offspring are assigned in both an isotropic self-adaptive ES and in a GA with an SBX operator. In both cases, the offspring closer to the parent solutions are assigned a larger probability to be created than solutions away from parents, and the variance of this probability distribution depends on the current population diversity.

Simulation Results

We consider the following 30-variable ellipsoidal model:

$$\text{ELP}: \begin{cases} \text{Minimize} & \sum_{i=1}^{30} 1.5^{i-1} x_i^2, \\ & -1 \le x_i \le 1 \quad \text{for } i = 1, 2, \ldots, 30. \end{cases} \tag{4.51}$$

Since each variable has an unequal contribution to the objective function, we use non-isotropic self-adaptive ESs. A tournament selection operator with size 3 and no mutation operator is used in GAs with the SBX operator (with $\eta_c = 1$). For the recombinative ES, we use the dominant crossover on object variables and intermediate recombination on strategy parameters, as suggested elsewhere (Bäck and Schwefel, 1993). Figure 78 shows the objective function value of the best solution in the population with a GA and with an (10/10,100)-ES. Since the vertical axis is drawn on a logarithmic scale, both approaches proceed towards the true optimum solution exponentially quickly. Since the slopes are more or less the same, the rate of convergence to the optimum is also similar in both approaches.

Figure 79 plots the population standard deviation in x_1, x_{15} and x_{30} variables in the population for the real-parameter GA with the SBX operator. Since they are scaled as 1.0, $1.5^{14} \approx 292$, and $1.5^{29} \approx 127\,834$, respectively, the 30-th variable is likely to

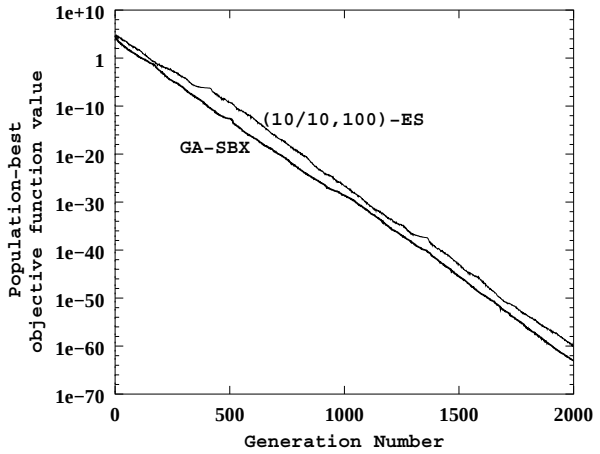


Figure 78 The best objective function value in the population is shown for two evolutionary algorithms – a real-parameter GA with SBX and a self-adaptive (10/10,100)-ES. Reproduced from Deb and Beyer (2001) (© 2001 by the Massachusetts Institute of Technology).

have a smaller population variance than the first variable. The figure shows this fact clearly. Since, the ideal mutation strengths for these variables are also likely to be inversely proportionate to 1.5^{i-1}, we find a similar ordering with the non-isotropic self-adaptive ES as well (Figure 80). Thus, there is a remarkable similarity in how both real-parameter GAs with SBX operator and self-adaptive ESs work. In the former case, the population diversity becomes adapted by the fitness landscape, which in turn helps the SBX operator to create x_i value of offspring proportionately. In the latter case, the population diversity becomes controlled by independent mutation strengths, which become adapted based on the fitness landscape. Comparative simulations on other problems can also be found in the original study (Deb and Beyer, 2001).

4.4 Evolutionary Programming (EP)

Evolutionary programming (EP) is a mutation-based evolutionary algorithm applied to discrete search spaces. David Fogel (Fogel, 1988) extended the initial work of his father Larry Fogel (Fogel, 1962) for applications involving real-parameter optimization problems. Real-parameter EP is similar in principle to evolution strategy (ES), in that normally distributed mutations are performed in both algorithms. Both algorithms encode mutation strength (or variance of the normal distribution) for each decision variable and a self-adapting rule is used to update the mutation strengths. Several variants of EP have been suggested (Fogel, 1992). Here, we will discuss the so-called *standard* EP, for tackling continuous parameter optimization problems.

EP begins its search with a set of solutions initialized randomly in a given bounded space. Thereafter, EP is allowed to search anywhere in the real space, similar to the real-parameter GAs. Each solution is evaluated to calculate its objective function value

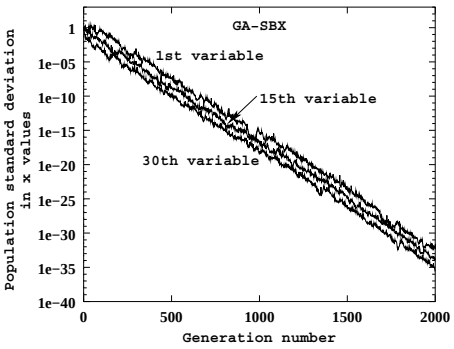


Figure 79 Population standard deviations in the variables x_1, x_{15} and x_{30} shown with a real-parameter GA with the SBX operator for the function ELP. Reproduced from Deb and Beyer (2001) (© 2001 by the Massachusetts Institute of Technology).

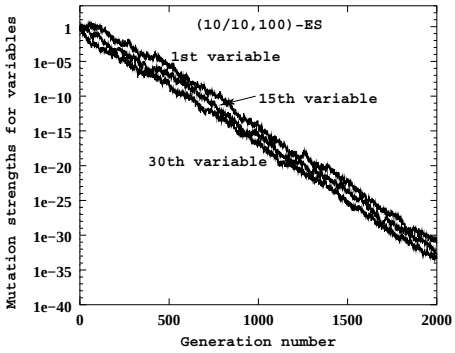


Figure 80 Mutation strengths for the variables x_1, x_{15} and x_{30} shown with a self-adaptive (10/10,100)-ES for the function ELP. Reproduced from Deb and Beyer (2001) (© 2001 by the Massachusetts Institute of Technology).

$f(\mathbf{x})$. Then, a fitness value $F(\mathbf{x})$ is calculated by some user-defined transformation of the objective function. However, the fitness function can also be the same as the objective function.

After each solution is assigned a fitness, it is mutated by using a zero-mean normally distributed probability distribution and a variance dependent on the fitness function, as follows:

$$x_i^{t+1} = x_i^t + \left(\sqrt{\beta_i F(\mathbf{x}) + \gamma_i}\right) N_i(0,1), \tag{4.52}$$

where the parameters β_i and γ_i must be tuned for a particular problem, such that the term inside the square-root function is positive. If the fitness function $F(\mathbf{x})$ is normalized between zero and one (with $F(\mathbf{x}) = 0$ corresponding to the global minimum), the parameters $\gamma_i = 0$ can be used. In such an event, as the solutions approach the global minimum, the mutation strength becomes smaller and smaller. However, for other fitness functions, a proper γ_i must be chosen. To avoid this user-dependent parameter setting, Fogel (1992) suggested a meta-EP, where the mutation strength for each decision variable is also evolved, similar to the self-adaptive evolution strategies. In a meta-EP, problem variables constitute decision variables $\mathbf{x}$ and mutation strengths $\boldsymbol{\sigma}$. Each is mutated as follows:

$$x_i^{t+1} = x_i^t + \sigma_i^t N_i(0,1), \tag{4.53}$$

$$(\sigma_i^{t+1})^2 = (\sigma_i^t)^2 + \zeta \sigma_i^t N_i(0,1), \tag{4.54}$$

where ζ is a user-defined exogenous parameter, which must be chosen to make sure that the mutation strength remains positive. It is important to mention here that there also exists a Rmeta-EP, where in addition to the mutation strengths, correlation

coefficients are used as decision variables, as in the case of correlated evolution strategies (Fogel et al., 1992).

After the mutation operation, both parent and offspring populations are combined together and the best N solutions are probabilistically selected for the next generation. The selection operator is as follows. For each solution $x^{(i)}$ in the combined population, a set of η members are chosen at random from the combined population. Thereafter, the fitness of $x^{(i)}$ is compared with that of each member of the chosen set and a *score* equal to the number of solutions having the worse fitness is counted. Thereafter, all 2N population members are sorted according to the descending order of their scores and the best N solutions having the better score are selected. It is clear that if $\eta = 2N$ is chosen, the procedure is identical to the deterministic selection of the best N solutions. For $\eta < 2N$, the procedure is probabilistic. Since both parent and offspring populations are combined before selecting N solutions, the selection procedure preserves elitism. Fogel (1992) proved the above algorithm for convergence. Bäck (1996) showed that for the sphere model having a large number of variables, the implicit mutation strength used in the EP is larger than the optimal mutation strength, thereby causing the EP to perform worse than an optimal algorithm. Fogel's correction to equation (4.52) with:

$$\beta_i = \frac{1}{n^2}, \tag{4.55}$$

for $n > 2$ helps make the implicit mutation strength of the same order as the optimal mutation strength. An EP with the above fix-up is expected to perform as good as an evolution strategy with an optimal mutation strength.

