13

GENETIC ALGORITHMS

Chapter Objectives

- Identify effective algorithms for approximate solutions of optimization problems described with large data sets.
- Compare basic principles and concepts of natural evolution and simulated evolution expressed through genetic algorithms (GAs).
- Describe the main steps of a genetic algorithm with illustrative examples.
- Explain standard and nonstandard genetic operators such as a mechanism for improving solutions.
- Discuss a schema concept with "don't care" values and its application to approximate optimization.
- Apply a GA to the traveling-salesman problem (TSP) and optimization of classification rules as examples of hard optimizations.

There is a large class of interesting problems for which no reasonably fast algorithms have been developed. Many of these problems are optimization problems that arise frequently in applications. The fundamental approach to optimization is to

Data Mining: Concepts, Models, Methods, and Algorithms, Second Edition. Mehmed Kantardzic.

formulate a single standard of measurement—a cost function—that summarizes the performance or value of a decision and iteratively improves this performance by selecting from among the available alternatives. Most classical methods of optimization generate a deterministic sequence of trial solutions based on the gradient or higher order statistics of the cost function. In general, any abstract task to be accomplished can be thought of as solving a problem, which can be perceived as a search through a space of potential solutions. Since we are looking for "the best" solution, we can view this task as an optimization process. For small data spaces, classical, exhaustive search methods usually suffice; for large spaces, special techniques must be employed. Under regular conditions, the techniques can be shown to generate sequences that asymptotically converge to optimal solutions, and in certain cases they converge exponentially fast. But the methods often fail to perform adequately when random perturbations are imposed on the function that is optimized. Further, locally optimal solutions often prove insufficient in real-world situations. Despite such problems, which we call *hard-optimization problems*, it is often possible to find an effective algorithm whose solution is approximately optimal. One of the approaches is based on GAs, which are developed on the principles of natural evolution.

Natural evolution is a population-based optimization process. Simulating this process on a computer results in stochastic-optimization techniques that can often outperform classical methods of optimization when applied to difficult, real-world problems. The problems that the biological species have solved are typified by chaos, chance, temporality, and nonlinear interactivity. These are the characteristics of the problems that have proved to be especially intractable to classical methods of optimization. Therefore, the main avenue of research in simulated evolution is a GA, which is a new, iterative, optimization method that emphasizes some facets of natural evolution. GAs approximate an optimal solution to the problem at hand; they are by nature stochastic algorithms whose search methods model natural phenomena such as genetic inheritance and the Darwinian strife for survival.

13.1 FUNDAMENTALS OF GAs

GAs are derivative-free, stochastic-optimization methods based loosely on the concepts of natural selection and evolutionary processes. They were first proposed and investigated by John Holland at the University of Michigan in 1975. The basic idea of GAs was revealed by a number of biologists when they used computers to perform simulations of natural genetic systems. In these systems, one or more chromosomes combine to form the total genetic prescription for the construction and operation of some organisms. The chromosomes are composed of genes, which may take a number of values called allele values. The position of a gene (its locus) is identified separately from the gene's function. Thus, we can talk of a particular gene, for example, an animal's eye-color gene, with its locus at position 10 and its allele value as blue eyes.

Before going into details of the applications of GAs in the following sections, let us understand their basic principles and components. GAs encode each point in a parameter or solution space into a binary-bit string called a chromosome. These points

TABLE 13.1. Basic Concepts in Genetic Algorithms

Concept in Natural Evolution	Concept in Genetic Algorithms
Chromosome	String
Gene	Features in the string
Locus	Position in the string
Allele	Position value (usually 0 or 1)
Genotype	String structure
Phenotype	Set of characteristics (features)

in an n-dimensional space do not represent samples in the terms that we defined them at the beginning of this book. While samples in other data-mining methodologies are data sets given in advance for training and testing, sets of n-dimensional points in GAs are a part of a GA, and they are produced iteratively in the optimization process. Each point or binary string represents a potential solution to the problem that is to be solved. In GAs, the decision variables of an optimization problem are coded by a structure of one or more strings, which are analogous to chromosomes in natural genetic systems. The coding strings are composed of features that are analogous to genes. Features are located in different positions in the string, where each feature has its own position (locus) and a definite allele value, which complies with the proposed coding method. The string structures in the chromosomes go through different operations similar to the natural-evolution process to produce better alternative solutions. The quality of new chromosomes is estimated based on the "fitness" value, which can be considered as the objective function for the optimization problem. The basic relations between concepts in natural evolution and GAs are given in Table 13.1. Instead of single a point, GAs usually keep a set of points as a population, which is then evolved repeatedly toward a better overall fitness value. In each generation, the GA constructs a new population using genetic operators such as crossover and mutation. Members with higher fitness values are more likely to survive and participate in mating or *crossover operations*.

As a general-purpose optimization tool, GAs are moving out of academia and finding significant applications in many other venues. Typical situations where GAs are particularly useful are in difficult optimization cases for which analytical methods do not work well. GAs have been quite successfully applied to optimization problems like wire routing, scheduling, adaptive control, game playing, transportation problems, TSPs, database query optimization, and machine learning. In the last decades, the significance of optimization has grown even further because many important large-scale, combinatorial-optimization problems and highly constrained engineering problems can only be solved approximately. GAs aim at such complex problems. They belong to the class of probabilistic algorithms, yet they are very different from random algorithms as they combine elements of directed and stochastic search. Another important property of genetic-based search methods is that they maintain a population of potential solutions while all other methods process a single point of the search space. Because of these characteristics, GAs are more robust than existing directed-search methods.

GAs are popular because they do not depend on functional derivatives, and they have the following characteristics:

1. GAs are parallel-search procedures that can be implemented on parallel-processing machines to massively speed up their operations.
2. GAs are applicable to both continuous- and discrete-optimization problems.
3. GAs are stochastic and less likely to get trapped in local minima, which inevitably are present in any practical optimization application.
4. GAs' flexibility facilitates both structure and parameter identification in complex models.

The GA theory provides some explanations of why, for a given problem formulation, we may obtain convergence to the sought optimal point. Unfortunately, practical applications do not always follow the theory, the main reasons being:

1. the coding of the problem often moves the GA to operate in a different space than that of the problem itself;
2. there are practical limits on the hypothetically unlimited number of iterations (generations in the GA); and
3. there is a limit on the hypothetically unlimited population size.

One of the implications of these observations is the inability of GAs, under certain conditions, to find the optimal solution or even an approximation to the optimal solution; such failures are usually caused by premature convergence to a local optimum. Do not forget that this problem is common not only for the other optimization algorithms but also for the other data-mining techniques.

