CHAPTER 2

Fundamentals of Genetic Algorithms

ALEXANDRE P. ALVES DA SILVA and DJALMA M. FALCÃO

2.1 INTRODUCTION

Research on genetic algorithms (GAs) has shown that the initial proposals are incapable of solving hard problems in a robust and efficient way. Usually, for large-scale optimization problems, the execution time of first-generation GAs increases dramatically whereas solution quality decreases. The aim of this chapter is to point out the main design issues in tailoring GAs to large-scale optimization problems. Important topics such as encoding schemes, selection procedures, and self-adaptive and knowledge-based operators are discussed.

2.2 MODERN HEURISTIC SEARCH TECHNIQUES

Optimization is the basic concept behind the application of GAs, or any other evolutionary algorithm [1–3], to any field of interest. Over and above the problems in which optimization itself is the final goal, it is also a way for achieving (or the main idea behind) modeling, forecasting, control, simulation, and so forth.

Traditional optimization techniques begin with a single candidate and search iteratively for the optimal solution by applying static heuristics. On the other hand, the GA approach uses a population of candidates to search several areas of a solution space, simultaneously and adaptively.

Evolutionary computation allows precise modeling of the optimization problem, although not usually providing mathematically optimal solutions. Another advantage of using evolutionary computation techniques is that there is no need for having an explicit objective function. Moreover, when the objective function is available, it does not have to be differentiable. Genetic algorithms have been most commonly applied to solve combinatorial optimization problems. Combinatorial optimization usually involves a huge number of possible solutions, which makes the use of

Modern Heuristic Optimization Techniques. Edited by K. Y. Lee and M. A. El-Sharkawi

enumeration techniques (e.g., cutting plane, branch and bound, or dynamic programming) hopeless.

Thermal unit commitment, hydrothermal coordination, expansion planning (generation, transmission, and distribution), reactive compensation placement, maintenance scheduling, and so forth, have the typical features of a large-scale combinatorial optimization problem. In problems of this kind, the number of possible solutions grows exponentially with the problem size. Therefore, the application of optimization methods to find the optimal solution is computationally impracticable. Heuristic search techniques are frequently employed in this case for achieving high-quality solutions within reasonable run time.

Among the heuristic search methods, there are the ones that apply local search (e.g., hill climbing) and the ones that use a nonconvex optimization approach, in which cost-deteriorating neighbors are accepted also. The most popular methods that go beyond simple local search are GAs [4–7] (and other evolutionary techniques, such as evolutionary programming, evolutionary strategies, etc.), simulated annealing (SA) [8], and tabu search (TS) [9]. Particle swarm [10] is another optimization technique that has shown great potential lately. However, more experience is still necessary to indicate its efficiency and robustness.

Simulated annealing uses a probability function that allows a move to a worse solution with a decreasing probability as the search progresses. With GAs, a pool of solutions is used, and the neighborhood function is extended to act on pairs of solutions. Tabu search uses a deterministic rather than stochastic search. Tabu search is based on a neighborhood search with local optima avoidance. In order to avoid cycling, a short-term adaptive memory is used in TS. Genetic algorithms have a basic distinction when compared with other methods based on stochastic search. They often use a coding (genotypic space) for representing the problem. The other methods often solve the optimization problem in the original representation space (phenotypic).

The most rigorous global search methods have asymptotic convergence proof (also known as convergence in probability); that is, the optimal solution is guaranteed to be found if infinite time is available. Among SA, GA, and TS algorithms, simulated annealing and genetic algorithms are the only ones with proof of convergence. However, there is no such proof for the canonical GA [11], that is, the one with proportional selection (Section 2.6.1.6) and crossover/mutation with constant probabilities (Section 2.6.4), which in fact is divergent.

Although all the mentioned algorithms have been applied successfully to real-world problems, several of their crucial parameters have been selected empirically. Theoretical knowledge of the impact of these parameters on convergence is still an open problem. In fact, there is only a beginning of theoretical results for tabu and particle swarm searches.

The choice of representation for a GA is fundamental to achieving good results. Encoding allows a kind of *tunneling* in the original search space. That means a particle has a nonzero probability of passing a potential barrier even when it does not have enough energy to jump over the barrier. The tunneling idea is that rather than escaping from local minima by random uphill moves, escape can be achieved with the quantum tunnel effect. It is not the height of the barrier that determines the rate

of escape from a local optimum, but rather its width relative to current population variance.

The main shortcoming of the standard SA procedure is the slow asymptotic convergence with respect to the *temperature* parameter T. In the standard SA algorithm, the cooling schedule for asymptotic global convergence is inversely proportional to the logarithm of the number of iterations (k); that is, $T(k) = c/(1 + \log k)$. The constant c is the largest depth of any local minimum that is not the global minimum. Convergence in probability cannot be guaranteed for faster cooling rates (e.g., lower values for c).

Tabu search owes its efficiency to an experience-based fine-tuning of a large collection of parameters. Tabu search is a general search scheme that must be tailored to the details of the problem at hand. Unfortunately, as mentioned before, there is little theoretical knowledge for guiding this tailoring process.

Heuristic search methods utilize different mechanisms in order to explore the state space. These mechanisms are based on three basic features:

- The use of memoryless search (e.g., standard SA and GA) or adaptive memory (e.g., TS);
- The kind of neighborhood exploration used, that is, random (e.g., SA and GAs) or systematic (e.g., TS); and
- The number of current solutions taken from one iteration to the next (GAs, as opposed to SA and TS, take multiple solutions to the next iteration).