4.5 Genetic Programming (GP)

Genetic programming (GP) is a genetic algorithm applied to computer programs to evolve efficient and meaningful programs for solving a task (Koza, 1992; Koza, 1994; Banzhaf et al., 1998). Instead of decision variables, a program (usually a C, C++, or Lisp program) represents a procedure to solve a task. For example, let us consider solving the following differential equation using a GP:

$$\left.\begin{array}{rl} \text{Solve} & \frac{dy}{dx} = 2x, \\ \text{with} & y(x=0) = 2. \end{array}\right\} \tag{4.56}$$

The objective in this problem is to find a solution $y(x)$ (y as a function of x) which will satisfy the above problem description. Here, a solution in a GP method is nothing but an arbitrary function of x. Some such solutions are $y(x) = 2x^2 - 2x + 1$, $y(x) = \exp(x)+2$, or $y(x) = 2x+1$. It is the task of a GP technique to find the function which is the correct solution to the above problem through genetic operations.

In order to create the initial population, a *terminal* set T and a *function* set F must be pre-specified. A terminal set consists of constants and variables, whereas a function set consists of operators and basic functions. For example, for the above problem $T = \{x, 1, 2, -1, -2, \ldots\}$ and $F = \{\div, \times, +, -, \times, \exp, \sin, \cos, \log, \ldots\}$ can be used. By

the use of two such sets, valid syntactically correct programs can be developed. Two such programs are shown in Figure 81. Like a GA, a GP process starts with a set of randomly created syntactically correct programs. Each program (or solution) is evaluated by testing the program in a number of S instances in a supervised manner. By comparing the outcome of the program on each of these instances with the actual outcome, a fitness is assigned. Usually, the number of *matched* instances (say I_i) are counted and a fitness equal to I_i/S is assigned to the i-th individual. By maximizing this artificial fitness metric, one would hope to find a solution with a maximum of S matched instances. In the above problem, the fitness of a solution can be calculated in the following manner. First, a set of solutions can be chosen. We illustrate the fitness assignment procedure by choosing a set of 11 instances of x, as shown in Table 8. The true value of the right side of the differential equation can be computed for each solution (column 3 in the table). Thereafter, for each solution, the left side of the differential equation can be computed numerically. In column 4 of the table, we compute the exact value of the left side of the differential equation for the solution $y^{(1)}(x) = 2x^2 - 2x + 1$. Now, an error measure between these two columns (3 and 4) can be calculated as the fitness of the solution. Obviously, such a fitness metric must be minimized in order to get the exact solution of the differential equation. By computing the absolute differences between columns 3 and 4 and summing, we determine the fitness of the solution as 11.0. Since $y^{(1)}(x = 0) = 1$ does not match with the given initial condition, $y(x = 0) = 2$, we add a penalty to the above fitness proportional to this difference and calculate the overall fitness of the solution. With a penalty parameter of one, the overall fitness becomes $11.0 + 1.0$, or 12.0. This way, even if the differential equation is satisfied by a function but the initial condition is not satisfied, then the overall fitness will be non-zero. Only when both are satisfied, will the overall fitness take the lowest possible value.

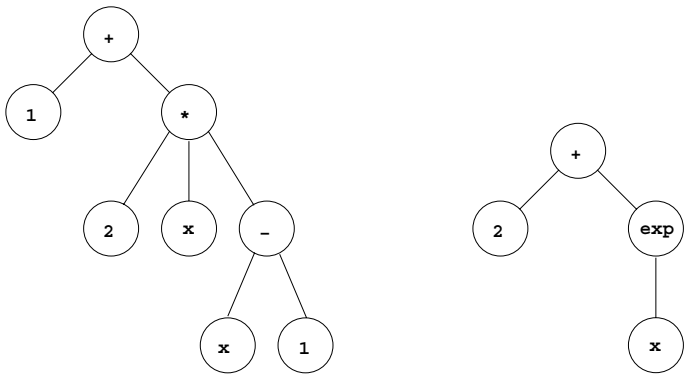


Figure 81 Two GP solutions.

Table 8 A set of 11 instances and evaluation of a solution $y^{(1)}(x) = 2x^2 - 2x + 1$.

Instances	x	$2x$	$dy^{(1)}/dx$
1	0.0	0.0	−2.0
2	0.1	0.2	−1.6
3	0.2	0.4	−1.2
4	0.3	0.6	−0.8
5	0.4	0.8	−0.4
6	0.5	1.0	0.0
7	0.6	1.2	0.4
8	0.7	1.4	0.8
9	0.8	1.6	1.2
10	0.9	1.8	1.6
11	1.0	2.0	2.0

The crossover and mutation operators are similar to that in GAs. We illustrate the working of a typical GP crossover operator on two parent solutions in Figure 82. A sub-program to be exchanged between two parents is first chosen at random in each parent. Thereafter, the crossover operator is completed by exchanging the two sub-programs, as shown in Figure 82.

The mutation operator can be applied to a terminal or to a function. If applied to a terminal, the current value is exchanged with another value from the terminal set, as shown in Figure 83. Similarly, if the mutation operation is to be applied to a function (a node), a different function is chosen from the function set. Here, care must be taken to keep the syntax of the chosen function valid.

Like GAs, GP operators are usually run for a pre-specified number of generations, or until a solution with a best fitness value is found. In addition to the above simple GP operators, there exist a number of other advanced operators, such as automatically defined functions, meaningful crossover, and various others, which have been used to solve a wide variety of problems (Koza, 1994).

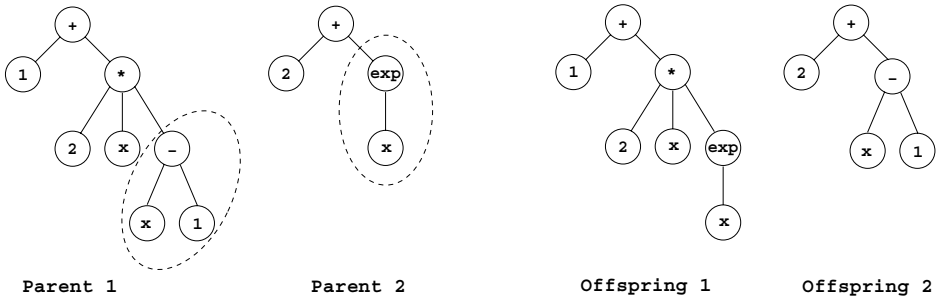


Figure 82 The GP crossover operator.

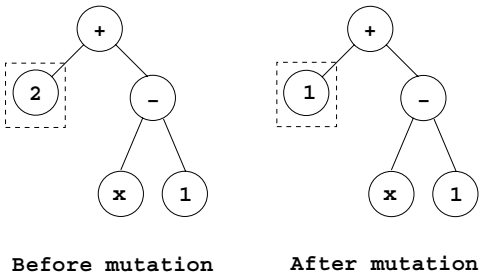


Figure 83 The GP mutation operator.

4.6 Multi-Modal Function Optimization

As the name suggests, multi-modal functions have multiple optimum solutions, of which many are local optimal solutions. Multi-modality in a search and optimization problem usually causes difficulty to any optimization algorithm in terms of finding the global optimum solutions. This is because in these problems there exist many attractors in which an algorithm can become directed to. Finding the global optimum solution in such a problem becomes a challenge to any optimization algorithm.

However, in this section, we will discuss an optimization task which is much more difficult than just finding the global optimum solution in a multi-modal optimization problem. The objective in a multi-modal function optimization problem is to find multiple optimal solutions having either equal or unequal objective function values. This means that in addition to finding the global optimum solution(s), we are interested in finding a number of other local optimum solutions. It may apparently look puzzling as to why one would be interested in finding local optimum solutions. However, the knowledge of multiple local and global optimal solutions in the search space is particularly useful in obtaining an insight into the function landscape. Such information is also useful to design engineers and practitioners for choosing an alternative optimal solution, as and when required. In practice, because of the changing nature of a problem, new constraints may make a previous optimum solution infeasible to implement. The knowledge of alternate optimum solutions (whether global or local optimum) allows a user to conveniently shift from one solution to another.

Most classical search and optimization methods begin with a single solution and modify the solution iteratively and finally recommend only one solution as the obtained optimum solution (refer to Figure 39). Since our goal is to find multiple optimal solutions, there are at least two difficulties with the classical methods:

1. A classical optimization method has to be applied a number of times, each time starting with a different initial guess solution to hopefully find multiple optimum solutions.
2. Different applications of a classical method from different initial guess solutions do not guarantee finding different optimum solutions. This scenario is particularly true if the chosen initial solutions lie in the basin of attraction of an identical optimum.