13.2 OPTIMIZATION USING GAs

Let us note first that without any loss of generality we can assume that all optimization problems can be analyzed as maximization problems only. If the optimization problem is to minimize a function f(x), this is equivalent to maximizing a function g(x) = −f(x). Moreover, we may assume that the objective function f(x) takes positive values in its domain. Otherwise, we can translate the function for some positive constant C so that it will be always positive, that is,

$$\max f^*(x) = \max \{f(x) + C\}$$

If each variable x_i, with real values, is coded as a binary string of length *m*, then the relation between the initial value and the coded information is

$$x_i = a + \text{decimal}\,(\text{binary-string}_i)\,\{[b-a]/[2^m - 1]\}$$

where the variable x_i can take the values from a domain $D_i = [a, b]$, and m is the smallest integer such that the binary code has the required precision. For example, the value for variable x given on the domain [10, 20] is a binary-coded string with the length equal to 3 and the code 100. While the range of codes is between 000 and 111, the question is: What is the real value of the coded variable x? For this example, m = 3 and the corresponding precision is

$$[b - a]/[2^m - 1] = (20 - 10)/(2^3 - 1) = 10/7 = 1.42$$

and that is the difference between two successive x_i values that could be tested as candidates for extreme. Finally, the attribute with the code 100 has a decimal value

$$x = 10 + \text{decimal}\ (100) \cdot 1.42 = 10 + 4 \cdot 1.42 = 15.68$$

Each chromosome as a potential solution is represented by a concatenation of binary codes for all features in the problem to be optimized. Its total length m is a sum of the features' code lengths m_i:

$$m = \sum_{i=1}^{k} m_i$$

where k is the number of features or input variables for the problem at hand. When we introduce these basic principles of a code construction, it is possible to explain the main steps of a GA.

13.2.1 Encoding Schemes and Initialization

A GA starts with designing a representation of a solution for the given problem. A solution here means any value that is a candidate for a correct solution that can be evaluated. For example, suppose we want to maximize function $y = 5 - (x - 1)^2$. Then $x = 2$ is a solution, $x = 2.5$ is another solution, and $x = 3$ is the correct solution of the problem that maximizes y. The representation of each solution for a GA is up to the designer. It depends on what each solution looks like and which solution form will be convenient for applying a GA. The most common representation of a solution is as a string of characters, that is, a string of codes for feature representation, where the characters belong to a fixed alphabet. The larger the alphabet is, the more the information that can be represented by each character in the string is. Therefore, fewer elements in a string are necessary to encode specific amounts of information. However, in most real-world applications, GAs usually use a binary-coding schema.

The encoding process transforms points in a feature space into a bit-string representation. For instance, a point (11, 6, 9) in a three-dimensional (3-D) feature space, with ranges [0, 15] for each dimension, can be represented as a concatenated binary string:

$$(11, 6, 9) \Rightarrow (101101101001)$$

in which each feature's decimal value is encoded as a gene composed of four bits using a binary coding.

Other encoding schemes, such as Gray coding, can also be used and, when necessary, arrangements can be made for encoding negative, floating-point, or discrete-value numbers. Encoding schemes provide a way of translating problem-specific knowledge directly into the GA framework. This process plays a key role in determining GAs' performances. Moreover, genetic operators can and should be designed along with the encoding scheme used for a specific application.

A set of all feature values encoded into a bit string represents one chromosome. In GAs we are manipulating not a single chromosome but a set of chromosomes called a population. To initialize a population, we can simply set some *pop-size* number of chromosomes randomly. The size of the population is also one of the most important choices faced by any user of GAs and may be critical in many applications: Will we reach the approximate solution at all, and if yes, how fast? If the population size is too small, the GA may converge too quickly and maybe to a solution that is only the local optimum; if it is too large, the GA may waste computational resources and the waiting time for an improvement might be too long.

13.2.2 Fitness Evaluation

The next step, after creating a population, is to calculate the fitness value of each member in the population because each chromosome is a candidate for an optimal solution. For a maximization problem, the fitness value f_i of the *i*th member is usually the objective function evaluated at this member (or the point in parameter space). The fitness of a solution is a measure that can be used to compare solutions to determine which is better. The fitness values may be determined from complex analytical formulas, simulation models, or by referring to observations from experiments or real-life problem settings. GAs will work correctly if fitness values are determined appropriately, keeping in mind that a selection of the objective function is highly subjective and problem-dependent.

We usually need fitness values that are positive, so some kind of scaling and/or translation of data may become necessary if the objective function is not strictly positive. Another approach is to use the rankings of members in a population as their fitness values. The advantage of this approach is that the objective function does not need to be accurate, as long as it can provide the correct ranking information.

13.2.3 Selection

In this phase, we have to create a new population from the current generation. The selection operation determines which parent chromosomes participate in producing offspring for the next generation. Usually, members are selected for mating with a selection probability proportional to their fitness values. The most common way to implement this method is to set the selection probability p equal to

$$p_i = f_i \Big/ \sum_{k=1}^{n} f_k$$

where n is the population size and f_i is a fitness value for the *i*th chromosome. The effect of this selection method is to allow members with above-average values to reproduce and replace members with below-average fitness values.

For the selection process (selection of a new population with respect to the probability distribution based on fitness values), a roulette wheel with slots sized according to fitness for each chromosome is used. We construct such a roulette wheel as follows:

1. Calculate the fitness value $f(v_i)$ for each chromosome v_i.
2. Find the total fitness of the population:

$$F = \sum_{i=1}^{pop\text{-}size} f(v_i)$$

3. Calculate the probability of a selection p_i for each chromosome v_i:

$$p_i = f(v_i)/F$$

4. Calculate a cumulative probability q_i after each chromosome v_i is included:

$$q_i = \sum_{j=1}^{i} p_j$$

where q increases from 0 to maximum 1. Value 1 shows that all chromosomes from the population are included into a cumulative probability.

The selection process is based on spinning the roulette wheel *pop-size* times. Each time, we select a single chromosome for a new population. An implementation could repeat steps 1 and 2 *pop-size* times:

1. Generate a random number r from the range [0, 1].
2. If $r < q_1$, then select the first chromosome v_1; otherwise, select the *i*th chromosome v_i such that $q_{i-1} < r \leq q_i$.

Obviously, some chromosomes would be selected more than once. That is in accordance with the theory. The GA performs a multidirectional search by maintaining a population of potential solutions and encourages good solutions. The population undergoes a simulated evolution—in each generation the relatively "good" solutions reproduce while the relatively "bad" solutions die. To distinguish between different solutions, we use an objective or evaluation function, which plays the role of an environment.

13.2.4 Crossover

The strength of GAs arises from the structured information exchange of crossover combinations of highly fit individuals. So what we need is a crossover-like operator that would exploit important similarities between chromosomes. The probability of crossover (PC) is the parameter that will define the expected number of chromosomes—PC · *pop-size*—which undergo the crossover operation. We define the chromosomes for crossover in a current population using the following iterative procedure. Steps 1 and 2 have to be repeated for all chromosomes:

1. Generate a random number r from the range [0, 1].
2. If r < PC, select the given chromosome for crossover.

If PC is set to 1, all chromosomes in the population will be included into the crossover operation; if PC = 0.5, only half of the population will perform crossover and the other half will be included into a new population directly without changes.

To exploit the potential of the current gene pool, we use crossover operators to generate new chromosomes that will retain the good features from the previous generation. Crossover is usually applied to selected pairs of parents.

One-point crossover is the most basic crossover operator, where a crossover point on the genetic code is selected at random, and two parent chromosomes are interchanged at this point. In *two-point crossover*, two points are selected, and a part of chromosome string between these two points is then swapped to generate two children of the new generation. Examples of one- and two-point crossover are shown in Figure 13.1.

We can define an n-point crossover similarly, where the parts of strings between points 1 and 2, 3 and 4, and finally n-1 and n are swapped. The effect of crossover is similar to that of mating in the natural evolutionary process in which parents pass segments of their own chromosomes on to their children. Therefore, some children are able to outperform their parents if they get "good" genes or genetic traits from their parents.

