Evolutionary computation 7

In which we consider the field of evolutionary computation, including genetic algorithms, evolution strategies and genetic programming, and their applications to machine learning.

7.1 Introduction, or can evolution be intelligent?

Intelligence can be defined as the capability of a system to adapt its behaviour to an ever-changing environment. According to Alan Turing (Turing, 1950), the form or appearance of a system is irrelevant to its intelligence. However, from our everyday experience we know that evidences of intelligent behaviour are easily observed in humans. But we are products of evolution, and thus by modelling the process of evolution, we might expect to create intelligent behaviour. Evolutionary computation simulates evolution on a computer. The result of such a simulation is a series of optimisation algorithms, usually based on a simple set of rules. Optimisation iteratively improves the quality of solutions until an optimal, or at least feasible, solution is found.

But is evolution really intelligent? We can consider the behaviour of an individual organism as an inductive inference about some yet unknown aspects of its environment (Fogel *et al.*, 1966). Then if, over successive generations, the organism survives, we can say that this organism is capable of learning to predict changes in its environment. Evolution is a tortuously slow process from the human perspective, but the simulation of evolution on a computer does not take billions of years!

The evolutionary approach to machine learning is based on computational models of natural selection and genetics. We call them **evolutionary computation**, an umbrella term that combines **genetic algorithms**, **evolution strategies** and **genetic programming**. All these techniques simulate evolution by using the processes of **selection**, **mutation** and **reproduction**.

7.2 Simulation of natural evolution

On 1 July 1858, Charles Darwin presented his theory of evolution before the Linnean Society of London. This day marks the beginning of a revolution in biology. Darwin's classical theory of evolution, together with Weismann's theory of natural selection and Mendel's concept of genetics, now represent the neo-Darwinian paradigm (Keeton, 1980; Mayr, 1988).

Neo-Darwinism is based on processes of reproduction, mutation, competition and selection. The power to reproduce appears to be an essential property of life. The power to mutate is also guaranteed in any living organism that reproduces itself in a continuously changing environment. Processes of competition and selection normally take place in the natural world, where expanding populations of different species are limited by a finite space.

If the process of evolution is to be emulated on a computer, what is being optimised by evolution in natural life? Evolution can be seen as a process leading to the maintenance or increase of a population's ability to survive and reproduce in a specific environment (Hartl and Clark, 1989). This ability is called **evolutionary fitness**. Although fitness cannot be measured directly, it can be estimated on the basis of the ecology and functional morphology of the organism in its environment (Hoffman, 1989). Evolutionary fitness can also be viewed as a measure of the organism's ability to anticipate changes in its environment (Atmar, 1994). Thus, the fitness, or the quantitative measure of the ability to predict environmental changes and respond adequately, can be considered as the quality that is being optimised in natural life.

To illustrate fitness, we can use the concept of **adaptive topology** (Wright, 1932). We can represent a given environment by a landscape where each peak corresponds to the optimised fitness of a species. As evolution takes place, each species of a given population moves up the slopes of the landscape towards the peaks. Environmental conditions change over time, and thus the species have to continuously adjust their routes. As a result, only the fittest can reach the peaks.

Adaptive topology is a continuous function; it simulates the fact that the environment, or natural topology, is not static. The shape of the topology changes over time, and all species continually undergo selection. The goal of evolution is to generate a population of individuals with increasing fitness.

But how is a population with increasing fitness generated? Michalewicz (1996) suggests a simple explanation based on a population of rabbits. Some rabbits are faster than others, and we may say that these rabbits possess superior fitness because they have a greater chance of avoiding foxes, surviving and then breeding. Of course, some of the slower rabbits may survive too. As a result, some slow rabbits breed with fast rabbits, some fast with other fast rabbits, and some slow rabbits with other slow rabbits. In other words, the breeding generates a mixture of rabbit genes. If two parents have superior fitness, there is a good chance that a combination of their genes will produce an offspring with even higher fitness. Over time the entire population of rabbits becomes faster to meet their environmental challenges in the face of foxes. However, environmental conditions could change in favour of say, fat but smart rabbits. To optimise survival, the genetic structure of the rabbit population will change accordingly. At the same time, faster and smarter rabbits encourage the breeding of faster and smarter foxes. Natural evolution is a continuous, never-ending process.

Can we simulate the process of natural evolution in a computer?

Several different methods of evolutionary computation are now known. They all simulate natural evolution, generally by creating a population of individuals, evaluating their fitness, generating a new population through genetic operations, and repeating this process a number of times. However, there are different ways of performing evolutionary computation. We will start with **genetic algorithms** (GAs) as most of the other evolutionary algorithms can be viewed as variations of GAs.