Since EAs work with a population of solutions instead of one solution, some change in the basic EA, which is described in the previous section, can be made so that the final EA population consists of multiple optimal solutions. This will allow an EA to find multiple optimal solutions simultaneously in one single simulation run. The basic framework for the working of an EA suggests such a plausibility and makes EAs unique algorithms in terms of finding multiple optimal solutions. In this section, we will discuss different plausible modifications to a basic genetic algorithm for finding and maintaining multiple optimal solutions in multi-modal optimization problems. These ideas are then borrowed in subsequent chapters to find multiple Pareto-optimal solutions in handling multi-objective optimization problems.

4.6.1 Diversity Through Mutation

Finding solutions closer to different optimal solutions in a population and maintaining them over many generations are two different matters. Early on in an EA simulation, solutions are distributed all over the search space. Because of parallel schema processing in multiple schema partitions, solutions closer to multiple optimal solutions get emphasized in early generations. Eventually, when the population contains clusters of good solutions near many optima, competitions among clusters of different optima begins. This continues until the complete population converges to a single optimum. Therefore, although it is possible to discover multiple solutions during early generations, it may not be possible to maintain them automatically in a GA. In order to maintain multiple optimal solutions, an explicit diversity-preserving operator must be used.

The mutation operator is often used as a diversity-preserving operator in an EA. Although, along with selection and crossover operators, it can help find different optimal solutions, it can help little in preserving useful solutions over a large number of generations. The mutation operator has a constructive as well as a destructive effect. As it can create a better solution by perturbing a solution, it can also destroy a good solution. Since we are interested in accepting the constructive effect and since it is computationally expensive to check the worth of every possible mutation for its outcome, the mutation is usually used with a small probability in a GA. This makes the mutation operator insufficient to use as a sole diversity-preserving operator for maintaining multiple optimal solutions.

4.6.2 Preselection

Cavicchio (1970) was the first to introduce an explicit mechanism of maintaining diversity in a GA. Replacing an individual with a *like* individual is the main concept in a preselection operator. Cavicchio introduced preselection in a simple way. When an offspring is created from two parent solutions, the offspring is compared with respect to both parents. If the offspring has a better fitness than the worst parent solution, it replaces that parent. Since an offspring is likely to be similar (in genotypic space

and in phenotypic space) to both parents, replacement of a parent with an offspring allows many different solutions to co-exist in the population. This operation is also carried out in all pairs of the mating parents. Thus, the resulting successful offspring replace one of their parents, thereby allowing multiple important solutions to co-exist in the population. Although Cavicchio's study did not bring out the effect of introducing preselection because of the use of many other operators, Mahfoud (1992) later suggested a number of variations, where the preselection method is used to solve a couple of multi-modal optimization problems.

4.6.3 Crowding Model

DeJong (1975) in his doctoral dissertation used a crowding model to introduce diversity among solutions in a GA population. Although his specific implementation turns out to be weak for multi-modal optimization, he laid the foundation for many popular diversity preservation techniques which are in use to date.

As the name suggests, in a crowding model, crowding of solutions anywhere in the search space is discouraged, thereby providing the diversity needed to maintain multiple optimal solutions. In his crowding model, DeJong used an overlapping population concept and a crowding strategy. In his GA, only a proportion G (called the *generation gap*) of the population is permitted to reproduce in each generation. Furthermore, when an offspring is to be introduced in the overlapping population, CF solutions (called the *crowding factor*) from the population are chosen at random. The offspring is compared with these CF solutions and the solution which is most *similar* to the offspring is replaced. In his study, DeJong used $G = 0.1$ and $CF = 2$ or 3. Since a similar string is replaced by the offspring, diversity among different strings can be preserved by this process. Similarity among two strings can be defined in two spaces – phenotypic (the decision variable space) and genotypic space (the Hamming space). DeJong's crowding model has been subsequently used in machine learning applications (Goldberg, 1983) and has also been analyzed (Mahfoud, 1995). The main contribution of DeJong's study was the suggestion of replacing one solution by a similar solution in maintaining multiple optimum solutions in an evolving population.

4.6.4 Sharing Function Model

Goldberg and Richardson (1987) suggested another revolutionary concept, where instead of replacing a solution by a similar solution, the focus was more on degrading the fitness of similar solutions. The investigators viewed the population as a battlefield for multiple optimal solutions, with each trying to survive with the largest possible occupants. Since most subsequent GA studies have used this model in solving multi-modal optimization problems, we will describe it here in somewhat greater detail.

Let us imagine that we are interested in finding q optimal solutions in a multi-modal optimization problem. The only resource available to us is a finite population with N slots. It is assumed here that $q \ll N$, so that each optimum can work with an adequate

subpopulation (niche) of solutions. Since the population will need representative solutions of all q optima, somehow the available resource of N population slots must have to be *shared* among all representative solutions. This means that if for each optimum i, there is an expected occupancy m_i in the population, $\sum_{i=1}^{q} m_i = N$. If, in any generation, an optimum i has more than this expected number of representative solutions, this has to come at the expense of another optimum. Thus, the fitness of each of these representative solutions must be degraded so that in an overall competition the selection operator does not choose all representative solutions. On the other hand, solutions of an under-represented optimum must be emphasized by the selection operator. Arguing from a two-armed bandit game-playing scenario, the investigators showed that if such a sharing concept is introduced, the arm with a payoff f gets populated by a number of solutions (m) proportionate to f, or $m \propto f$. For q armed bandits (or q optimal solutions), this means that the ratio of payoff f (optimal objective value) and m (number of subpopulation slots) for each optimum would be identical, or:

$$\frac{f_1}{m_1} = \frac{f_2}{m_2} = \cdots = \frac{f_q}{m_q}. \tag{4.57}$$

This suggests an interesting phenomenon. If a hypothetical fitness function f' (called the 'shared fitness function') is defined as follows:

$$f_i' = \frac{f_i}{m_i}, \tag{4.58}$$

then all optima (whether local or global) would have an identical shared fitness value. If a proportionate selection is now applied with the shared fitness values, all optima will get the same expected number of copies, thereby emphasizing each optimum equally. Since this procedure will be performed in each generation, the maintenance of multiple optimal solutions is also possible.

Although the above principle seems like a reasonable scheme for maintaining multiple optimal solutions in a finite population from generation after generation, there is a practical problem. The precise identification of solutions belonging to each true optimum demands a lot of problem knowledge and in most problems is not available. Realizing this, Goldberg and Richardson (1987) suggested an adaptive strategy, where a *sharing function* is used to obtain an estimate of the number of solutions belonging to each optimum. Although a general class of sharing functions is suggested, they used the following function in their simulation studies:

$$Sh(d) = \begin{cases} 1 - \left(\frac{d}{\sigma_{share}}\right)^{\alpha}, & \text{if } d \leq \sigma_{share}; \\ 0, & \text{otherwise.} \end{cases} \tag{4.59}$$

The parameter d is the distance between any two solutions in the population. The above function takes a value in $[0, 1]$, depending on the values of d and σ_{share}. If d is zero (meaning that two solutions are identical or their distance is zero), $Sh(d) = 1$. This means that a solution has full sharing effect on itself. On the other hand, if $d \geq \sigma_{share}$ (meaning that two solutions are at least a distance of σ_{share} away from

each other), $Sh(d) = 0$. This means that two solutions which are a distance of σ_{share} away from each other do not have any sharing effect on each other. Any other distance d between two solutions will have a partial effect on each. If $\alpha = 1$ is used, the effect linearly reduces from one to zero. Thus, it is clear that in a population, a solution may not get any sharing effect from some solutions, may get partial sharing effect from few solutions, and will get a full effect from itself. Figure 84 shows the sharing function $Sh(d)$ as it varies with the normalized distance d/σ_{share} for different values of α. If these sharing function values are calculated with respect to all population members (including itself), are added and a niche count nc_i is calculated for the i-th solution, as follows:

$$nc_i = \sum_{j=1}^{N} Sh(d_{ij}), \tag{4.60}$$

then the niche count provides an estimate of the extent of crowding near a solution. Here, d_{ij} is the distance between the i-th and j-th solutions. It is important to note that nc_i is always greater than or equal to one. This is because the right side includes the term $Sh(d_{ii}) = Sh(0.0) = 1$. The final task is to calculate the shared fitness value as $f'_i = f_i/nc_i$, as suggested above. Since all over-represented optima will have a larger nc_i value, the fitness of all representative solutions of these optima would be degraded by a large amount. All under-represented optima will have a smaller nc_i value and the degradation of the fitness value would not be large, thereby emphasizing the under-represented optima. The following describes the step-by-step procedure for calculating the shared fitness value of a solution i in a population.