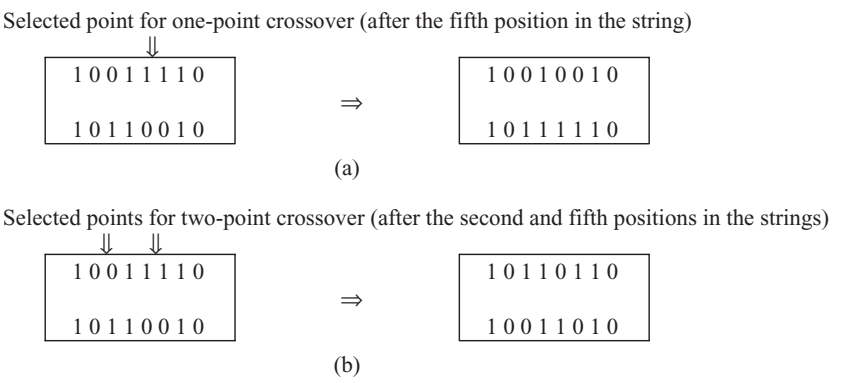

Figure 13.1. Crossover operators. (a) One-point crossover; (b) two point crossover.

13.2.5 Mutation

Crossover exploits existing gene potentials, but if the population does not contain all the encoded information needed to solve a particular problem, no amount of gene mixing can produce a satisfactory solution. For this reason, a mutation operator capable of spontaneously generating new chromosomes is included. The most common way of implementing mutation is to flip a bit with a probability equal to a very low given mutation rate (MR). A mutation operator can prevent any single bit from converging to a value through the entire population, and, more important, it can prevent the population from converging and stagnating at any local optima. The MR is usually kept low so good chromosomes obtained from crossover are not lost. If the MR is high (e.g., above 0.1), GA performance will approach that of a primitive random search. Figure 13.2 provides an example of mutation.

In the natural evolutionary process, selection, crossover, and mutation all occur simultaneously to generate offspring. Here we split them into consecutive phases to facilitate implementation of and experimentation with GAs. Note that this section only gives a general description of the basics of GAs. Detailed implementations of GAs vary considerably, but the main phases and the iterative process remain.

At the end of this section we can summarize that the major components of GA include encoding schemata, fitness evaluation, parent selection, and application of crossover operators and mutation operators. These phases are performed iteratively, as presented in Figure 13.3.

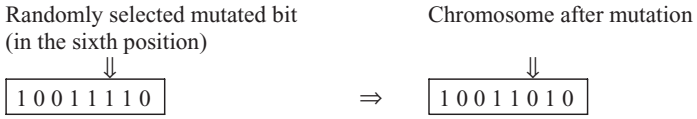

Figure 13.2. Mutation operator.

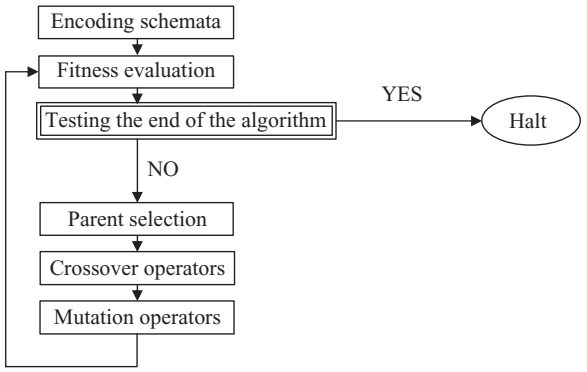

Figure 13.3. Major phases of a genetic algorithm.

It is relatively easy to keep track of the best individual chromosomes in the evolution process. It is customary in GA implementations to store "the best ever" individual at a separate location. In that way, the algorithm would report the best value found during the whole process, just in the final population.

Optimization under constraints is also the class of problems for which GAs are an appropriate solution. The constraint-handling techniques for genetic algorithms can be grouped into different categories. One typical way of dealing with GA candidates that violate the constraints is to generate potential solutions without considering the constraints, and then to penalize them by decreasing the "goodness" of the evaluation function. In other words, a constrained problem is transformed to an unconstrained one by associating a penalty with all constraint violations. These penalties are included in the function evaluation, and there are different kinds of implementations. Some penalty functions assign a constant as a penalty measure. Other penalty functions depend on the degree of violation: the larger the violation, the greater the penalty. The growth of the penalty function can be logarithmic, linear, quadratic, exponential, and so on, depending upon the size of the violation. Several implementations of GA optimization under constraints are given in the texts recommended for further study (Section 13.9).

13.3 A SIMPLE ILLUSTRATION OF A GA

To apply a GA for a particular problem, we have to define or to select the following five components:

1. a genetic representation or encoding schema for potential solutions to the problem;
2. a way to create an initial population of potential solutions;
3. an evaluation function that plays the role of the environment, rating solutions in terms of their "fitness";
4. Genetic operators that alter the composition of the offspring; and
5. values for the various parameters that the GA uses (population size, rate of applied operators, etc.).

We discuss the main features of GAs by presenting a simple example. Suppose that the problem is the optimization of a simple function of one variable. The function is defined as

$$f(x) = x^2$$

The task is to find x from the range [0,31], which maximizes the function f(x). We selected this problem because it is relatively easy to analyze optimization of the function f(x) analytically, to compare the results of the analytic optimization with a GA, and to find the approximate optimal solution.

13.3.1 Representation

The first step in the GA is to represent the solution alternative (a value for the input feature) in a coded-string format. Typically, the string is a series of features with their values; each feature's value can be coded with one from a set of discrete values called the allele set. The allele set is defined according to the needs of the problem, and finding the appropriate coding method is a part of the art of using GAs. The coding method must be minimal but completely expressive. We will use a binary vector as a chromosome to represent real values of the single variable x. The length of the vector depends on the required precision, which, in this example, is selected as 1. Therefore, we need a minimum five-bit code (string) to accommodate the range with required precision:

$$(b - a)/(2^m - 1) \leq \text{Required precision}$$

$$(31 - 0)/(2^m - 1) \leq 1$$

$$2^m \geq 32$$

$$m \geq 5$$

For this example, the mapping from a real number to a binary code is defined by the relation (because a = 0):

$$\text{Code} = \text{binary}\,(x_{\text{decimal}})$$

Opposite mapping, from the binary code to the real value of the argument, is also unique:

$$x = \text{decimal}\,(\text{Code}_{\text{binary}})$$

and it will be used only for checking the intermediate results of optimization. For example, if we want to transform the value x = 11 into a binary string, the corresponding code will be 01011. On the other hand, code 11001 represents the decimal value x = 25.

13.3.2 Initial Population

The initialization process is very simple: We randomly create a population of chromosomes (binary codes) with the given length. Suppose that we decide that the parameter for the number of strings in the population is equal to four. Then one possible randomly selected population of chromosomes is

$$CR_1 = 01101$$

$$CR_2 = 11000$$

$$CR_3 = 01000$$

$$CR_4 = 10011$$

13.3.3 Evaluation

The evaluation function for binary vectors representing chromosomes is equivalent to the initial function f(x) where the given chromosome represents the binary code for the real value x. As noted earlier, the evaluation function plays the role of the environment, rating potential solutions in terms of their fitness. For our example, four chromosomes CR_1 to CR_4 correspond to values for input variable x:

$$x_1(CR_1) = 13$$
$$x_2(CR_2) = 24$$
$$x_3(CR_3) = 8$$
$$x_4(CR_4) = 19.$$

Consequently, the evaluation function would rate them as follows:

$$f(x_1) = 169$$
$$f(x_2) = 576$$
$$f(x_3) = 64$$
$$f(x_4) = 361$$

The results of evaluating the chromosomes initially generated may be given in a tabular form, and they are represented in Table 13.2. The expected reproduction column shows "the evaluated quality" of chromosomes in the initial population. Chromosomes CR_2 and CR_4 are more likely to be reproduced in the next generation than CR_1 and CR_3.