In the early 1970s, John Holland, one of the founders of evolutionary computation, introduced the concept of genetic algorithms (Holland, 1975). His aim was to make computers do what nature does. As a computer scientist, Holland was concerned with algorithms that manipulate strings of binary digits. He viewed these algorithms as an abstract form of natural evolution. Holland's GA can be represented by a sequence of procedural steps for moving from one population of artificial 'chromosomes' to a new population. It uses 'natural' selection and genetics-inspired techniques known as crossover and mutation. Each chromosome consists of a number of 'genes', and each gene is represented by 0 or 1, as shown in Figure 7.1.

Nature has an ability to adapt and learn without being told what to do. In other words, nature finds good chromosomes blindly. GAs do the same. Two mechanisms link a GA to the problem it is solving: **encoding** and **evaluation**.

In Holland's work, encoding is carried out by representing chromosomes as strings of ones and zeros. Although many other types of encoding techniques have been invented (Davis, 1991), no one type works best for all problems. We will use bit strings as the most popular technique.

An evaluation function is used to measure the chromosome's performance, or fitness, for the problem to be solved (an evaluation function in GAs plays the same role the environment plays in natural evolution). The GA uses a measure of fitness of individual chromosomes to carry out reproduction. As reproduction takes place, the crossover operator exchanges parts of two single chromosomes, and the mutation operator changes the gene value in some randomly chosen location of the chromosome. As a result, after a number of successive reproductions, the less fit chromosomes become extinct, while those best able to survive gradually come to dominate the population. It is a simple approach, yet even crude reproduction mechanisms display highly complex behaviour and are capable of solving some difficult problems.

Let us now discuss genetic algorithms in more detail.

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

Figure 7.1 A 16-bit binary string of an artificial chromosome

7.3 Genetic algorithms

We start with a definition: genetic algorithms are a class of stochastic search algorithms based on biological evolution. Given a clearly defined problem to be solved and a binary string representation for candidate solutions, a basic GA can be represented as in Figure 7.2. A GA applies the following major steps (Davis, 1991; Mitchell, 1996):

Step 1: Represent the problem variable domain as a chromosome of a fixed length, choose the size of a chromosome population N, the crossover probability p_c and the mutation probability p_m.

Step 2: Define a fitness function to measure the performance, or fitness, of an individual chromosome in the problem domain. The fitness function establishes the basis for selecting chromosomes that will be mated during reproduction.

Step 3: Randomly generate an initial population of chromosomes of size N:

$$x_1, x_2, \ldots, x_N$$

Step 4: Calculate the fitness of each individual chromosome:

$$f(x_1), f(x_2), \ldots, f(x_N)$$

Step 5: Select a pair of chromosomes for mating from the current population. Parent chromosomes are selected with a probability related to their fitness. Highly fit chromosomes have a higher probability of being selected for mating than less fit chromosomes.

Step 6: Create a pair of offspring chromosomes by applying the genetic operators – crossover and mutation.

Step 7: Place the created offspring chromosomes in the new population.

Step 8: Repeat Step 5 until the size of the new chromosome population becomes equal to the size of the initial population, N.

Step 9: Replace the initial (parent) chromosome population with the new (offspring) population.

Step 10: Go to Step 4, and repeat the process until the termination criterion is satisfied.

As we see, a GA represents an iterative process. Each iteration is called a **generation**. A typical number of generations for a simple GA can range from 50 to over 500 (Mitchell, 1996). The entire set of generations is called a **run**. At the end of a run, we expect to find one or more highly fit chromosomes.