The combination of these mechanisms for exploring the state space determines the search diversification (global exploration) and intensification (local exploitation) capabilities. The standard SA algorithm is notoriously deficient with respect to the diversification aspect. On the other hand, the standard GA is poor in intensification.

When the objective function has very many equally good local minima, wherever the starting point is, a small random disturbance can avoid the small local minima and reach one of the good ones, making this an appropriate problem for SA. However, SA is less suitable for a problem in which there is one global minimum that is much better than all the other local ones. In this case, it is very important to find that valley. Therefore, it is better to spend less time improving any set of parameters and more time working with an ensemble to examine different regions of the space. This is what GAs do best. Hybrid methods have been proposed in order to improve the robustness of the search.

2.3 INTRODUCTION TO GAs

Genetic algorithms operate on a population of individuals. Each individual is a potential solution to a given problem and is typically encoded as a fixed-length binary string (other representations have also been used, including character-based and real-valued encodings, etc.), which is an analogy with an actual chromosome. After an initial population is randomly or heuristically generated, the algorithm evolves the

population through sequential and iterative application of three operators: selection, crossover, and mutation. A new generation is formed at the end of each iteration.

For large-scale optimization problems, the initial population can incorporate prior knowledge about solutions. This procedure should not drastically restrict the population diversity, otherwise premature convergence could occur. Typical population sizes vary between 30 and 200. The population size is usually set as a function of the chromosome length.

The execution of a GA iteration is basically a two-stage process. It starts with the current population. Selection is applied to create an intermediate population (mating pool). Then, crossover and mutation are applied to the intermediate population to create the next generation of potential solutions. Although a lot of emphasis has been placed on the three above-mentioned operators, the coding scheme and the fitness function are the most important aspects of any GA, because they are problem dependent.

The original explanation about how GAs could result in robust search relied on the argument of hyperplane sampling. In order to understand this idea, assume a problem encoded with 3 bits. The search space is represented by a cube with one of its vertices at the origin 000. For example, the upper surface of the cube contains all the points of the form $*1*$, where $*$ could be either 0 or 1.

A string that contains the symbol $*$ is referred to as a schema. It can be viewed as a (hyper)plane representing a set of solutions with common properties. The order of a schema is the number of fixed positions present in the string. The defining length is the distance between the first and last fixed positions of a particular schema. Building blocks are highly fit strings of low defining length and low order.

The true fitness of a hyperplane partition corresponds with the average fitness of all strings that lie in that hyperplane. Genetic algorithms use the population as a sample for estimating the fitness of that hyperplane partition. After the initial generation, the pool of new strings is biased toward regions that have previously contained strings that were above average with respect to previous populations. In order to further explore the search space, crossover and mutation generate new sample points while partially preserving the distribution of strings that is observed after selection. Recently, the widespread belief that GAs are robust by virtue of their schema processing has been proved to be false [12].

In the following sections, several important design stages of a GA are presented. Section 2.4 shows different possibilities for encoding. It emphasizes the importance of the encoding scheme on GA convergence. Section 2.5 treats the formulation of the fitness function. Section 2.6 presents different propositions for the selection, crossover, and mutation operators. Parameter control in GAs is addressed in Section 2.6, too. This chapter concludes with a short presentation of niching methods and parallel GAs.

2.4 ENCODING

In order to apply a GA to a given problem, the first decision one has to make is the kind of genotype the problem needs. That means a decision must be taken on how the

parameters of the problem will be mapped into a finite string of symbols, known as genes (with constant or dynamic length), encoding a possible solution in a given problem space. The issue of selecting an appropriate representation is crucial for the search. The symbol alphabet used is often binary, though other representations have also been used, including character-based and real-valued encodings.

Many GA applications use a binary alphabet, and their length is constant during the evolutionary process. Also, all the parameters decode to the same range of values and are allocated the same number of bits for the genes in the string. A problem occurs when a gene may only have a finite number of discrete valid values if a binary representation is used. If the number of values is not a power of 2, then some of the binary codes are redundant (i.e., they will not correspond to any valid gene value). The most popular compromise is to map the invalid code to a valid one.

Another shortcoming of binary encoding is the so-called Hamming cliffs (e.g., although the integers 3 and 4 are neighbors in decimal representation, the Hamming distance between the corresponding binary representation, i.e., [0 1 1] and [1 0 0], respectively, is three [different bits]). It is worthwhile to mention that Gray coding, although frequently recommended as a solution to Hamming cliffs, because adjacent numbers differ by a single bit, has an analogous drawback for numbers at the opposite extremes of the decimal scale (e.g., the minimum and maximum gene values differ by only 1 bit, too). Binary encoding can also introduce an additional nonlinearity, thus making the combined objective function (the one in the genotype space) more multimodal than the original one (in the phenotype space).

At the beginning of GA research, the binary representation was recommended because it was supposed to give the largest number of schemata (plural of schema), therefore providing the highest degree of implicit parallelism. However, new interpretations have shown that high-cardinality alphabets (e.g., real numbers) can be more effective due to the higher expression power and low effective cardinality [13–15]. Complex applications suggest nonbinary alphabets. Integer or continuous-valued genes are typically used in large-scale function optimization problems. Another advantage of nonbinary representations, particularly the real-valued one, is the easy definition of problem-specific operators.