13.3.4 Alternation

In the alternation phase, the new population is selected based on the population evaluated in the previous iteration. Clearly, the chromosome CR_4 in our example is the best

TABLE 13.2. Evaluation of the Initial Population

CR_i	Code	x	f(x)	$f(x)/\sum f(x)$	Expected Reproduction: f(x)/fav
1	01101	13	169	0.14	0.58
2	11000	24	576	0.49	1.97
3	01000	8	64	0.06	0.22
4	10011	19	361	0.31	1.23
$\sum$			1170	1.00	4.00
Average			293	0.25	1.00
Max			576	0.49	1.97

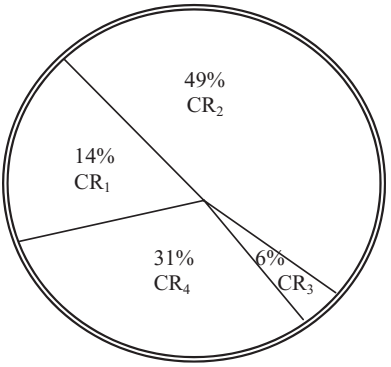

Figure 13.4. Roulette wheel for selection of the next population.

of the four chromosomes, since its evaluation returns the highest value. In the alternation phase, an individual may be selected depending on its objective-function value or fitness value. For maximization problems, the higher the individual's fitness is, the more probable that it can be selected for the next generation. There are different schemes that can be used in the selection process. In the simple GA we proposed earlier, the roulette wheel selection technique, an individual is selected randomly depending on a computed probability of selection for each individual. The probability of selection is computed by dividing the individual's fitness value by the sum of fitness values of the corresponding population, and these values are represented in column 5 of Table 13.2.

In the next step we design the roulette wheel, which is, for our problem, graphically represented in Figure 13.4.

Using the roulette wheel, we can select chromosomes for the next population. Suppose that the randomly selected chromosomes for the next generation are CR_1, CR_2, CR_2, CR_4 (the selection is in accordance with the expected reproduction—column 6 in Table 13.2). In the next step, these four chromosomes undergo the genetic operations: crossover and mutation.

13.3.5 Genetic Operators

Crossover is not necessarily applied to all pairs of selected individuals. A choice is made depending on a specified probability called PC, which is typically between 0.5 and 1. If crossover is not applied (PC = 0), the offspring are simply a duplication of the parents. For the process of crossover it is necessary to determine the percentage of the population that will perform the crossover. For our particular problem we use the following parameters of the GA:

1. population size, *pop-size* = 4 (the parameter was already used).
2. probability of crossover, PC = 1.
3. probability of mutation, PM = 0.001 (the parameter will be used in a mutation operation).

A value of 1 for the probability of crossover translates into a 100% crossover—all chromosomes will be included in the crossover operation.

The second set of parameters in this phase of a GA is the random selection of parents for crossover and positions in the strings where the crossover will be performed. Suppose that these are randomly selected pairs: CR_1–CR_2 and CR_2–CR_4, and crossover is after the third position in the strings for both pairs. Then the selected strings

$$\begin{aligned} \text{First pair} \quad & CR_1 = 01101 \\ & CR_2 = 11000 \\ & \Uparrow \\ \text{Second pair} \quad & CR_2 = 11000 \\ & CR_4 = 10011 \\ & \Uparrow \end{aligned}$$

will become, after crossover, a new population:

$$\begin{aligned} CR_1' &= 01100 \\ CR_2' &= 11001 \\ CR_3' &= 11011 \\ CR_4' &= 10000 \end{aligned}$$

The second operator that can be applied in every iteration of a GA is mutation. For our example, the mutation operator has a probability of 0.1%, which means that in the 1000 transformed bits, a mutation will be performed only once. Because we transformed only 20 bits (one population of 4×5 bits is transformed into another), the probability that a mutation will occur is very small. Therefore, we can assume that the strings CR_1' to CR_4' will stay unchanged with respect to a mutation operation in this first iteration. It is expected that only one bit will be changed for every 50 iterations.

That was the final processing step in the first iteration of the GA. The results, in the form of the new population CR_1' to CR_4', are used in the next iteration, which starts again with an evaluation process.

13.3.6 Evaluation (Second Iteration)

The process of evaluation is repeated in the new population. These results are given in Table 13.3.

The process of optimization with additional iterations of the GA can be continued in accordance with Figure 13.3. We will stop here with a presentation of computational steps for our example, and give some additional analyses of results useful for a deeper understanding of a GA.

TABLE 13.3. Evaluation of the Second Generation of Chromosomes

CR_i	Code	x	f(x)	f(x)/∑f(x)	Expected Reproduction: f(x)/fav
1	01100	12	144	0.08	0.32
2	11001	25	625	0.36	1.44
3	11011	27	729	0.42	1.68
4	10000	16	256	0.14	0.56
∑			1754	1.00	4.00
Average			439	0.25	1.00
Max			729	0.42	1.68

Although the search techniques used in the GA are based on many random parameters, they are capable of achieving a better solution by exploiting the best alternatives in each population. A comparison of sums, and average and max values from Tables 13.2 and 13.3

$$\sum_1 = 1170 \Rightarrow \sum_2 = 1754$$

$$\text{Average}_1 = 293 \Rightarrow \text{Average}_2 = 439$$

$$\text{Max}_1 = 576 \Rightarrow \text{Max}_2 = 729$$

shows that the new, second population is approaching closer to the maximum of the function f(x). The best result obtained from the evaluation of chromosomes in the first two iterations is the chromosome CR3′ = 11011 and it corresponds to the feature's value x = 27 (theoretically it is known that the maximum of f(x) is for x = 31, where f(x) reaches the value 961). This increase will not be obtained in each GA iteration, but on average, the final population is much closer to a solution after a large number of iterations. The number of iterations is one possible stopping criterion for the GA algorithm. The other possibilities for stopping the GA are when the difference between the sums in two successive iterations is less than the given threshold, when a suitable fitness value is achieved, and when the computation time is limited.

13.4 SCHEMATA

The theoretical foundations of GAs rely on a binary-string representation of solutions, and on the notation of a *schema*—a template allowing exploration of similarities among chromosomes. To introduce the concept of a schema, we have to first formalize some related terms. The search space Ω is the complete set of possible chromosomes or strings. In a fixed-length string l, where each bit (gene) can take on a value in the alphabet A of size k, the resulting size of the search space is k^l. For example, in binary-coded strings where the length of the string is 8, the size of the search space is $2^8 = 256$. A string in the population S is denoted by a vector $x \in \Omega$. So, in the previously

described example, x would be an element of $\{0, 1\}^8$. A schema is a similarity template that defines a subset of strings with fixed values in certain positions.