Shared Fitness Calculation

Step 1 Calculate the sharing function value $Sh(d_{ij})$ with all population members by using equation (4.59).

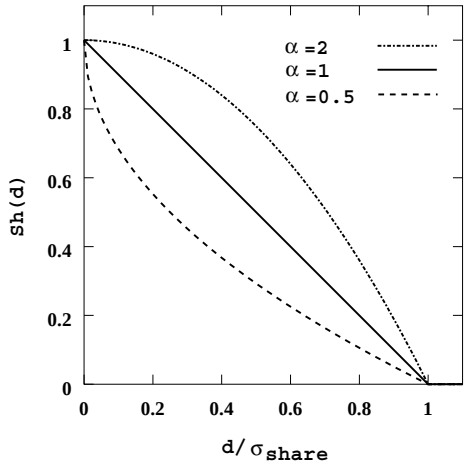


Figure 84 The sharing function plotted with three α values.

Step 2 Add all sharing function values to calculate the niche count nc_i by using equation (4.60).

Step 3 Calculate the shared fitness value as $f'_i = f_i/nc_i$.

After the shared fitness is calculated for all population members, a proportionate selection method is used to find the mating pool. The crossover and mutation operators are applied to the whole population as usual.

Careful thinking will reveal that the shared fitness computation of a population member i is $O(N)$, since this involves finding the distance d_{ij} with every population member. Although some bookkeeping can be used, the computation for calculating the shared fitness value of all population members is $O(N^2)$. However, it is important to highlight that in most real-world problems the calculation of the distance between vectors of decision variables or strings could be negligible compared to expensive objective function computations. Hence, the above computational complexity may not be critical. However, there exists another factor, which has been the source of some criticisms of the sharing function method, a matter we will discuss later (see page 154).

An Example Problem

In order to illustrate the working of the sharing function approach, let us take a bimodal function:

$$\left.\begin{array}{ll} \text{Maximize} & f(x) = |\sin(\pi x)|, \\ & 0 \leq x \leq 2. \end{array}\right\} \tag{4.61}$$

Let us also consider a population of six solutions, as shown in Table 9. A six-bit coding is used and decoded value of each string is mapped in $[0, 2]$ to calculate the corresponding decision variable value. These six solutions are also shown as circles on a function plot in Figure 85. This figure shows that of the two maxima, the second maximum ($x^* = 1.5$) is populated by more solutions than the first maximum ($x^* = 0.5$). If the sharing approach is not used, this population is most likely to lead a GA to converge to the second maximum alone, thereby not finding the first maximum.

Table 9 Six solutions and corresponding shared fitness values.

Sol. i	String	Decoded value	$x^{(i)}$	f_i	nc_i	f'_i
1	110100	52	1.651	0.890	2.856	0.312
2	101100	44	1.397	0.948	3.160	0.300
3	011101	29	0.921	0.246	1.048	0.235
4	001011	11	0.349	0.890	1.000	0.890
5	110000	48	1.524	0.997	3.364	0.296
6	101110	46	1.460	0.992	3.364	0.295

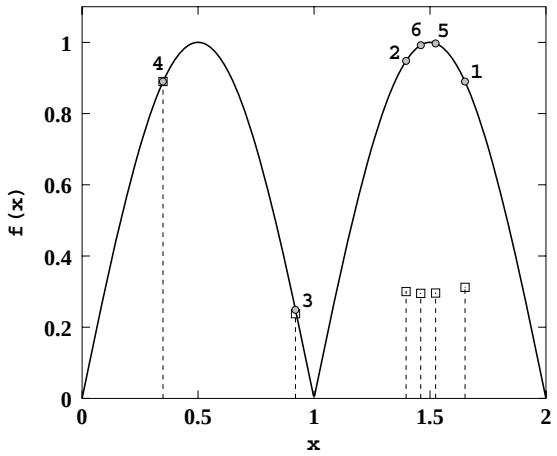


Figure 85 Six solutions and their shared fitness values (shown by dashed lines) on the bimodal function.

We will now illustrate the sharing function approach and calculate the shared fitness values of each population member.

To illustrate this, we use phenotypic sharing and a $\sigma_{\text{share}} = 0.5$. In the next subsection, we will show that with two optima and a range of 2.0 for the decision variable, the suggested σ_{share} is indeed 0.5. We also choose $\alpha = 1$. We use the above procedure to calculate the shared fitness of the first solution only.

Step 1 From the first solution, the distances from all population members are as follows:

$$\begin{aligned} &d_{11} = 0, \quad && d_{12} = 0.254, \quad d_{13} = 0.731, \quad d_{14} = 1.302, \\ &d_{15} = 0.127, \quad && d_{16} = 0.191. \end{aligned}$$

The corresponding sharing function values are calculated by using equation (4.59):

$$\begin{aligned} &Sh(d_{11}) = 1, \quad && Sh(d_{12}) = 0.492, \quad Sh(d_{13}) = 0, \quad Sh(d_{14}) = 0, \\ &Sh(d_{15}) = 0.746, \quad && Sh(d_{16}) = 0.618. \end{aligned}$$

Note that since solutions 3 and 4 are more than a 0.5 unit away from solution 1, their sharing effect is zero.

Step 2 The niche count of the first solution is simply the addition of the above sharing function values, or:

$$nc_1 = 1 + 0.492 + 0 + 0 + 0.746 + 0.618 = 2.856.$$

Table 10 Computation of niche counts for all six solutions.

	Sharing function values						nc_i
	1	2	3	4	5	6	
1	1	0.492	0	0	0.746	0.618	2.856
2	0.492	1	0.048	0	0.746	0.874	3.160
3	0	0.048	1	0	0	0	1.048
4	0	0	0	1	0	0	1.000
5	0.746	0.746	0	0	1	0.872	3.364
6	0.618	0.874	0	0	0.872	1	3.364

Step 3 The shared fitness value of the first solution is:

$$f'_1 = f(x^{(1)})/nc_1 = 0.890/2.856 = 0.312.$$

Table 10 shows the niche count calculations of all six solutions. The right-most column of this table shows the niche count values, which are copied in the sixth column of Table 9. The corresponding shared fitness values are shown in the seventh column of Table 9. Figure 85 also shows these shared fitness values with dashed lines. Since, solution 4 is near-optimal to $x^* = 0.5$ and this is the only solution representing this optimum, its shared fitness value is higher than any other solution in the population. On the other hand, since there are four representatives (solutions 1, 2, 5 and 6) of the second maximum, their shared fitness values are somewhat less. Since the selection operator will be performed with these shared fitness values, more copies of solution 4 are expected in the mating pool. By calculating the average shared fitness of the population, it is found that $f'_4/f'_{avg} = 2.29$, whereas the average of f'_i/f'_{avg} of the four solutions near the second maximum is 0.77. Thus, by the action of a proportionate selection, it is expected that solution 4 would have two or more copies and the four solutions near the second maximum may reduce to three solutions ($4 \times 0.77 = 3.08$). These calculations show how the sharing function approach can restore a balance between the number of solutions in each optimum.

Parameter Setting

The sharing function $Sh(d)$ defined in equation (4.59) introduced two user-defined parameters – α and σ_{share}. Although α does not have too much effect on the performance of the sharing function method (Deb, 1989), the second parameter σ_{share} must be set right in order to define the *niche size* of an optimum. Ideally, the latter is directly related to the basin of attraction of the optimum. Since this information is not available in an arbitrary problem, Deb and Goldberg (1989) suggested different ways to set this parameter. We will first present the suggestions for phenotypic sharing and then discuss the same for genotypic sharing. In most applications, an $\alpha = 1$ or 2 is used.