When using binary coding, the positions of the genes in the chromosome are extremely important for a successful GA design, unless uniform crossover is applied (Section 2.6.2). A bad choice can make the problem harder than necessary. Therefore, correlated binary genes should be coded together in order to form building blocks, thus diminishing the disruptive effects of crossover. However, this information is usually unavailable beforehand.

Epistasis is a possible measure of problem difficulty for GAs. It represents the interaction among different genes in a chromosome. This depends on the extent to which the change in chromosome fitness resulting from a small change in one gene varies according to the values of other genes. The higher the epistasis level, the harder the problem is. This is obviously also true when applying uniform crossover or real-valued encoding. As mentioned earlier, a possibility for making the gene ordering irrelevant is to apply uniform crossover, because the result of this operation

is not affected by the positions of the genes. The same goal can be achieved with real-valued encoding and recombination operators that also turn the genes positions irrelevant (Section 2.6.2). However, making the gene ordering irrelevant does not necessarily mean an easier way to a good solution.

One possible answer for the binary gene position problem is to use an operator called inversion. This is implemented by extending every gene by adding the position it occupies in the string. Inversion is interesting because it can freely mix the genes of the same string in order to put together the building blocks, automatically, during evolution (e.g., [(2 1) (3 0) (1 0) (4 1)], where the first number is a bit tag that indexes the bit and the second one represents the bit value; i.e., (3 0) means that the third bit is equal to zero). At first sight, the inversion operator looks very useful when the correlated parameters are not known *a priori*. With the association of a position to every gene, the string can be correctly reordered before evaluation. However, for large-scale problems, inversion has not demonstrated any utility. Reordering greatly expands the search space, making the problem much more difficult to solve.

Therefore, the very hard encoding problem still remains in the hands of the designer. In order to achieve good performance for large tasks, GAs must be matched to the search problem at hand. The only way to succeed is by using domain-specific knowledge to select an appropriate representation.

2.5 FITNESS FUNCTION

Each string is evaluated and assigned a fitness value after the creation of an initial population. It is useful to distinguish between the objective function and the fitness function used by a GA. The objective function provides a measure of performance with respect to a particular set of gene values, independently of any other string. The fitness function transforms that measure of performance into an allocation of reproductive opportunities (i.e., the fitness of a string is defined with respect to other members of the current population). After decoding the chromosomes (i.e., applying the genotype to phenotype transformation), each string is assigned a fitness value. The phenotype is used as input to the fitness function. Then, the fitness values are employed to relatively ponder the strings in the population.

The specification of an appropriate fitness function is crucial for the correct operation of a GA [16]. As an optimization tool, GAs face the task of dealing with problem constraints [17]. Crossover and mutation, that is, the perturbation (variation) mechanism of GAs, are general operators that do not take into account the feasibility region. Therefore, infeasible offspring appear quite frequently. There are four basic techniques for handling constraints when using GAs.

The simplest alternative is the rejecting technique in which infeasible chromosomes are discarded all over the generations. A different strategy is the repairing procedure, which uses a converter to transform an infeasible chromosome into a feasible one. Another possible technique is the creation of problem-specific genetic operators to preserve feasibility of chromosomes.

The previous procedures do not generate infeasible solutions. This is not usually an advantage. In fact, for large-scale, highly constrained optimization problems, this is certainly a great drawback. Particularly for power system problems, where the optimal solutions usually are on the boundaries of feasible regions, the above-mentioned techniques for handling constraints often lead to poor solutions. One possible way for overcoming this drawback is to apply the repairing procedure only to a fraction (10%, for instance) of the infeasible population.

It has been suggested that constraint handling for such types of optimization problem should be performed allowing search through infeasible regions. Penalty functions allow the exploration of infeasible subspaces [18]. An infeasible point close to the optimum solution generally contains much more information about it than a feasible point far from the optimum. On the other hand, the design of penalty functions is difficult and problem dependent. Usually, there is no *a priori* information about the distance to optimal points. Therefore, penalty methods consider only the distance from the feasible region. Penalties based on the number of violated constraints do not work well.

There are two main possible forms to build a fitness function with penalty term: the addition and multiplication forms. The former is represented as $g(\underline{x}) = f(\underline{x}) + p(\underline{x})$; where for maximization problems $p(\underline{x}) = 0$ for feasible points, and $p(\underline{x}) < 0$ otherwise. The maximum absolute $p(\underline{x})$ value cannot be greater than the minimum absolute $f(\underline{x})$ value for any generation, in order to avoid negative fitness values. The multiplication form is represented as $g(\underline{x}) = f(\underline{x})p(\underline{x})$; where for maximization problems $p(\underline{x}) = 1$ for feasible points, and $0 \leq p(\underline{x}) < 1$ otherwise.

The penalty term should vary not only with respect to the degree of constraint violations, but also with respect to the GA iteration count. Therefore, besides the amount of violation, the penalty term usually contains variable penalty factors, too (one per violated constraint). The key for a successful penalty technique is the proper setting of these penalty factors. Small penalty factors can lead to infeasible solutions, and very large ones totally neglect infeasible subspaces. In average, the absolute values of the objective and penalty functions should be similar. At least in theory, the parameters of the penalty functions can, also, be encoded as GA parameters. This procedure creates an adaptive method, which is optimized as the GA evolves toward the solution.