A schema is built by introducing a *don't care* symbol (*) into the alphabet of genes. Each position in the scheme can take on the values of the alphabet (fixed positions) or a "don't care" symbol. In the binary case, for example, the schemata of the length l are defined as $H \in \{0, 1, *\}^l$. A schema represents all the strings that match it on all positions other than "*". In other words, a schema defines a subset of the search space, or a hyperplane partition of this search space. For example, let us consider the strings and schemata of the length ten. The schema

$$(*\,1\,1\,1\,1\,0\,0\,1\,0\,0)$$

matches two strings

$$\{(0\,1\,1\,1\,1\,0\,0\,1\,0\,0), (1\,1\,1\,1\,1\,0\,0\,1\,0\,0)\},$$

and the schema

$$(*\,1\,*\,1\,1\,0\,0\,1\,0\,0)$$

matches four strings

$$\{(0\,1\,0\,1\,1\,0\,0\,1\,0\,0), (0\,1\,1\,1\,1\,0\,0\,1\,0\,0), (1\,1\,0\,1\,1\,0\,0\,1\,0\,0), (1\,1\,1\,1\,1\,0\,0\,1\,0\,0)\}.$$

Of course, the schema

$$(1\,0\,0\,1\,1\,1\,0\,0\,0\,1)$$

represents one string only, and the schema

$$(*\,*\,*\,*\,*\,*\,*\,*\,*\,*)$$

represents all strings of length 10. In general, the total number of possible schemata is $(k + 1)^l$, where k is the number of symbols in the alphabet and l is the length of the string. In the binary example of coding strings, with a length of 10, it is $(2+1)^{10} = 3^{10} = 59049$ different strings. It is clear that every binary schema matches exactly 2^r strings, where r is the number of *don't care* symbols in a schema template. On the other hand, each string of length m is matched by 2^m different schemata.

We can graphically illustrate the representation of different schemata for five-bit codes used to optimize the function $f(x) = x^2$ on interval [0, 31]. Every schema represents a subspace in the 2-D space of the problem. For example, the schema 1**** reduces the search space of the solutions on the subspace given in Figure 13.5a, and the schema 1*0** has a corresponding search space in Figure 13.5b.

Different schemata have different characteristics. There are three important schema properties: *order* (O), *length* (L), and *fitness* (F). The *order* of the schema S denoted

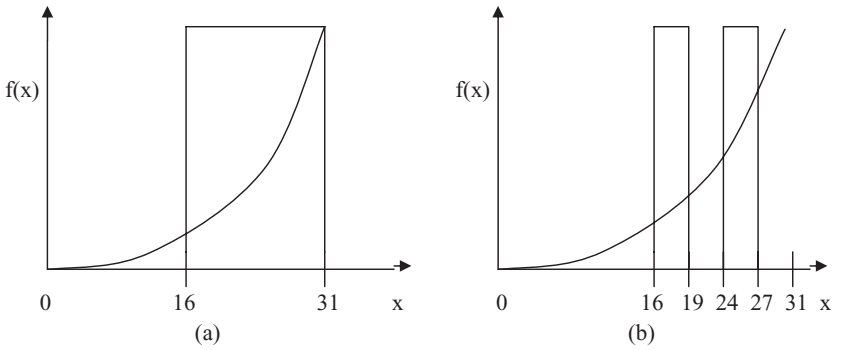

Figure 13.5. f(x) = x^2: Search spaces for different schemata. (a) Schema 1****; (b) Schema 1*0**.

by O(S) is the number of 0 and 1 positions, that is, fixed positions presented in the schema. A computation of the parameter is very simple: It is the length of the template minus the number of *don't care* symbols. For example, the following three schemata, each of length 10

$$S_1 = (* * * 0\ 0\ 1 * 1\ 1\ 0)$$

$$S_2 = (* * * * * 0 * * 0 *)$$

$$S_3 = (1\ 1\ 1\ 0\ 1 * * 0\ 0\ 1)$$

have the following orders:

$$O(S_1) = 10 - 4 = 6, \quad O(S_2) = 10 - 8 = 2, \quad O(S_3) = 10 - 2 = 8$$

The schema S_3 is the most specific one and the schema S_2 is the most general one. The notation of the order of a schema is useful in calculating survival probabilities of the schema for mutations.

The *length* of the schema S, denoted by L(S), is the distance between the first and the last fixed-string positions. It defines the compactness of information contained in a schema. For example, the values of this parameter for the given schemata S_1 to S_3 are

$$L(S_1) = 10 - 4 = 6, \quad L(S_2) = 9 - 6 = 3, \quad L(S_3) = 10 - 1 = 9.$$

Note that the schema with a single fixed position has a length of 0. The length L of a schema is a useful parameter in calculating the survival probabilities of the schema for crossover.

Another property of a schema S is its *fitness* F(S, t) at time t (i.e., for the given population). It is defined as the average fitness of all strings in the population matched by the schema S. Assume there are p strings $\{v_1, v_2, \ldots, v_p\}$ in a population matched by a schema S at the time t. Then

$$F(S,t) = \left[\sum_{i=1}^{p} f(v_i) \right] \Big/ p$$

The fundamental theorem of schema construction given in this book without proof explains that the *short* (high O), *low-order* (low L), and *above-average* schemata (high F) receive an exponentially increasing number of strings in the next generations of a GA. An immediate result of this theorem is that GAs explore the search space by short, low-order schemata that subsequently are used for information exchange during crossover and mutation operations. Therefore, a GA seeks near-optimal performance through the analysis of these schemata, called the *building blocks*. Note, however, that the building-blocks approach is just a question of empirical results without any proof, and these rules for some real-world problems are easily violated.

13.5 TSP

In this section, we explain how a GA can be used to approach the TSP. Simply stated, the traveling salesman must visit every city in his territory exactly once and then return to the starting point. Given the cost of travel between all the cities, how should he plan his itinerary at a minimum total cost for the entire tour? The TSP is a problem in combinatorial optimization and arises in numerous applications. There are several branch-and-bound algorithms, approximate algorithms, and heuristic search algorithms that approach this problem. In the last few years, there have been several attempts to approximate the TSP solution using GA.

The TSP description is based on a graph representation of data. The problem could be formalized as: Given an undirected weighted graph, find the shortest route, that is, a shortest path in which every vertex is visited exactly once, except that the initial and terminal vertices are the same. Figure 13.6 shows an example of such a graph and its

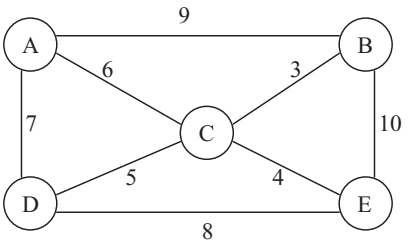

Figure 13.6. Graphical representation of the traveling-salesman problem with a corresponding optimal solution.

optimal solution. A, B, C, and so on, are the cities that were visited, and the numbers associated with the edges are the cost of travel between the cities.

It is natural to represent each solution of the problem, even if it is not optimal, as a permutation of the cities. The terminal city can be omitted in the representation since it should always be the same as the initial city. For the computation of the total distance of each tour, the terminal city must be counted.

By representing each solution as a permutation of the cities, each city will be visited exactly once. Not every permutation, however, represents a valid solution, since some cities are not directly connected (e.g., A and E in Fig. 10.6). One practical approach is to assign an artificially large distance between cities that are not directly connected. In this way, invalid solutions that contain consecutive, nonadjacent cities will disappear, and all solutions will be allowed.