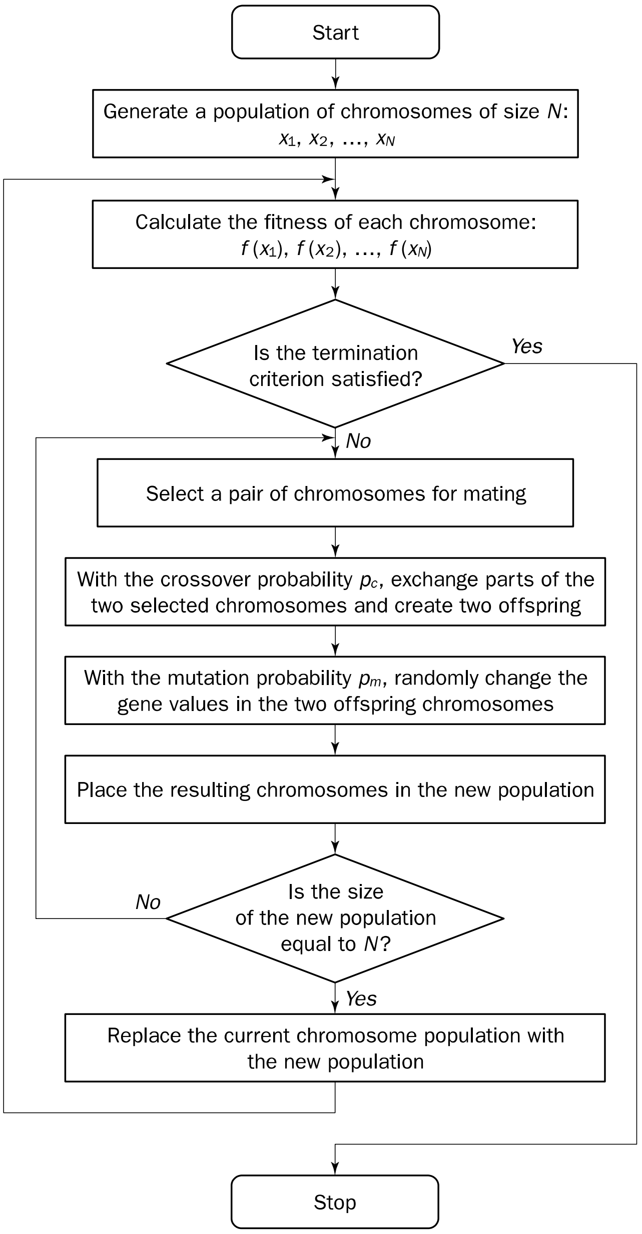

Start
Generate a population of chromosomes of size N: $x_1, x_2, \ldots, x_N$
Calculate the fitness of each chromosome: $f(x_1), f(x_2), \ldots, f(x_N)$
Is the termination criterion satisfied?
Yes
No
Select a pair of chromosomes for mating
With the crossover probability p_c, exchange parts of the two selected chromosomes and create two offspring
With the mutation probability p_m, randomly change the gene values in the two offspring chromosomes
Place the resulting chromosomes in the new population
Is the size of the new population equal to N?
No
Yes
Replace the current chromosome population with the new population
Stop

Figure 7.2 A basic genetic algorithm

Are any conventional termination criteria used in genetic algorithms?

Because GAs use a stochastic search method, the fitness of a population may remain stable for a number of generations before a superior chromosome appears. This makes applying conventional termination criteria problematic. A common practice is to terminate a GA after a specified number of generations and then examine the best chromosomes in the population. If no satisfactory solution is found, the GA is restarted.

A simple example will help us to understand how a GA works. Let us find the maximum value of the function $(15x - x^2)$ where parameter x varies between 0 and 15. For simplicity, we may assume that x takes only integer values. Thus, chromosomes can be built with only four genes:

Integer	Binary code	Integer	Binary code	Integer	Binary code
1	0 0 0 1	6	0 1 1 0	11	1 0 1 1
2	0 0 1 0	7	0 1 1 1	12	1 1 0 0
3	0 0 1 1	8	1 0 0 0	13	1 1 0 1
4	0 1 0 0	9	1 0 0 1	14	1 1 1 0
5	0 1 0 1	10	1 0 1 0	15	1 1 1 1

Suppose that the size of the chromosome population N is 6, the crossover probability p_c equals 0.7, and the mutation probability p_m equals 0.001. (The values chosen for p_c and p_m are fairly typical in GAs.) The fitness function in our example is defined by

$$f(x) = 15x - x^2$$

The GA creates an initial population of chromosomes by filling six 4-bit strings with randomly generated ones and zeros. The initial population might look like that shown in Table 7.1. The chromosomes' initial locations on the fitness function are illustrated in Figure 7.3(a).

A real practical problem would typically have a population of thousands of chromosomes.

The next step is to calculate the fitness of each individual chromosome. The results are also shown in Table 7.1. The average fitness of the initial population is 36. In order to improve it, the initial population is modified by using selection, crossover and mutation, the genetic operators.

In natural selection, only the fittest species can survive, breed, and thereby pass their genes on to the next generation. GAs use a similar approach, but

Table 7.1 The initial randomly generated population of chromosomes

Chromosome label	Chromosome string	Decoded integer	Chromosome fitness	Fitness ratio, %
X1	1 1 0 0	12	36	16.5
X2	0 1 0 0	4	44	20.2
X3	0 0 0 1	1	14	6.4
X4	1 1 1 0	14	14	6.4
X5	0 1 1 1	7	56	25.7
X6	1 0 0 1	9	54	24.8

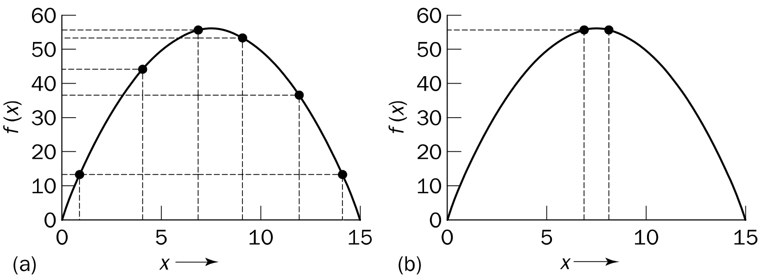


Figure 7.3 The fitness function and chromosome locations: (a) chromosome initial locations; (b) chromosome final locations

unlike nature, the size of the chromosome population remains unchanged from one generation to the next.