In summary, the main problems associated with the fitness function specification are the following:

- Dependence on whether the problem is related to maximization or minimization;
- When the fitness function is noisy for a nondeterministic environment [19];
- The fitness function may change dynamically as the GA is executed;
- The fitness function evaluation can be so time consuming that only approximations to fitness values can be computed;
- The fitness function should allocate very different values to strings in order to make the selection operator work easier (Section 2.6.1.6);

- It must consider the constraints of the problem; and
- It could incorporate different subobjectives.

The fitness function is a black box for the GA. Internally, this may be achieved by a mathematical function, a simulator program, or a human expert that decides the quality of a string. At the beginning of the iterative search, the fitness function values for the population members are usually randomly distributed and widespread over the problem domain. As the search evolves, particular values for each gene begin to dominate. The fitness variance decreases as the population converges. This variation in fitness range during the evolutionary process often leads to the problems of premature convergence and slow finishing.

2.5.1 Premature Convergence

A frequent problem with GAs, known as deception, is that the genes from a few comparatively highly fit (but not optimal) individuals may rapidly come to dominate the population, causing it to converge on a local maximum or stagnate somewhere in the search space. Once the population has converged, the ability of the GA to continue searching for better solutions is nearly eliminated. Crossover (Section 2.6.2) of almost identical chromosomes generally produces similar offspring. Only mutation (Section 2.6.3), with its random perturbation mechanism, remains to explore new regions of the search space.

The schema theorem, which was proved erroneous a few years ago [20], says that reproductive opportunities should be given to individuals in proportion to their relative fitnesses. However, by doing that, premature convergence occurs because the population is not infinite (basic hypothesis of the theorem). This is due to genetic drift (Section 2.7). In order to make GAs work effectively on finite populations, the (proportional) way individuals are selected for reproduction must be modified. Different ways of performing selection are described in Section 2.6.1 The basic idea is to control the number of reproductive opportunities each individual gets. The strategy is to compress the range of fitnesses, without loosing selection pressure (Section 2.5.2), and avoid any super-fit individual from suddenly dominating the population.

2.5.2 Slow Finishing

This is the opposite problem of premature convergence. After many generations, the population has almost converged, but it is still possible that the global maximum (or a high-quality local one) has not been found. The average fitness is high, and the difference between the best and the average individuals is small. Therefore, there is insufficient variance in the fitness function values to localize the maxima.

The same techniques used to tackle premature convergence are used also for fighting slow finishing. An expansion of the range of population fitnesses is produced, instead of a compression. Both procedures are prone to bad remapping (underexpansion or overcompression) due to super-poor or super-fit individuals.

2.6 BASIC OPERATORS

In this section, several important design issues for the selection, crossover, and mutation operators are presented. Selection implements the survival of the fittest according to some predefined fitness function. Therefore, high-fitness individuals have a better chance of reproducing, whereas low-fitness ones are more likely to disappear. Selection alone cannot introduce any new individuals into the population (i.e., it cannot find new points in the search space). Crossover and mutation are used to explore the solution space.

Crossover, which represents mating (recombination) of two individuals, is performed by exchanging parts of their strings to form two new individuals (offspring). In its simplest form, substrings are exchanged after a crossover point is randomly determined. The crossover operator is applied with a certain probability, usually in the range [0.5, 1.0]. This operator allows the evolutionary process to move toward promising regions of the search space. It is likely to create even better individuals by recombining portions of good individuals. The new offspring created from mating, after being subject to mutation, are put into the next generation.

The purpose of the mutation operator is to maintain diversity within the population and inhibit premature convergence to local optima by randomly sampling new points in the search space. The GA stopping criterion may be specified as a maximal number of generations or as the achievement of an appropriate level for the generation average fitness (stagnation).

2.6.1 Selection

Selection, more than crossover and mutation, is the operator responsible for determining the convergence characteristics of GAs [21, 22]. Selection pressure is the degree to which the best individuals are favored [23]. The higher the selection pressure, the more the best individuals are favored. The selection intensity of GAs is the expected change of average fitness in a population after selection is performed. Analyses of selection schemes show that the change in mean fitness at each generation is a function of the population fitness variance.

The convergence rate of a GA is determined largely by the magnitude of the selection pressure. Higher selection pressures imply higher convergence rates. If the selection pressure is too low, the convergence rate will be slow, and the GA will unnecessarily take longer to find a high-quality solution. If the selection pressure is too high, it is very probable that the GA will converge prematurely to a bad solution. In fact, selection schemes should also preserve population diversity, in addition to providing selection pressure. One possibility to achieve this goal is to maximize the product of selection intensity and population fitness standard deviation. Therefore, if two selection methods have the same selection intensity, the method giving the higher standard deviation of the selected parents is the best choice.

Many selection schemes are currently in use. They can be classified in two groups: proportionate selection and ordinal-based selection. Proportionate-based procedures select individuals based on their fitness values relative to the fitness of the other

individuals in the population. Ordinal-based procedures select individuals not based on their fitness but based on their rank within the population.

An ordinal selection scheme has a fundamental advantage over a proportional selection one. The former is translation and scale invariant (i.e., the selection pressure does not change when every individual's fitness is multiplied and added by a constant. The selection intensity of proportionate selection is the only one that is sensitive to the current population distribution [24]. However, conclusive statements about the performance of rank-based selection schemes are difficult to make because, by suitable (but tricky!) adjustment, proportionate selection can give similar performance.