Our objective here is to minimize the total distance of each tour. We can select different fitness functions that will reflect this objective. For example, if the total distance is s, then f(s) could be a simple $f(s) = s$ if we minimize the fitness function; alternatives for the maximization of a fitness function are $f(s) = 1/s$, $f(s) = 1/s^2$, $f(s) = K - s$, where K is a positive constant that makes $f(s) \geq 0$. There is no general formula to design the best fitness function. But, when we do not adequately reflect the goodness of the solution in the fitness function, finding an optimal or near-optimal solution will not be effective.

When dealing with permutations as solutions, simple crossover operations will result in invalid solutions. For example, for the problem in Figure 10.6, a crossover of two solutions after the third position in the strings

$$\Downarrow$$

A D E B C

A E C D B

$$\Uparrow$$

will produce new strings

A D E D B

A E C B C

which are invalid solutions because they do not represent permutations of initial elements in the strings. To avoid this problem, a modified crossover operation is introduced that directly operates on permutations and still gives permutations. This is a *partially matched crossover* (PMC) operation. It can be used not only for the TSPs, but also for any other problems that involve permutations in a solution's representation. We illustrate the effects of the PMC operation by an example. Assume that two solutions are given as permutations of the same symbols, and suppose that the PMC is a two-point operation. Selecting two strings and two random crossing points is the first step in the process of applying the PMC operation.

⇓⇓

A D E B C

A E C D B

⇑⇑

The substrings between crossing points are called matching sections. In our example, we have two elements in the matching sections: E B for the first string and C D for the second one. The crossover operation requires an exchange of the symbols E with C, denoted as an ordered pair (E, C), and B with D, represented as (B, D). The next step in the PMC operation is to permute each of these two-element permutations in each string. In other words, it is necessary to exchange the places for pairs (E, C) and (B, D) in both strings. The result of (E, C) changes in the first string is A D C B E, and after the second pair (B, D) has been changed, the final version of the first string is A B C D E. The second string after application of the same permutations will become A C E D B first, and then A C E B D finally. If we analyze the two strings obtained by PMC operation

⇓⇓

A B C D E

A C E B D

⇑⇑

we can see that middle parts of the strings were really exchanged as in a standard crossover operation. On the other hand, the two new strings remain as valid permutations since the symbols are actually permuted within each string.

The other steps of a GA applied to the TSP are unchanged. A GA based on the above operator outperforms a random search for TSP, but still leaves much room for improvement. Typical results from the algorithm, when applied to 100 randomly generated cities gave after 20,000 generations, a value of the whole tour 9.4% above minimum.

13.6 MACHINE LEARNING USING GAs

Optimization problems are one of the most common application categories of GAs. In general, an optimization problem attempts to determine a solution that, for example, maximizes the profit in an organization or minimizes the cost of production by determining values for selected features of the production process. Another typical area where GAs are applied is the discovery of input-to-output mapping for a given, usually complex, system, which is the type of problem that all machine-learning algorithms are trying to solve.

The basic idea of input-to-output mapping is to come up with an appropriate form of a function or a model, which is typically simpler than the mapping given originally—usually represented through a set of input–output samples. We believe that a function best describes this mapping. Measures of the term "best" depend on the specific application. Common measures are accuracy of the function, its robustness, and its computational efficiency. Generally, determining a function that satisfies all these criteria is not necessarily an easy task; therefore, a GA determines a "good" function, which can then be used successfully in applications such as pattern classification, control, and prediction. The process of mapping may be automated, and this automation, using GA technology, represents another approach to the formation of a model for inductive machine learning.

In the previous chapters of this book we described different algorithms for machine learning. Developing a new model (input-to-output abstract relation) based on some given set of samples is an idea that can also be implemented in the domain of GAs. There are several approaches to GA-based learning. We will explain the principles of the technique that is based on schemata and the possibilities of its application to classification problems.

Let us consider a simple database, which will be used as an example throughout this section. Suppose that the training or learning data set is described with a set of attributes where each attribute has its categorical range: a set of possible values. These attributes are given in Table 13.4.

The description of a classification model with two classes of samples, C1 and C2, can be represented in an *if-then* form where on the left side is a Boolean expression of the input features' values and on the right side its corresponding class:

$$([A_1 = x] \wedge [A_5 = s]) \vee ([A_1 = y] \wedge [A_4 = n]) \Rightarrow C_1$$

$$([A_3 = y] \wedge [A_4 = n]) \vee (A_1 = x) \Rightarrow C_2$$

These classification rules or classifiers can be represented in a more general way: as some strings over a given alphabet. For our data set with six inputs and one output, each classifier has the form

$$(p_1, p_2, p_3, p_4, p_5, p_6) : d$$

TABLE 13.4. Attributes A_i with Possible Values for a Given Data Set s

Attributes	Values
A_1	x, y, z
A_2	x, y, z
A_3	y, n
A_4	m, n, p
A_5	r, s, t, u
A_6	y, n

where p_i denotes the value of the *i*th attribute ($1 \le i \le 6$) for the domains described in Table 13.4, and d is one of two classes. To describe the classification rules in a given form, it is necessary to include the "don't care" symbol "*" into the set of values for each attribute. For example, the new set of values for attribute A_1 is {x, y, z, *}. Similar extensions are given for other attributes. The rules that were given earlier for classes C_1 and C_2 can be decomposed to the segments under conjunction (AND logical operation) and expressed as

$$(x * * * s *): C_1$$
$$(y * * n * *): C_1$$
$$(* * y n * *): C_2$$
$$(x * * * * *): C_2$$

To simplify the example, we assume that the system has to classify into only two classes: C_1 and *not*C_1. Any system can be easily generalized to handle multiple classes (multiple classification). For a simple classification with a single rule C_1 we can accept only two values for d: d = 1 (member of the class C_1) and d = 0 (not a member of the class C_1).

Let us assume that at some stage of the learning process, there is a small and randomly generated population of classifiers Q in the system, where each classifier is given with its strength s:

$$Q_1 \quad (* * * m s *): 1, \quad s_1 = 12.3$$
$$Q_2 \quad (* * y * * n): 0, \quad s_2 = 10.1$$
$$Q_3 \quad (x y * * * *): 1, \quad s_3 = 8.7$$
$$Q_4 \quad (* z * * * *): 0, \quad s_4 = 2.3$$

Strengths s_i are parameters that are computed based on the available training data set. They show the fitness of a rule to the training data set, and they are proportional to the percentage of the data set supported by the rule.