For phenotypic sharing, the Euclidean distance between two decision variable vectors $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(j)}$ can be calculated as d_{ij}:

$$d_{ij} = \sqrt{\sum_{k=1}^{n} (x_k^{(i)} - x_k^{(j)})^2}. \tag{4.62}$$

Imagining that the n-dimensional hypersphere of radius r must be divided among q optima equally (each small hypersphere having a radius σ_{share}), this reduces to the following equation:

$$\begin{aligned} q(c\sigma_{share}^n) &= cr^n, \quad \text{or} \\ \sigma_{share} &= \frac{r}{\sqrt[n]{q}}. \end{aligned}$$

Here, $r = \frac{1}{2}\sqrt{\sum_{k=1}^{n} (x_k^{(U)} - x_k^{(L)})^2}$. Thus, as estimate of σ_{share} for introducing q equispaced niches in the search space is:

$$\sigma_{share} = \frac{\sqrt{\sum_{k=1}^{n} (x_k^{(U)} - x_k^{(L)})^2}}{2\sqrt[n]{q}}. \tag{4.63}$$

For $n = 1$ (single variable problems), the above equation reduces to $\sigma_{share} = (x_k^{(U)} - x_k^{(L)})/(2q)$. If normalized distance values $d_{ij} = \sqrt{\sum_{k=1}^{n} [(x_k^{(i)} - x_k^{(j)})/(x_k^{(U)} - x_k^{(L)})]^2}$ are used, the following normalized sharing parameter value can be employed:

$$\sigma_{share} = \frac{0.5}{\sqrt[n]{q}}. \tag{4.64}$$

For genotypic sharing, the Hamming distance between two ℓ-bit binary strings $s^{(i)}$ and $s^{(j)}$ is used as d_{ij}. The Hamming distance is defined as the number of positions where bits are different in two strings. This distance value only takes an integer value in the range $[0, \ell]$. Counting the number of strings having different bit differences with a given string in the entire search space, Deb and Goldberg (1989) calculated the number of allowed bit differences K for which all strings having K or less bit differences occupy at least the $1/q$-th portion of the entire search space, or:

$$\sum_{i=0}^{K} \binom{\ell}{i} = \frac{2^\ell}{q}. \tag{4.65}$$

Since there may not exist an integer K satisfying the above equation, we calculate the minimum value of K for which the left side becomes more than the right side. This minimum value of K is assigned as σ_{share}.

Simulation Results

In order to demonstrate the working of the sharing function method, we show one GA simulation run on the following five-peaked function:

$$\left.\begin{array}{ll} \text{Maximize} & f(x) = 2^{-2((x-0.1)/0.8)^2} \sin^6(5\pi x), \\ & 0 \leq x \leq 1. \end{array}\right\} \tag{4.66}$$

We use a population size of 100, the proportionate selection method, and a single-point crossover operator with a probability of 0.9. No mutation is used. The GA is run until 200 generations have taken place, in order to demonstrate that subpopulations in all five optima are possible to maintain even for a large number of generations. Figure 86 shows that without the sharing function approach, a GA converges to the best maximum. However, Figure 87 shows that all five maxima are found in a single simulation run of a GA with the sharing function approach with $\sigma_{\text{share}} = 0.1$ (obtained using equation (4.63) with $n = 1$, $q = 5$, $x^{(U)} = 1$ and $x^{(L)} = 0$).

Difficulties of the Sharing Function Approach

There are two difficulties with the sharing function approach. First, there is a need for choosing a σ_{share}. We have already suggested estimates for choosing an appropriate σ_{share} earlier. Both types of sharing require a user to fix a value of the number of optima q to calculate an estimate of an adequate σ_{share}. If the q chosen to compute σ_{share} is larger than the actual number of optima in the search space, the sharing function approach tends to form more niches than the function can allow. This may lead to finding a number of suboptimal solutions in addition to the optimal solutions. On the other hand, if the chosen q underestimates the actual number of optima in the search space, not all optima may be found by the sharing function approach. In

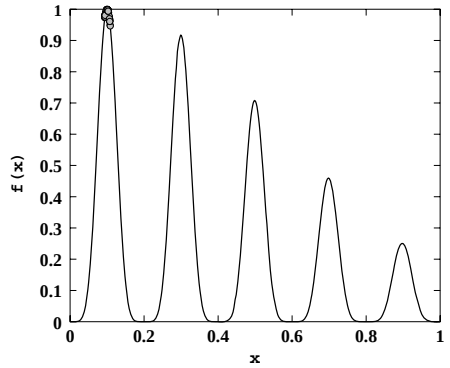


Figure 86 The population obtained without the sharing function approach.

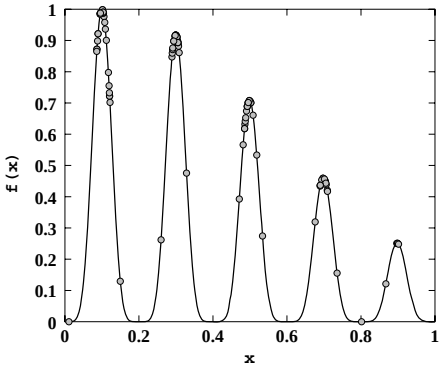


Figure 87 The population obtained with the sharing function approach. Reproduced from Deb (1997b) (© 1997 IOP Publishing Ltd and Oxford University Press).

most applications where an exact number of optima is not known a priori, a $q = 5$ to 10 may be tried. It is also important to note that if q niches are needed to be found, a GA needs an adequate population size to support all q subpopulations. We suggest using $N = \kappa q$, where κ is a constant and must be calculated as the minimum population size needed to find one optimum independently.

The computed niche count for each solution provides an idea of the number of solutions around an optimum and can be used to resize σ_{share} at every generation. Such a dynamic update of σ_{share} may provide a better performance of an EA than a fixed value of σ_{share}. More research in this direction must be pursued to evaluate the efficacy of such techniques. We will discuss one such dynamic update procedure in Section 5.8.6.

Secondly, there is an inherent scaling problem that shared fitness values can introduce. Consider the population shown using circles on the two-optimum function in Figure 88. The shared fitness values (marked by dashed lines in the figure) suggest that a sub-optimal solution (solution A) has the best shared fitness value. This is because the solution has a fitness value which is not too much worse than the optimal solutions, but is not crowded by many solutions. Since the proportionate selection will emphasize this solution the most (even more than the near-optimal solutions), the population may contain some sub-optimal solutions. Figure 87 also shows the existence of some such solutions, even after 200 generations. However, many of the non-optimal solutions shown in Figure 87 are also the outcome of crossovers between solutions from two different optima, a matter we will discuss below in Section 4.6.7.

Real-Parameter GAs for Multi-Modal Optimization

In the above simulations, binary-coded GAs are used. Since the implementation of the sharing function approach affects only the selection operator, it can also be used

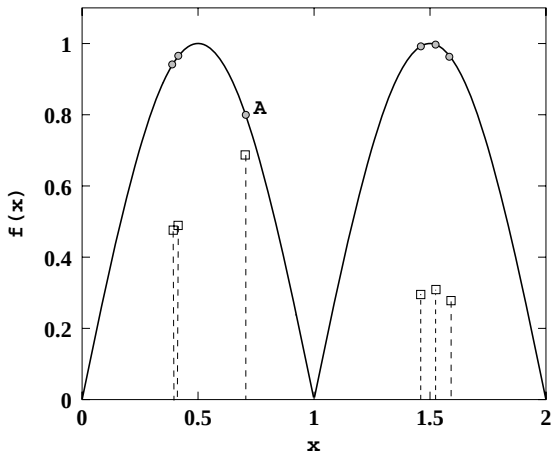


Figure 88 The shared fitness of solution A is maximum here.

in a straightforward manner with real-parameter GAs. However, there is one matter which needs discussing.

The role of any niching method is to emphasize solutions corresponding to poorly represented regions in the population. The sharing function approach achieves this task by artificially degrading the fitness of the crowded solutions in the population. Although the mating pool obtained by such a procedure may contain multiple copies of important solutions, if highly destructive recombination and mutation operators are used, some such solutions may get destroyed. Thus, there is a need to use restricted search operators. Fortunately, in most of the real-parameter crossover operators described earlier, the *search power* can be controlled with a tunable parameter. In the SBX operator, this parameter is η_c, while in the BLX operator, the parameter is α. To achieve a restricted search power, in SBX a large value of η_c must be used, while in BLX a small value of α must be used.

Deb and Kumar (1995) studied the effect of η_c on the performance of a real-parameter GA to maintain a good distribution of solutions among multiple optima on a number of multi-modal functions. We reproduce here the outcome of one such function. This function is the same five-peaked function used in the previous subsections. Instead of showing the population members on the function, Figure 89 presents a performance measure derived from the distribution of solutions in the final population. The actual number of solutions (n_i) near each of the five optima are calculated and a chi-square like deviation measure is evaluated:

$$\iota = \sqrt{\sum_{i=1}^{q+1} \left(\frac{n_i - \overline{n}_i}{\sigma_i} \right)^2}, \tag{4.67}$$

where q is the number of optima, $\overline{n}_i$ is the desired number of solutions in the i-th optimum, and σ_i is the standard deviation in the number of solutions in the i-th optimum. The desired number of solutions is calculated in proportion to the optimal function value of each optimum. For the above problems, the optimal function values are 1.0, 0.917, 0.707, 0.458 and 0.25. Thus, the corresponding desired numbers of solutions in each optimum in a population of size 100 are $\overline{n}_1 = 30$, $\overline{n}_2 = 28$, $\overline{n}_3 = 21$, $\overline{n}_4 = 14$, and $\overline{n}_5 = 7$. The $(q+1)$-th set is considered as all solutions in the population not belonging to any of the optima. For this set, $\overline{n}_{q+1} = 0$. The variances of all groups can be calculated as follows:

$$\sigma_i^2 = \begin{cases} \overline{n}_i \left(1 - \dfrac{\overline{n}_i}{N}\right) & \text{for } i = 1, 2, \ldots, q, \\ \sum_{j=1}^{q} \sigma_j^2, & \text{for } i = q+1. \end{cases}$$