2.6.1.1 Tournament Selection This selection scheme is implemented by choosing some number of individuals randomly from the population, copying the best individual from this group into the intermediate population, and repeating it until the mating pool is complete. Tournaments are frequently held only between two individuals. Bigger tournaments are also used with arbitrary group sizes (not too big in comparison with the population size). Tournament selection can be implemented very efficiently because no sorting of the population is required.

The tournament procedure selects the mating pool without remapping the fitnesses. By adjusting the tournament size, the selection pressure can be made arbitrarily large or small. Bigger tournaments have the effect of increasing the selection pressure, because below-average individuals do not have good chances of winning a competition.

2.6.1.2 Truncation Selection In truncation selection, only a subset of the best individuals is chosen to be possibly selected, with the same probability. This procedure is repeated until the mating pool is complete. As a sorting of the population is required, truncation selection has a greater time complexity than tournament selection. As in tournament selection, there is no fitness remapping in truncation selection.

2.6.1.3 Linear Ranking Selection The individuals are sorted according to their fitness values, and the last position is assigned to the best individual, whereas the first position is allocated to the worst one. The selection probability is linearly assigned to the individuals according to their ranks. All individuals get a different selection probability, even when equal fitness values occur.

2.6.1.4 Exponential Ranking Selection Exponential ranking selection differs from linear ranking selection only in that the probabilities of the ranked individuals are exponentially weighted.

2.6.1.5 Elitist Selection Preservation of the elite solutions from the preceding generation ensures that the best solutions known so far will remain in the population and have more opportunities to produce offspring. Elitist selection is used in combination with other selection strategies.

2.6.1.6 Proportional Selection This is the first selection method proposed for GAs. The probability that an individual will be selected is simply proportionate to its fitness value. The time complexity of the method is the same as in tournament selection. This mechanism works only if all fitness values are greater than zero. The selection probabilities strongly depend on the scaling of the fitness function. In fact, most of the scaling procedures described in the next sections have been proposed to keep proportional selection working. One big drawback of proportional selection is that the selection intensity is usually low, because a single individual, either the fittest or the worst, dictates the degree of compression of the range of fitnesses. This is quite common even during the early stage of the search, when the population variance is high. Negative selection intensity is also possible.

Notice that in ordinal-based selection schemes, the effect of extreme individuals is negligible, irrespective of how much greater or smaller their fitnesses are than the rest of the population. Therefore, despite its popularity inside the power system research community, proportional selection (i.e., roulette wheel) is usually an inferior scheme. There are different scaling operators that help in separating the fitness values in order to improve the work of the proportional selection mechanism. The most common ones are linear scaling, sigma truncation, and power scaling.

2.6.1.6.1 Linear Scaling Linear scaling (i.e., $f' = af + b$) works well except when most populations members are highly fit, but a few very poor individuals are present. The coefficients a and b are usually chosen to enforce equality of the objective and fitness functions average values and also cause maximum scaled fitness to be a specified multiple (usually two) of the average fitness. These two conditions ensure that average population members receive one offspring copy on average, and the best receives the specified multiple number of copies. Notice that proportional selection with linear scaling is not the same as linear ranking selection.

2.6.1.6.2 Sigma Truncation In order to overcome the presence of super-poor individuals, the use of population variance information has been suggested to preprocess objective function values before scaling. This procedure subtracts a constant from the objective function values; $f' = \max[0, f - (\bar{f} - d\sigma)]$, where $\bar{f}$ is the mean objective function value in the population. The constant d is chosen as a multiple (between 1 and 3) of the population standard deviation, and negative results are arbitrarily set to zero.

2.6.1.6.3 Power Scaling Another possibility is power scaling (i.e., $f' = f^p$). In general, the p value is problem dependent and may require adaptation during a run to expand or compress the range of fitness function values. The problem with all fitness scaling schemes is that the degree of compression can be determined by a single extreme individual, degrading the GA performance.

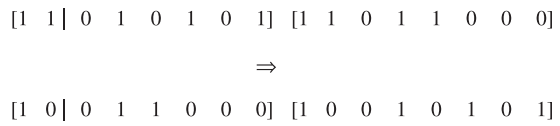

FIGURE 2.1 Example of one-point crossover.

2.6.2 Crossover

Crossover is a very controversial operator due to its disruptive nature (i.e., it can split important information). In fact, the usefulness of crossover is problem dependent. The traditional GA uses one-point crossover (Fig. 2.1), where the two parents are each cut once at specific points and the segments located after the cuts exchanged. The positions of the bits in the schema determine the likelihood that these bits will remain together after crossover. Obviously, an order-1 schema is not affected by recombination, because the critical bit is always inherited by one of the offspring.

The crossover operator presented above can be generalized in order to apply multiple-point crossover. However, more than two crossover points, although giving a better exploration capacity, can be too disruptive. The crossover mechanism can be better visualized treating strings as rings. In Fig. 2.2, two-point crossover is applied to the example shown in Fig. 2.1. Each offspring takes one ring segment, in between adjacent cut points, from each parent. The contiguous ring segment(s) is taken from the other parent. For more than two crossover points, this procedure is repeated until the last segment is filled. An extra cut is assumed at the beginning of the string (i.e., between genes g8 and g1) for an odd number of cut points.