The basic iterative steps of a GA, including corresponding operators, are applied here to optimize the set of rules with respect to the fitness function of the rules to training data set. The operators used in this learning technique are, again, mutation and crossover. However, some modifications are necessary for mutation. Let us consider the first attribute A_1 with its domain {x, y, z, *}. Thus, when mutation is called, we would change the mutated character (code) to one of the other three values that have equal probability. The strength of the offspring is usually the same as that of its parents. For example, if mutation is the rule Q_3 on the randomly selected position 2, replacing the value y with a randomly selected value *, the new mutated classifier will be

$$Q_{3M} \quad (x * * * * *): 1, \quad s_{3M} = 8.7$$

The crossover does not require any modification. We take advantage of the fact that all classifiers are of equal length. Therefore, to crossover two selected parents, say Q_1 and Q_2

$$\Downarrow$$

$$Q1 \quad (* * * \text{ m s } *):1$$

$$Q2 \quad (* * \text{ y } * * \text{ n}):0$$

we generate a random crossover-position point. Suppose that we crossover after the third character in the string as marked; then the offspring are

$$Q_{1c} \quad (* * * * * \text{ n}):0, \quad \text{and}$$

$$Q_{2c} \quad (* * \text{ y m s } *):1$$

The strength of the offspring is an average (possibly weighted) of the parents' strengths. Now the system is ready to continue its learning process: starting another cycle, accepting further positive and negative samples from the training set, and modifying the strengths of classifiers as a measure of fitness. We can note that the training data set is included in the learning process through evaluation of schema's strengths for each iteration. We expect that the population of classifiers converges to some rules with very high strengths.

One of the possible implementations of the previous ideas is the *GIL system*, which moves the GA closer to the symbolic level—mainly by defining specialized operators that manipulate binary strings. Previous symbolic classifiers are translated into binary strings, where for each attribute a binary string of fixed length is generated. The length is equal to the number of possible values for the given attribute. In the string, the required value is set to 1 and all others are set to 0. For example, if attribute A_1 has the value z, it is represented with the binary string 001 (0's are for values x, y). If the value of some attribute is *, that means that all values are possible, so it is represented with value 1 in all positions of the binary string.

For our previous example, with six attributes and a total number of 17 different values for all attributes, the classifier symbolically represented as

$$(\text{x } * * * \text{ r } *) \vee (\text{y } * * \text{ n } * *):1$$

can be transformed into the binary representation

$$(100 \,|\, 111 \,|\, 11 \,|\, 111 \,|\, 1000 \,|\, 11 \vee 010 \,|\, 111 \,|\, 11 \,|\, 010 \,|\, 1111 \,|\, 11)$$

where bars separate bit sets for each attribute. The operators of the GIL system are modeled on inductive reasoning, which includes various inductive operators such as RuleExchange, RuleCopy, RuleGeneralization, RuleSpecialization, RuleSplit, SelectorDrop, ReferenceChange, and ReferenceExtension. We discuss some of them in turn.

13.6.1 RuleExchange

The RuleExchange operator is similar to a crossover of the classical GA as it exchanges selected complex between two parent chromosomes. For example, two parents (two rules)

$$\Downarrow$$

$$(100\,|\,111\,|\,11\,|\,111\,|\,1000\,|\,11 \vee 010\,|\,111\,|\,11\,|\,010\,|\,1111\,|\,11) \quad \text{and}$$

$$(111\,|\,001\,|\,01\,|\,111\,|\,1111\,|\,01 \vee 110\,|\,100\,|\,10\,|\,010\,|\,0011\,|\,01)$$

$$\Uparrow$$

may produce the following offspring (new rules)

$$(100\,|\,111\,|\,11\,|\,111\,|\,1000\,|\,11 \vee 110\,|\,100\,|\,10\,|\,010\,|\,0011\,|\,01) \quad \text{and}$$

$$(111\,|\,001\,|\,01\,|\,111\,|\,1111\,|\,01 \vee 010\,|\,111\,|\,11\,|\,010\,|\,1111\,|\,11).$$

13.6.2 RuleGeneralization

This unary operator generalizes a random subset of complexes. For example, for a parent

$$(100\,|\,111\,|\,11\,|\,111\,|\,1000\,|\,11 \vee 110\ 100\ 10\ 010\ 0011\ 01 \vee 010\,|\,111\,|\,11\,|\,010\,|\,1111\,|\,11)$$

and the second and third complexes selected for generalization, the bits are *OR*ed and the following offspring is produced:

$$(100\,|\,111\,|\,11\,|\,111\,|\,1000\,|\,11 \vee 110\,|\,111\,|\,11\,|\,010\,|\,1111\,|\,11).$$

13.6.3 RuleSpecialization

This unary operator specializes a random subset of complexes. For example, for a parent

$$(100\,|\,111\,|\,11\,|\,111\,|\,1000\,|\,11 \vee 110\,|\,100\,|\,10\,|\,010\,|\,0011\,|\,01 \vee 010\,|\,111\,|\,11\,|\,010\,|\,1111\,|\,11)$$

and the second and third complexes selected for specialization, the bits are *AND*ed, and the following offspring is produced:

$$(100\,|\,111\,|\,11\,|\,111\,|\,1000\,|\,11 \vee 010\,|\,100\,|\,10\,|\,010\,|\,0011\,|\,01).$$

13.6.4 RuleSplit

This operator acts on a single complex, splitting it into a number of complexes. For example, a parent

$$(100\,|\,111\,|\,11\,|\,111\,|\,1000\,|\,11)$$

$$===$$

$$\Uparrow$$

may produce the following offspring (the operator splits the second selector):

$$(100\,|\,011\,|\,11\,|\,111\,|\,1000\,|\,11 \vee 100\,|\,100\,|\,11\,|\,111\,|\,1000\,|\,11)$$

The GIL system is a complex, inductive-learning system based on GA principles. It requires a number of parameters, such as the probabilities of applying each operator. The process is iterative. At each iteration, all chromosomes are evaluated with respect to their completeness, consistency, and fitness criteria, and a new population is formed with those chromosomes that are better and more likely to appear. The operators are applied to the new population, and the cycle is repeated.

13.7 GAS FOR CLUSTERING

Much effort has been undertaken toward applying GAs to provide better solutions than those found by traditional clustering algorithms. The emphasis was on appropriate encoding schemes, specific genetic operators, and corresponding fitness functions. Several encoding schemes have been proposed specifically for data clustering, and the main three types are *binary*, *integer*, and *real encoding*.

The *binary encoding* solution is usually represented as a binary string of length N, where N is the number of data set samples. Each position of the binary string corresponds to a particular sample. The value of the ith gene is 1 if the ith sample is a prototype of a cluster, and 0 otherwise. For example, the data set s in Table 13.5 can be encoded by means of the string [0100001010], in which samples 2, 7, and 9 are prototypes for clusters C_1, C_2, and C_3. The total number of 1s in the string is equal to an a priori defined number of clusters. Clearly, such an encoding scheme leads to a *medoid-based representation* in which the cluster prototypes coincide with representative samples from the data set. There is an alternative way of representing a data partition using a binary encoding. The matrix of $k \times N$ dimensions is used in which the rows represent clusters, and the columns represent samples. In this case, if the jth sample belongs to the ith cluster then 1 is assigned to (i,j) genotype, whereas the other elements of the same column receive 0. For example, using this representation, the data set in Table 13.5 would be encoded as 3×10 matrix in Table 13.6.

Integer encoding uses a vector of N integer positions, where N is the number of data set samples.