Thus, the smaller the above measure ι, then the closer is the distribution to the desired distribution. In Figure 89, we observe that with η_c, the distributing ability (calculated by using equation (4.67)) of real-parameter GAs with the SBX operator gets better in solving a problem having five optima. The figure also shows that at around $\eta_c = 30$, the distributing ability of real-parameter GAs with an SBX is equivalent to that of binary-coded GAs with phenotypic sharing. It is important to realize that with such

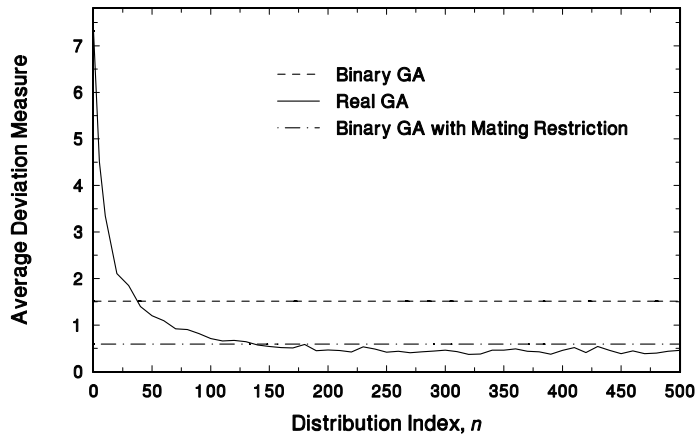


Figure 89 Real-parameter GAs with an SBX operator performs well in distributing solutions across multiple optima (Deb and Kumar, 1995). Reprinted with permission.

a large η_c, on most occasions offspring are created close to the parent solutions. This allows a slow yet steady search for better solutions in a multi-modal problem, instead of wild searches which may lead to lethal solutions.

Effect of Selection Operators

Oei et al. (1991) argued that the sharing function approach will not work well with the tournament or any ranking selection operator. Because of the proportionality law between the number of expected solutions and the optimal function value, only a proportionate selection can maintain subpopulations with a proportionate number of solutions. The investigators suggested a different niching strategy for the tournament selection. In their continuously updating sharing technique, a tournament between two solutions is played by finding the number of solutions in the vicinity of each solution in the new partial population. Instead of calculating the niche count with solutions in the old population, as in the sharing function approach described earlier, the niche count is calculated with solutions of the new population, as and when the population is formed. Between two solutions, the one with a better shared fitness value computed in the new population is chosen.

Instead of using the above continuously updated sharing approach, the investigators have also suggested a few practical implementation procedures. A niche size parameter n^* is defined by the user beforehand. For each of the two solutions participating in a tournament, the number of individuals in a niche with a σ_{share} parameter can be computed. Alternatively, the niche counts can also be computed. If the number of individuals in a niche or niche count of both solutions is less than n^*, the one with the better fitness value wins; otherwise, the one with the smaller number of crowded

individuals or smaller niche count wins. Thus, if one solution is overly crowded beyond n^* and the other is not, the second solution is chosen. However, if both solutions competing in a tournament are adequately crowded, the one which is comparatively less crowded is selected. The fitness comparison is used only when both solutions are not adequately crowded. The attractive aspect of this strategy is that the importance to fitness or to crowding can be controlled by using the n^* parameter. Since the tournament selection operator is found to have both a fast *take-over* time and a quick execution time, this shared tournament selection is now attracting a lot of attention.

Instead of using the complete population to compute the niche count or number of crowded individuals, these computations can be performed on a small subpopulation randomly chosen from the complete population. This will reduce the computational complexity at the expense of introducing noise in the niching calculations.

Another simple procedure to introduce niching with a tournament selection would be to seek a maximum of k individuals as a partner for the selection of an individual i. If any of k individuals lie within σ_{share}, that solution competes with the solution i and the one with a better fitness value wins. On the other hand, if none of the k solutions is found to lie within σ_{share} of solution i, it is automatically declared selected. Once again, the selection pressure based on fitness can be controlled by changing k. Such an idea has been implemented in a recent study (Deb, 2000).

4.6.5 Ecological GA

Davidor (1991) suggested an ecological GA to evolve both the artificial niche and species in a population. In order to achieve this, he placed the population members on a two-dimensional grid (formed to make a torus) so that each member is surrounded by eight other members. In a steady-state GA, one member in the grid is chosen at random. The genetic operations are performed within a subpopulation consisting of nine individuals (the chosen individual and eight other surrounding individuals) to create an offspring string. This string contests with parent strings and survives according to the fitness of the string. Since an offspring can only survive by replacing its parents, like population members (individuals in the basin of each optimal solution in a multi-modal problem) are expected to stay close to each other, thereby forming islands of different optimal solutions. Davidor has shown that in solving a multi-modal problem, separate islands can be formed in the torus. However, the ecological model does not have any mechanism for maintaining islands of local peaks for many generations. Thus, local peaks are formed but eventually get overwhelmed by the global optimum solution due its slow diffusion effect.

4.6.6 Other Models

Goldberg and Wang (1998) proposed an alternate niching method (coevolutionary shared niching (CSN)), which is inspired by the economic model of monopolistic competition. In a population of customers, a set of businessmen is needed to be

placed in such a way that each businessman, staying at a minimum distance to each other, serves the maximum number of customers. In a coevolutionary context of two populations involving businessmen and customers, each population seeks to maximize their separate interests, thereby creating adequately spaced niches. The usual GA population of solutions is named as the customer population here. The diversity in solutions is maintained by the usual fitness sharing scheme, where the fitness of a customer is calculated by degrading its objective function by the total number of customers its businessman serves. On the other hand, the fitness of a businessman is assigned by summing the objective function value of all of its customers. Initially, all customers and businessmen are picked from the search space at random. A selecto-recombinative GA is applied to the customer population and a selecto-mutation GA is applied to the businessman population. For each customer, the closest businessman is chosen. Thereafter, its shared fitness is calculated and the stochastic universal selection (SUS) operator is applied. The single-point crossover is used in the original study. For a businessman, a mutated solution is created and compared. If the mutated businessman is better than the original solution and is at least a critical distance away from other businessmen, it replaces the original businessman. On two test problems, it was possible to maintain stable subpopulations in all optima.

In solving an immune system problem, Smith et al. (1993) suggested an antigen–antibody matching procedure, which has an implicit fitness sharing strategy. The sharing function approach is not explicitly used here. In order to find a match in an antibody population for each antigen chosen from an antigen population, the procedure showed its ability to maintain a diverse set of antibodies covering a set of antigens. Such a coevolutionary strategy can be implemented in other optimization contexts without resorting to an explicit sharing approach.

Jelasity (1998) proposed the *universal evolutionary global optimization algorithm* (or UEGO) for multi-modal function optimization. UEGO works in different levels, starting from the complete search space. Each level embodies a number of species. Each species is designated with a center and a radius to mark its working domain in the search space. A new species is formed from two solutions of a previously formed species by comparing their objective function values with that of the centroid of the two solutions. If the centroid is worse than both parents, two new species are formed at two parent solutions. This heuristic is derived from the fact that there always lies a worse region between two optimal regions. UEGO also uses a fusion of species based on their nearness to each other, deletion of species to control over-growth of the number of species, and optimization within a species in search of better solutions representing the species. Each of these procedures involves some parameter settings. Several suggestions for fixing most of these parameters are outlined in the study. Although this procedure was used to find only one global optimum solution, multiple optimal solutions (that is, other global or local optimal solutions) are also found as byproducts of this procedure. This is because multiple species are maintained up until the end of a simulation run.

4.6.7 Need for Mating Restriction

In the previous subsections, we have seen that multiple optimal solutions in a multi-modal optimization problem can be found simultaneously by forming artificial niches (subpopulations) in the population. Each niche can be considered to represent an optimum. In order to capture a number of optima simultaneously and maintain these for many generations, a niching method is used. Niching helps to emphasize and maintain solutions around multiple optima, but it is the task of the search operators to find the best point within a niche efficiently. With the usual crossover operator applied uniformly to the whole population, some search effort is wasted in the recombination of solutions from different optima. A *speciation* method restricts mating among solutions in the basin of one optimum and helps find the best or a near-best optimum solution efficiently. If a speciation method is not used, individual optimum or near-optimum solutions may be obtained, but because of the creation of *lethal* solutions (solutions that do not represent any optimum) either a larger population size or more generations are necessary to achieve a specified accuracy in the obtained solutions. In speciation methods, only 'like individuals' are allowed to mate. Thus, the probability of creating lethal individuals is small and the search is mostly confined in the vicinity of each optimum. Thus, in the presence of both niching and speciation, niching maintains subpopulation of solutions around many optima and the speciation allows us to make a parallel search in all optimal basins to find multiple optimal solutions simultaneously.