From the linear string point of view, the elements in between the two crossover points are swapped between two parents to form two offspring (Fig. 2.2). One-point crossover can be represented by the ring geometry as a two-point crossover

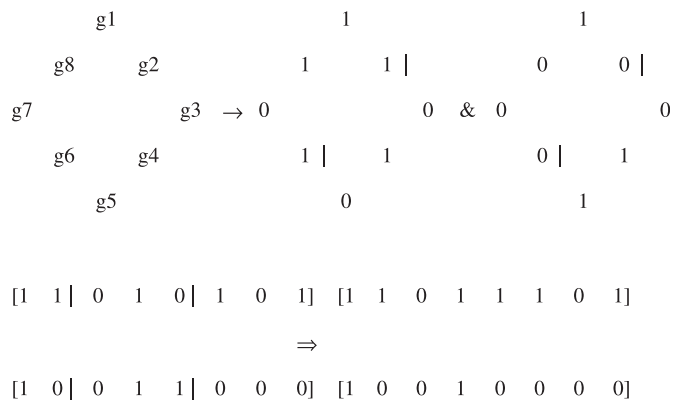

FIGURE 2.2 Ring representation and two-point crossover.

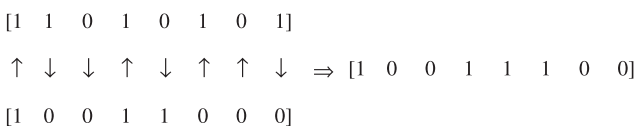

FIGURE 2.3 Example of uniform crossover, where each arrow points to the randomly picked gene value.

with the first cut point always between genes g8 and g1. For multiple-point crossover, the cut points can be anywhere, as long as they are not the same.

Uniform crossover is another important recombination mechanism [25]. Offspring is created by randomly picking each bit from either of the two parent strings (Fig. 2.3). This means that each bit is inherited independently from any other bit. Uniform crossover has the advantage that the ordering of the genes is irrelevant in terms of splitting building blocks.

Uniform crossover is more disruptive than two-point crossover. On the other hand, two-point crossover performs poorly when the population has largely converged because of the inability to promote diversity. For small populations, which is not usually the case for large-scale problems, more disruptive crossover operators such as uniform or m-point ($m > 2$) may perform better because they help overcome the limited amount of information.

Reduced surrogates can be used to improve two-point crossover exploration ability. It is highly recommended for large-scale problems. The idea is to ignore all bits that are equivalent in the two parent strings (Fig. 2.4). Afterwards, crossover is applied on the reduced surrogates (i.e., only one possible cut is considered between any pair of nonequivalent bits).

Notice that the reduced surrogate form implements the original crossover operation in an unbiased way. For example, the cut points between genes 2|3, 3|4, and 4|5 produce the same effect on offspring. Therefore, two-point reduced surrogate crossover considers these cut points as one single possible cross point.

The crossover operator can be redefined for real-valued encoding. Different combinations have been utilized (e.g., a convex combination such as $\lambda_1 \underline{x}_1 + \lambda_2 \underline{x}_2$, where $\lambda_1, \lambda_2 \in i^+$ and $\lambda_1 + \lambda_2 = 1$). One possibility is to take the average of the two corresponding parent genes. The square-root of the product of the two values can also be used. Another possibility is to take the difference between the two values and add it to the higher or subtract it from the lower.

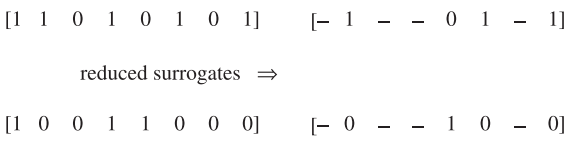

FIGURE 2.4 Implementation of reduced surrogates to improve crossover exploration capability.

2.6.3 Mutation

The GA literature has reflected a growing recognition of the importance of mutation in contrast with viewing it as responsible for reintroducing inadvertently lost gene values. The mutation operator is more important at the final generations when the majority of the individuals present similar quality. As is shown in Section 2.6.4, a variable mutation rate is very important for the search efficiency. Its setting is much more critical than that of crossover rate.

In the case of binary encoding, mutation is carried out by flipping bits at random, with some small probability (usually in the range [0.001; 0.05]). For real-valued encoding, the mutation operator can be implemented by random replacement (i.e., replace the value with a random one). Another possibility is to add/subtract (or multiply by) a random (e.g., uniformily or Gaussian distributed) amount. Mutation can also be used as a hill-climbing mechanism.

2.6.4 Control Parameters Estimation

Typical values for the population size, crossover, and mutation rates have been selected in the intervals [30, 200], [0.5, 1.0], and [0.001, 0.05], respectively. Fixed crossover and mutation operators do not provide enough search power for tackling large-scale optimization problems. Tuning parameters manually is common practice in GA design. Often, one parameter is tuned at a time in order to avoid the challenging task of simultaneous estimation. However, as they strongly interact in complex forms, this tuning procedure is prone to suboptimality.

In fact, any static set of parameters is inappropriate, regardless of how they are tuned. The GA search technique is an adaptive process, which requires continuous tracking of the search dynamics. Therefore, the use of constant parameters leads to inferior performance. For example, it is obvious that large mutations can be helpful during early generations to improve the GA exploration capability. This is not the case for the end of the search, when small mutation steps are needed to fine-tune suboptimal solutions.

The proper way for dealing with this problem is by using parameters that are functions of the number of generations. Deterministic rules are frequently applied for implementing this idea. However, besides being very difficult to define, they fail to take into account the actual progress of the population performance. Adaptive rules based on population variance, or even the search for optimal parameters as part of the GA processing (i.e., including parameters as part of the chromosomes), seem to be more promising [26, 27].