Each position corresponds to a particular sample; that is, the ith position (gene) represents the ith data sample. Assuming that there are k clusters, each gene has a value over the alphabet $\{1, 2, 3, \ldots, k\}$. These values define the cluster labels. For example, the integer vector [1111222233] represents the clusters depicted in Table 13.5. Another way of representing a partition by means of an integer-encoding scheme involves using

TABLE 13.5. Three Clusters Are Defined for a Given Data Set s

Samples	Feature 1	Feature 2	Cluster
1	1	1	C_1
2	1	2	C_1
3	2	1	C_1
4	2	2	C_1
5	10	1	C_2
6	10	2	C_2
7	11	1	C_2
8	11	2	C_2
9	5	5	C_3
10	5	6	C_3

TABLE 13.6. Binary Encoded Data Set s Given in Table 13.5

1	1	1	1	0	0	0	0	0	0
0	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	1	1

an array of only k elements to provide a *medoid-based representation* of the data set. In this case, each array element represents the index of the sample $\mathbf{x}_i$, $i = 1, 2, \ldots, N$ (with respect to the order the samples appear in the data set) corresponding to the prototype of a given cluster. As an example, the array [1 6 10] may represent a partition in which 1, 6, and 10 are indices of the cluster prototypes (medoids) for the data set in Table 13.5. Integer encoding is usually more computationally efficient than the binary-encoding schemes.

Real encoding is the third encoding scheme where the genotypes are made up of real numbers that represent the coordinates of the cluster centroids. In an n-dimensional space, the first n positions represent the n coordinates of the first centroid; the next n positions represent the coordinates of the second centroid, and so forth. To illustrate this, the genotype [1.5 1.5 10.5 1.5 5.0 5.5] encodes the three centroids: (1.5, 1.5), (10.5, 1.5), and (5.0, 5.5) of clusters C1, C2, and C3 in Table 13.5, respectively.

The second important decision, in applying GAs for clustering, is a selection of appropriate *genetic operators*. A number of crossover and mutation operators are proposed trying to solve an important problem of the *context-insensitivity* in GAs. When traditional genetic operators are employed in clustering problems, they usually just manipulate gene values without taking into account their connections with other genes. For example, the crossover operation presented in Figure 13.7 shows how two parents representing the same solution to the clustering problem (different labeling but the same integer encoding) produce the resulting offspring representing clustering solutions different from the ones encoded into their parents. Moreover, assuming that the number

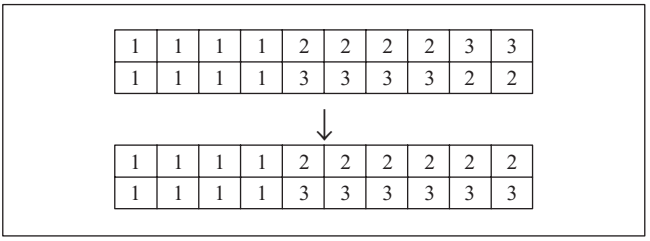

Figure 13.7. Equal parents produce different offspring through crossover.

of clusters has been fixed in advance as $k = 3$, invalid solutions with only two clusters have been derived. Therefore, it is necessary to develop specially designed genetic operators for clustering problems. For example, the crossover operator should be repeatedly applied or randomly scrambling mutation performed until a valid child has been found.

Different clustering validity criteria are adapted as *fitness functions* to evolve data partitions in clustering problems. They depend primarily on encoding schemes but also on a selected set of genetic operators. We will illustrate in this text only one example of a clustering fitness function when a *real, centroid-based* encoding scheme is used. Fitness function f minimizes the sum of squared Euclidean distances of the samples from their respective cluster means. Formally, the fitness function $f(C_1,C_2, \ldots, C_k)$ is:

$$f(C_1, C_2, \ldots, C_k) = \sum_{j=1}^{k} \sum_{x_i \in C_j} |x_i - z_j|^2$$

where $\{C_1, C_2, \ldots, C_k\}$ is the set of k clusters encoded into the genotype, x_i is a sample in a data set, and z_j is the centroid of cluster C_j. It is important to stress that this criterion is valid only if the number of clusters k is given in advance, and it minimizes the intra-cluster distances and maximizes the intercluster distances as well. In general, fitness functions are based on the distance between samples and either cluster centroids or medoids. Although these types of functions are widely used, usually they are biased toward the discovery of spherical clusters, which clearly will be inappropriate in many real-world applications. Other approaches are possible, including a density-based fitness function. In practice, the success of a GA to solve a clustering problem is highly dependent upon how it has been designed in terms of encoding scheme, operators, fitness function, selection procedure, and initial population generation.

13.8 REVIEW QUESTIONS AND PROBLEMS

1. Given a binary string that represents a concatenation of four attribute values:

$$\{2, 5, 4, 7\} = \{010101100111\}$$

use this example to explain the basic concepts of a GA and their equivalents in natural evolution.

2. If we want to optimize a function f(x) using a GA where the precision requirement for x is six decimal places and the range is [−1, 2], what will be the length of a binary vector (chromosome)?

3. If v1 = (0 0 1 1 0 0 1 1) and v2 = (0 1 0 1 0 1 0 1) are two chromosomes, and suppose that the crossover point is randomly selected after the fifth gene, what are the two resulting offspring?

4. Given the schema (* 1 * 0 0), what are the strings that match with it?

5. What is the number of strings that match with the schema (* * * * * * * *)?

6. The function $f(x) = -x^2 + 16x$ is defined on interval [0, 63]. Use two iterations of a GA to establish the approximate solution for a maximum of f(x).

7. For the function f(x) given in Problem number 6, compare three schemata

$$S_1 = (*1*1**)$$

$$S_2 = (*10*1*)$$

$$S_3 = (**1***)$$

with respect to order (O), length (L), and fitness (F).

8. Given a parent chromosome (1 1 0 0 0 1 0 0 0 1), what is the potential offspring (give examples) if the mutation probability is

(a) $p_m = 1.0$
(b) $p_m = 0.5$
(c) $p_m = 0.2$
(d) $p_m = 0.1$
(e) $p_m = 0.0001$

9. Explain the basic principles of the building-block hypothesis and its potential applications.

10. Perform a PMC operation for two strings S1 and S2, in which two randomly selected crossing points are given:

$$S_1 = \{A\ C\ B\ D\ F\ G\ E\}$$

$$S_2 = \{B\ D\ C\ F\ E\ G\ A\}$$

$$\uparrow\uparrow$$

11. Search the Web to find the basic characteristics of publicly available or commercial software tools that are based on GAs. Document the results of your search.

13.9 REFERENCES FOR FURTHER STUDY

Fogel, D. B., ed., *Evolutionary Computation*, IEEE Press, New York, 1998.

The book provides a collection of 30 landmark papers, and it spans the entire history of evolutionary computation—from today's research back to its very origins more than 40 years ago.

Goldenberg, D. E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison Wesley, Reading, MA, 1989.

This book represents one of the first comprehensive texts on GAs. It introduces in a very systematic way most of the techniques that are, with small modifications and improvements, part of today's approximate-optimization solutions.

Hruschka, E., R. Campello, A. Freitas, A. Carvalho, A Survey of Evolutionary Algorithms for Clustering, *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, Vol. 39, No. 2, 2009, pp. 133–155.

This paper presents a survey of evolutionary algorithms designed for clustering tasks. It tries to reflect the profile of this area by focusing more on those subjects that have been given more importance in the literature. In this context, most of the paper is devoted to partitional algorithms that look for hard clustering of data, although overlapping (i.e., soft and fuzzy) approaches are also covered in the paper. The paper ends by addressing some important issues and open questions that can be the subjects of future research.

Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer, Berlin, Germany, 1999.

This textbook explains the field of GAs in simple terms, and discusses the efficiency of its methods in many interesting test cases. The importance of these techniques is their applicability to many hard-optimization problems specified with large amounts of discrete data, such as the TSP, scheduling, partitioning, and control.