A number of speciation methods have been suggested and implemented in genetic algorithms. Of the earlier works related to mating restriction in genetic algorithms, Hollstein's inbreeding scheme (Hollstein, 1971), where mating was allowed between close individuals in his simulation of animal husbandry problems, Booker's taxon-exemplar scheme (Booker, 1982) for restrictive mating in his simulation of learning pattern classes, Holland's suggestion of a tag-template scheme (Holland, 1975), Perry's adaptive GA for creating multiple species (Perry, 1984), Sannier and Goodman's restrictive mating in forming separate coherent groups in a population (Sannier and Goodman, 1987), Deb's phenotypic and genotypic mating restriction schemes (Deb, 1989), and Davidor's ecological model (Davidor, 1991) are just a few studies. Here, we will briefly mention the phenotypic and the genotypic mating restriction schemes.

We have developed two mating restriction schemes based on the phenotypic and genotypic distances between mating individuals (Deb, 1989). These mating restriction schemes are straightforward. In order to choose a mate for an individual, their distances (for a phenotypic mating restriction, the Euclidean distance, and for a genotypic mating restriction, the Hamming distance) are computed. If the distance is closer than a parameter σ_{mating}, they are participated in the crossover operation; otherwise, another individual is chosen at random and their distances are computed. This process is continued until a suitable mate is found or all population members are exhausted, in which case a random individual is chosen as a mate. In all simulations of this study, σ_{mating} is kept the same as the value of the parameter σ_{share} chosen for the niching method. Figure 90 shows the population after 200 generations, obtained by using the phenotypic mating restriction scheme with $\sigma_{\text{mating}} = 0.1$ and with the

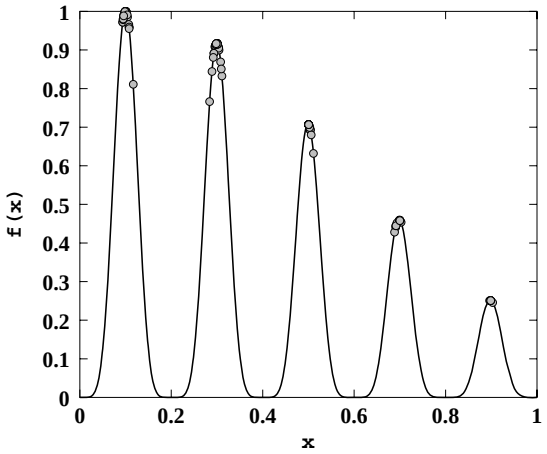


Figure 90 The population obtained after 200 generations using the sharing function and mating restriction approaches. Reproduced from Deb (1997b) (© 1997 IOP Publishing Ltd and Oxford University Press).

sharing function approach. The rest of the parameter settings are the same as before. This figure shows that the GA with niching and mating restrictions finds a better distributed population across the peaks than GAs with the sharing function approach alone (shown in Figure 87).

It is interesting to note in Figure 89 that real-parameter GAs with the SBX operator having $\eta_c \geq 140$ performed better than binary GAs with sharing and mating restriction operations.

4.7 Summary

Before understanding the working principles of multi-objective evolutionary algorithms, it is essential to know the workings of single-objective evolutionary algorithms. In this chapter we have discussed a number of evolutionary algorithms (EAs). Of them, genetic algorithms (GAs) and evolution strategy (ES) are used as baseline algorithms in most of the popular multi-objective evolutionary algorithms. Thus, we have discussed these two algorithms in somewhat more detail. Later, we have also described briefly two other evolutionary algorithms, namely evolutionary programming (EP) and genetic programming (GP). Although there exists a number of different evolutionary algorithms (Bäck et al., 1997), they are all motivated by the natural evolutionary principles. Most of these methods share a number of common properties, as follows:

1. They work with a population of solutions, instead of one solution in each iteration. By starting with a random set of solutions, an EA modifies the current population to a different population in each iteration. Working with a number of solutions

provides an EA with the ability to capture multiple optimal solutions in one single simulation run.
2. They have two distinct operations, namely selection and search. In the selection operation, better solutions in the current population are emphasized by duplicating them in the mating pool. In the search operation, new solutions are created by exchanging partial information (in the decision variable space) among solutions of the mating pool and by perturbing them in their neighborhood.
3. They do not use any gradient information during the above two operations. In addition, their representations are flexible. These properties make EAs flexible enough to be used in a wide variety of problem domains.
4. The operators use stochastic principles. Since deterministic rules are not used, EAs do not assume any particular structure of a problem to be solved.

Much of the research carried out in the area of evolutionary computation is available in a number of well-written books (Bäck, 1996; Fogel, 1995; Fogel et al., 1966; Goldberg, 1989; Gen and Cheng, 1997; Holland, 1975; Michalewicz, 1992; Mitchell, 1996; Koza, 1992, 1994; Schwefel, 1995). In addition, the following regular conference proceedings provide useful source material: The International Conference on Genetic Algorithms (ICGA), a biannual conference started in 1985, The Genetic and Evolutionary Computation Conference (GECCO), an annual conference started in 1999, The International Conference on Evolutionary Computation (ICEC), an annual conference started in 1994, The Congress on Evolutionary Computation (CEC), an annual conference started in 1999, The Evolutionary Programming Conference, an annual conference started in 1992, The Genetic Programming Conference (GP), an annual conference started in 1996, Parallel Problem Solving from Nature (PPSN) Conference, a biannual conference started in 1990, Foundations of Genetic Algorithms (FOGA), a biannual workshop started in 1990, and many other related events. The following journals also regularly publish papers on evolutionary computation: IEEE Transactions on Evolutionary Computation (TEC), published by the IEEE Press, Evolutionary Computation Journal (ECJ), published by the MIT Press, and Genetic Programming and Evolvable Machines published by the Kluwer Academic Publishers, plus several other domain-specific journals, including a couple of on-line publications.

Multi-objective optimization is a mixture of two optimization strategies: (i) minimizing the *distance* of the obtained solutions from the true optimal solutions, and (ii) maximizing the spread of solutions in the obtained set of solutions. The former task is similar to the usual task in a single-objective optimization, while the latter task is similar to finding multiple optimal solutions in a multi-modal optimization. In this chapter we have also discussed a number of multi-modal optimization techniques using evolutionary algorithms. The ability to capture multiple solutions in an EA population makes multi-modal and multi-objective evolutionary algorithms elegant and useful. With such descriptions of both of these optimization tasks, we have now set a platform from which we can now make our journey into the world of multi-objective evolutionary optimization algorithms.

Exercise Problems

1. In the mutation operator used in $(1+1)$-ES, a neighboring point is created with the following probability distribution:

$$p(x,\sigma) = \frac{a}{\sigma^2 + x^2}.$$

What must be the value of a, in order to have a valid probability distribution? The current point is $(x,\sigma) \equiv (10.0, 2.0)$. In order to perform mutation, a random number 0.723 is used. What is the new point x'?

2. In a three-variable problem, the following variable bounds are specified.

$$-5 \leq x \leq 10,$$
$$0.001 \leq y \leq 0.005,$$
$$1(10^3) \leq z \leq 1(10^4).$$

What should be the minimum string length of any point $(x,y,z)^T$ coded in binary string to achieve the minimum accuracy in the solution as given below:

(a) $\Delta x = 0.01$, $\Delta y = 1(10^{-5})$, and $\Delta z = 10$.
(b) $\Delta x = 0.1$, $\Delta y = 1(10^{-6})$, and $\Delta z = 1$.

3. We would like to use genetic algorithms to solve the following NLP problem:

$$\begin{aligned} \text{Minimize} \quad & (x_1 - 1.5)^2 + (x_2 - 4)^2 \\ \text{subject to} \quad & 4.5x_1 + x_2^2 - 18 \leq 0, \\ & 2x_1 - x_2 - 1 \geq 0, \\ & 0 \leq x_1, x_2 \leq 4. \end{aligned}$$

We decide to have three and two decimal places of accuracy for variables x_1 and x_2, respectively.

(a) How many bits are required for coding the variables?
(b) Write down the fitness function which you would be using in the selection procedure.
(c) Which schemata represent the following regions according to your coding: (a) $0 \leq x_1 \leq 2$, (b) $1 \leq x_1 \leq 2$, and $x_2 > 2$.