2.7 NICHING METHODS

Two agents cause the reduction of population fitness variance at each generation. The first, selection pressure, multiplies copies of the fitter individuals. The other agent is independent of fitness. It is called genetic drift [28] and is due to the stochastic nature

of the selection operator (i.e., bias on the random sampling of the population). When there is lack of selection pressure, genetic drift is responsible for premature convergence. The GA still ends up on a single peak, even when there are several ones of equal fitness.

Therefore, even when multiobjective optimization is not the main goal, the identification of multiple optima is beneficial for the GA performance. Niching methods extend standard GAs by creating stable subpopulations around global and local optimal solutions. Niching methods maintain population diversity and allow GAs to explore many peaks simultaneously. They are based on either fitness sharing or crowding schemes [29].

Fitness sharing decreases each element's fitness proportionately to the number of similar individuals in the population (i.e., in the same niche). The similarity measure is based on either the genotype (e.g., Hamming distance) or the phenotype (e.g., Euclidian distance). On the contrary, crowding schemes do not require the setting of a similarity threshold (niche radius). Crowding implicitly defines neighborhood by the application of tournament rules. It can be implemented as follows. When an offspring is created, one individual is chosen, from a random subset of the population, to disappear. The chosen one is the element that most closely resembles the new offspring.

Another idea used by niching methods is restricted mating. This mechanism avoids the recombination of individuals that do not belong to the same niche. Highly fit, but not similar, parents can produce highly unfit offspring. Restricted mating is based on the assumption that if similar parents (i.e., from the same niche) are mated, then offspring will be similar to them.

It is important to notice that similarity of genotypes does not necessarily imply similarity of the corresponding phenotypes. The hypothesis that highly fit parents generate highly fit offspring is valid only under the occurrence of building blocks and low epistasis. When the genes strongly interact, there is no guarantee that these offspring will not be lethals.

2.8 PARALLEL GENETIC ALGORITHMS

In several cases, the application of GAs to actual problems in business, engineering, and science requires long processing time (hours, days). Most of this time is usually spent in thousands of fitness function evaluations that may involve long calculations. Fortunately, these evaluations can be performed quite independently of each other, which makes GAs very adequate for parallel processing.

The many ways parallel GAs can be implemented depends on how the population is dealt with and on the computer hardware available. A single population or multiple populations may be considered. In the last case, the populations may be isolated or communicate by exchanging individuals or some other information. The computer hardware, on the other hand, may range from a simple cluster of PCs loosely coupled by a local area network to massive parallel single instruction, multiple data (SIMD) computers, which execute the same single instruction on all the processors.

The usual classification of parallel GAs found in the literature is [30, 31]:

- *Single-population master–slave GAs*: Conceptually, they are the same as the conventional GAs, with the difference that one master node executes the basic GA operations (selection, crossover, and mutation), and the fitness evaluations are distributed among several slave processors. The only gain in this type of parallel GA is in processing speed as the GA itself performs exactly in the same way as the sequential one.
- *Multiple-population GAs*: This type of GA consists of several subpopulations that exchange individuals occasionally. This operation is called *migration* and may be implemented according to several different strategies. Besides possible speedup if implemented in a parallel computer platform, this type of parallel GA may exhibit a better performance than the conventional GA as far as the quality of the solutions found. This type of parallel GA is also known as distributed GA because it is usually implemented in distributed-memory parallel computers.
- *Fine-grained GAs*: These consist of a single spatially structured population, usually in a two-dimensional rectangular grid, with one individual per grid point. The fitness evaluation is performed simultaneously for all the individuals, and selection and mating are usually restricted to a small neighborhood around each individual. This type of parallel GA is well suited for massively parallel multiple instruction, multiple data (MIMD) computers.

Other types of parallel GAs may be found in the literature but are less used than the above-mentioned or are a combination of them. Also, the parallel GAs may be implemented in a synchronous or asynchronous way.

Single-population master–slave GAs are easy to implement and usually present high speedup in inexpensive hardware like PC clusters. Multiple-population and fine-grained GAs, on the other hand, introduce new parameters such as the number of populations and their sizes, the topology of communications (e.g., each population is connected to all the others), and the migration rate. Although many implementations of parallel GAs have been described in the literature, the effect of these new parameters on the quality of the search is still under analysis [32].

2.9 FINAL COMMENTS

This tutorial on GAs has pointed out the main topics on their design. The focus on the essential topics helps one to not miss the forest for the trees. The first generation of GAs, based on the canonical algorithm, considering proportional selection and crossover/mutation with constant probabilities, was not originally proposed for solving static optimization problems [33]. Almost three decades of research has adapted the original proposal [34] to deal with this type of problem.

The first applications to power systems appeared after 1991 [35]. Since then, GAs have been applied not only to pure optimization problems in power systems but also

to model identification, control, and neural network training. After a necessary period of maturing, GAs are being used now, frequently in combination with conventional optimization techniques, for solving large-scale problems.

ACKNOWLEDGMENTS

This work was supported by the Brazilian Research Council (CNPq) and by the State of Rio de Janeiro Research Foundation (FAPERJ).