4. In order to solve the following problem:

$$\text{Maximize} \quad f(x) = (2x+1)^5$$

in the range $0 \leq x \leq 1$ using GAs, we are interested in the growth of the schema (1 1 0 * * * * *) under the following selection operators:

(i) Proportionate selection.

(ii) Binary tournament selection.

Assume random initial population. What proportions of the population will be occupied by the above schema after three iterations due to the above selection operators?

5. (a) Write at least four fundamental differences of GAs with most traditional search and optimization methods.
   (b) Do you support the following remark?

   Selection and mutation are the main operators in genetic algorithms.

   Justify your argument through examples/sketches.
6. For a maximization problem, an initial population of binary strings and their fitness values are shown below:

String	Fitness
∩∪∩∪∪∩	15
∪∪∩∩∩∪	5
∩∩∪∪∩∩	14
∪∩∪∩∪∪	8
∪∪∩∩∪∪	7
∩∪∪∪∩∪	10

   Using crossover and mutation probabilities $p_c = 0.8$ and $p_m = 0.1$, estimate using schema theorem how many strings representing the schema $(\cap * * \cup * *)$ would be produced after one iteration of GAs with roulette-wheel reproduction, single-point crossover, and bit-wise mutation operators. Which of the two schemata $(\cap * * * * *)$ and $(\cup * * * * *)$ is likely to take over the population in a GA run with above parameter setting? Give reasons.
7. A cantilever beam of circular cross-section (diameter d) has to be designed for minimizing the cost of the beam. The beam is of length $l = 0.30$ m and carries a maximum of $F = 1$ kN force at the free end. The beam material has $E = 100$ GPa, $S_y = 100$ MPa, and density $\rho = 7866$ Kg/m^3. The material costs Rs. 20 per Kg. There are two constraints: (i) Maximum stress in the beam ($\sigma = 32Fl/(\pi d^3)$) must not be more than the allowable strength S_y, (ii) Maximum deflection of the beam must not be more than 1 mm. The deflection at the free end due to the load F is $\delta = 64Fl^3/(3\pi Ed^4)$. The volume of the beam is $\pi d^2 l/4$. The initial population contains the following solutions: $d = 0.06$, 0.1, 0.035, 0.04, 0.12, and 0.02 m.

   (a) For each of six solutions, calculate cost in rupees and determine its feasibility in a tabular format.

   (b) Determine the fitness of each solution using Deb's penalty parameter-less constraint handling strategy. Which is the best solution in the above population?

8. **(i)** Using the simulated binary crossover (SBX) with $\eta_c = 2$, find two offspring from parent solutions $x^{(1)} = 10.53$ and $x^{(2)} = -0.68$. Use the random number 0.723.

   (ii) Show that, on an average, the polynomial probability distribution used in SBX operator produces diverging offspring (solutions which are outside the range of parent solutions) for any η_c.

(iii) How would you modify the polynomial probability distribution of the SBX operator so that the expected range of offspring is the same as that of the parent solutions?

9. Using real-coded GAs with simulated binary crossover (SBX) having $\eta = 2$, find the probability of creating children solutions in the range $0 \leq x \leq 1$ with two parents $x^{(1)} = 0.5$ and $x^{(2)} = 3.0$. Recall that for SBX operator, the children solutions are created using the following probability distribution:

$$\mathcal{P}(\beta) = \begin{cases} 0.5(\eta+1)\beta^{\eta}, & \text{if } \beta \leq 1; \\ 0.5(\eta+1)/\beta^{\eta+2}, & \text{if } \beta > 1. \end{cases}$$

where $\beta = |(c_2 - c_1)/(p_2 - p_1)|$ and c_1 and c_2 are children solutions created from parent solutions p_1 and p_2.

10. What are the differences between (μ, λ)-ES and $(\mu + \lambda)$-ES?
11. For the two parents

$$\mathbf{x}^{(1)} = (2.0, 5.0)^T, \quad \mathbf{x}^{(2)} = (4.0, 3.0)^T,$$

find the probability of finding the solution $(1.0, 1.0)^T$ using

(a) intermediate recombination operator of ES,
(b) discrete recombination operator of ES.

Use mutation strength $\sigma_i = 1.0$ for $i = 1, 2$.

12. Two parents are shown below:

$$\mathbf{x}^{(1)} = (10.0, 3.0)^T, \quad \mathbf{x}^{(2)} = (5.0, 5.0)^T.$$

Calculate the probability of finding a child in the range $x_i \in [0.0, 3.0]$ for $i = 1, 2$ using

(a) Simulated binary crossover (SBX) operator with $\eta_c = 2$,
(b) Blend crossover (BLX) operator with $\alpha = 0.67$,
(c) Fuzzy recombination (FR) operator with $d = 1.0$.

13. Apply the polynomial mutation operator to create a mutated child of the solution $x^{(t)} = 5.0$ using a random number 0.675.
14. Why is a mating restriction scheme used in a GA?
15. Consider the following population of five strings, having three choices in each place:

Strings	*Fitness*
♡♣◇♣♣♡	6
◇◇◇♡♣♣	1
♡◇◇♣◇♡	8
♣♡♣◇♡◇	2
◇♣♡♣◇♡	6

(a) How many copies would $(\heartsuit * * * * *)$ have after one iteration of a proportionate selection operation, single-point crossover operator with $p_c = 0.8$, and bit-wise mutation operator with $p_m = 0.1$?

(b) What proportion of the entire search space is represented by the schema (♣ ♢ ∗ ∗ ∗ ∗)?

(c) Assuming that the above representation maps the integer space, how many discrete regions are represented by the schema (♣ ∗ ∗ ♢ ♡ ∗)?

16. Consider the following non-linear programming problem:

$$\begin{aligned} \text{Minimize} \quad & f(\mathbf{x}) = x_1^2 - x_1 x_3 + x_2^2, \\ \text{subject to} \quad & g_1(\mathbf{x}) \equiv x_1 - 2 \geq 0, \\ & g_2(\mathbf{x}) \equiv x_1 x_2 x_3 - 10 \geq 0. \end{aligned}$$

Calculate the fitness of each solution of the population shown below using Powell and Skolnick's constraint-handling approach and Deb's constraint-handling approach:

$$\begin{aligned} \mathbf{x}^{(1)} &= (1,2,5)^T, \quad \mathbf{x}^{(2)} = (6,1,2)^T, \quad \mathbf{x}^{(3)} = (3,4,1)^T, \\ \mathbf{x}^{(4)} &= (0,2,1)^T, \quad \mathbf{x}^{(5)} = (5,1,1)^T, \quad \mathbf{x}^{(6)} = (1,2,1)^T. \end{aligned}$$

Normalize the constraints and use $R = 1$.

17. We would like to find all four maxima of the following multimodal function defined in the range $x_1, x_2 \in [0,1]$ using GAs with sharing functions:

$$F(x_1, x_2) = (1 - x_1 - x_2) \sum_{i=1}^{2} f_i(x_i),$$

where $f_i(x_i) = \sin^2(2\pi x_i)$. Assume optima lie at $x_1, x_2 = 0.25$ and 0.75. To avoid the stochastic sampling error, we require at least 5 copies at each optimum. What should be the minimum population size?

18. For the following maximization problem

$$\begin{aligned} \text{Maximize} \quad & f(x) = |\sin(\pi x)| \\ & 0 \leq x \leq 2, \end{aligned}$$

the following six strings are used.

(1) 110110 (2) 101100 (3) 011101
(4) 001011 (5) 110000 (6) 101110

Using $\sigma_{\text{share}} = 0.5$ and $\alpha = 1$, calculate the niche count and shared fitness value of each solution. Present your results in a tabular format.

Determine how many copies are expected in the population by the action of a proportionate selection alone in the left half and right half of the search space for the following two cases:

(a) Without sharing approach,

(b) With sharing approach.

19. In a binary decision problem with expected payoff $f_1 = 10$ for decision one and $f_0 = 5$ for decision zero, calculate the expected number of decision one in the next generation under the following selection schemes:

(i) Roulette-wheel selection only.

(ii) Roulette-wheel selection with genotypic sharing ($\sigma_{share} = 1$, $\alpha = 1$).

(iii) Roulette-wheel selection with genotypic sharing ($\sigma_{share} = 2$, $\alpha = 1$).

Assume the population currently contains 70 ones and 30 zeros.

20. For the bimodal maximization problem

$$f(x) = x \sin^2 \pi x, \qquad 0 \leq x \leq 2,$$

following four solutions are used in a population:

$$x^{(1)} = 0.4, \quad x^{(2)} = 1.6, \quad x^{(3)} = 1.2, \quad x^{(4)} = 0.9.$$

Calculate the shared fitness value of each of the above solutions using $\sigma_{\text{share}} = 0.4$.