REFERENCES

1. Fogel DB. An introduction to simulated evolutionary optimization. IEEE Trans Neural Networks 1994; 5(1):3–14.
2. Fogel DB. Evolutionary computation—Toward a new philosophy of machine intelligence. Piscataway, NJ: IEEE Press; 1995.
3. Bäck T, Hammel U, Schwefel H-P. Evolutionary computation: Comments on the history and current state. IEEE Trans Evol Computat 1997; 1:3–17.
4. Goldberg DE. Genetic algorithms in search, optimization and machine learning. Reading, MA: Addison-Wesley; 1989.
5. Beasley D, Bull DR, Martin RR. An overview of genetic algorithms: Part 1, fundamentals. University Computing 1993; 15(2):58–69.
6. Beasley D, Bull DR, Martin RR. An overview of genetic algorithms: Part 2, research topics. University Computing 1993; 15(4):170–181.
7. Whitley D. A genetic algorithm tutorial. Statistics Computing 1994; 4:65–85.
8. Aarts E, Korst J. Simulated annealing and Boltzmann. New York: John Wiley & Sons; 1989.
9. Glover F, Laguna M. Tabu search. Boston: Kluwer Academic; 1997.
10. Kennedy J, Eberhart RC. Swarm intelligence. San Mateo, CA: Morgan Kaufmann; 2001.
11. Rudolph G. Convergence analysis of canonical genetic algorithms. IEEE Trans Neural Networks 1994; 5(1):96–101.
12. Vose MD, Wright AH. Form invariance and implicit parallelism. Evol Computat 2001; 9(3):355–370.
13. Antonisse HJ. A new interpretation of schema notation that overturns the binary encoding constraint. In: Proc. 3rd Int. Conf. on Genetic Algorithms. San Mateo, CA: Morgan Kaufmann; 1989. 86–91.
14. Goldberg DE. The theory of virtual alphabets. In: Parallel problem solving from nature 1. Lecture Notes in Computer Science, Vol. 496. New York: Springer; 1991. 13–22.
15. Eshelman LJ, Schaffer JD. Real-coded genetic algorithms and interval-schemata. In: Foundations of genetic algorithms 2. San Mateo, CA: Morgan Kaufmann; 1993. 187–202.
16. Radcliffe NJ, Surry PD. Fitness variance of formae and performance prediction. In: Foundations of genetic algorithms 3. San Mateo, CA: Morgan Kaufmann; 1995. 51–72.
17. Michalewicz Z, Schoenauer M. Evolutionary algorithms for constrained parameter optimization problems. Evol Computat 1996; 4(1):1–32.

18. Gen M, Cheng R. A survey of penalty techniques in genetic algorithms. In: Proc. 3rd IEEE Conf. on Evolutionary Computation. Piscataway, NJ: IEEE Press; 1996. 804–809.
19. Miller BL, Goldberg DE. Genetic algorithms, selection schemes, and the varying effects of noise. University of Illinois at Urbana-Champaign; IlliGAL Report, No. 95009. 1995.
20. Macready WG, Wolpert DH. Bandit problems and the exploration/exploitation tradeoff. IEEE Trans Evol Computat 1998; 2(1):2–22.
21. Goldberg DE, Deb K. A comparative analysis of selection schemes used in genetic algorithms. In: Foundations of genetic algorithms. San Mateo, CA: Morgan Kaufmann; 1991. 69–93.
22. Blickle T, Thiele L. A comparison of selection schemes used in genetic algorithms. Swiss Federal Institute of Technology; TIK-Report, Nr. 11, 2nd Version. 1995.
23. Bäck T. Selective pressure in evolutionary algorithms: A characterization of selection mechanisms. In: Proc. 1st IEEE Conf. on Evolutionary Computation. Piscataway, NJ: IEEE Press; 1994. 57–62.
24. Whitley LD. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In: Proc. 3rd Int. Conf. on Genetic Algorithms. San Mateo, CA: Morgan Kaufmann; 1989. 116–121.
25. Syswerda G. Uniform crossover in genetic algorithms. In: Proc. 3rd Int. Conf. on Genetic Algorithms. San Mateo, CA: Morgan Kaufmann; 1989. 2–9.
26. Saravanan N, Fogel DB, Nelson KM. A comparison of methods for self-adaptation in evolutionary algorithms. BioSystems 1995; 36:157–166.
27. Eiben AE. Hinterding R, Michalewicz Z. Parameter control in evolutionary algorithms. IEEE Trans Evol Computat 1999; 3(2):124–141.
28. Rogers A, Prügel-Bennet A. Genetic drift in genetic algorithm selection schemes. IEEE Trans Evol Computat 1999; 3(4):298–303.
29. Sareni B, Krähenbühl L. Fitness sharing and niching methods revisited. IEEE Trans Evol Computat 1998; 2(3):97–106.
30. Cantú-Paz E. Efficient and accurate genetic algorithms. Boston: Kluwer; 2000.
31. Alba E, Troya JM. A survey of parallel distributed genetic algorithms. Complexity 1999; 4(4):31–52.
32. Cantu-Paz E. Markov chain models of parallel genetic algorithms. IEEE Trans Evol Computat 2000; 4(3):216–226.
33. De Jong KA. Genetic algorithms are NOT function optimizers. In: Foundations of genetic algorithms 2. San Mateo, CA: Morgan Kaufmann; 1993. 5–17.
34. Holland JH. Adaptation in natural and artificial systems. University of Michigan Press; Ann Arbor, MI; 1975.
35. Miranda V, Srinivasan D, Proença LM. Evolutionary computation in power systems. Electrical Power & Energy Systems 1998; 20(2):89–98.