How can we maintain the size of the population constant, and at the same time improve its average fitness?

The last column in Table 7.1 shows the ratio of the individual chromosome's fitness to the population's total fitness. This ratio determines the chromosome's chance of being selected for mating. Thus, the chromosomes X5 and X6 stand a fair chance, while the chromosomes X3 and X4 have a very low probability of being selected. As a result, the chromosome's average fitness improves from one generation to the next.

One of the most commonly used chromosome selection techniques is the **roulette wheel selection** (Goldberg, 1989; Davis, 1991). Figure 7.4 illustrates the roulette wheel for our example. As you can see, each chromosome is given a slice of a circular roulette wheel. The area of the slice within the wheel is equal to the chromosome fitness ratio (see Table 7.1). For instance, the chromosomes X5 and X6 (the most fit chromosomes) occupy the largest areas, whereas the chromosomes X3 and X4 (the least fit) have much smaller segments in the roulette wheel. To select a chromosome for mating, a random number is generated in the interval $[0, 100]$, and the chromosome whose segment spans the random number is selected. It is like spinning a roulette wheel where each chromosome has a segment on the wheel proportional to its fitness. The roulette wheel is spun, and

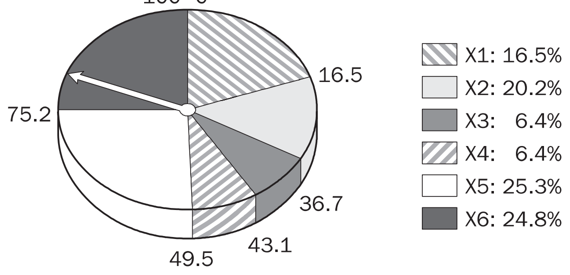


Figure 7.4 Roulette wheel selection

when the arrow comes to rest on one of the segments, the corresponding chromosome is selected.

In our example, we have an initial population of six chromosomes. Thus, to establish the same population in the next generation, the roulette wheel would be spun six times. The first two spins might select chromosomes X6 and X2 to become parents, the second pair of spins might choose chromosomes X1 and X5, and the last two spins might select chromosomes X2 and X5.

Once a pair of parent chromosomes is selected, the crossover operator is applied.

How does the crossover operator work?

First, the crossover operator randomly chooses a crossover point where two parent chromosomes 'break', and then exchanges the chromosome parts after that point. As a result, two new offspring are created. For example, the chromosomes X6 and X2 could be crossed over after the second gene in each to produce the two offspring, as shown in Figure 7.5.

If a pair of chromosomes does not cross over, then chromosome **cloning** takes place, and the offspring are created as exact copies of each parent. For example, the parent chromosomes X2 and X5 may not cross over. Instead, they create the offspring that are their exact copies, as shown in Figure 7.5.

A value of 0.7 for the crossover probability generally produces good results. After selection and crossover, the average fitness of the chromosome population has improved and gone from 36 to 42.

What does mutation represent?

Mutation, which is rare in nature, represents a change in the gene. It may lead to a significant improvement in fitness, but more often has rather harmful results.

So why use mutation at all? Holland introduced mutation as a background operator (Holland, 1975). Its role is to provide a guarantee that the search algorithm is not trapped on a local optimum. The sequence of selection and crossover operations may stagnate at any homogeneous set of solutions. Under such conditions, all chromosomes are identical, and thus the average fitness of the population cannot be improved. However, the solution might appear to become optimal, or rather locally optimal, only because the search algorithm is not able to proceed any further. Mutation is equivalent to a random search, and aids us in avoiding loss of genetic diversity.

How does the mutation operator work?

The mutation operator flips a randomly selected gene in a chromosome. For example, the chromosome X1′ might be mutated in its second gene, and the chromosome X2 in its third gene, as shown in Figure 7.5. Mutation can occur at any gene in a chromosome with some probability. The mutation probability is quite small in nature, and is kept quite low for GAs, typically in the range between 0.001 and 0.01.

Genetic algorithms assure the continuous improvement of the average fitness of the population, and after a number of generations (typically several hundred)

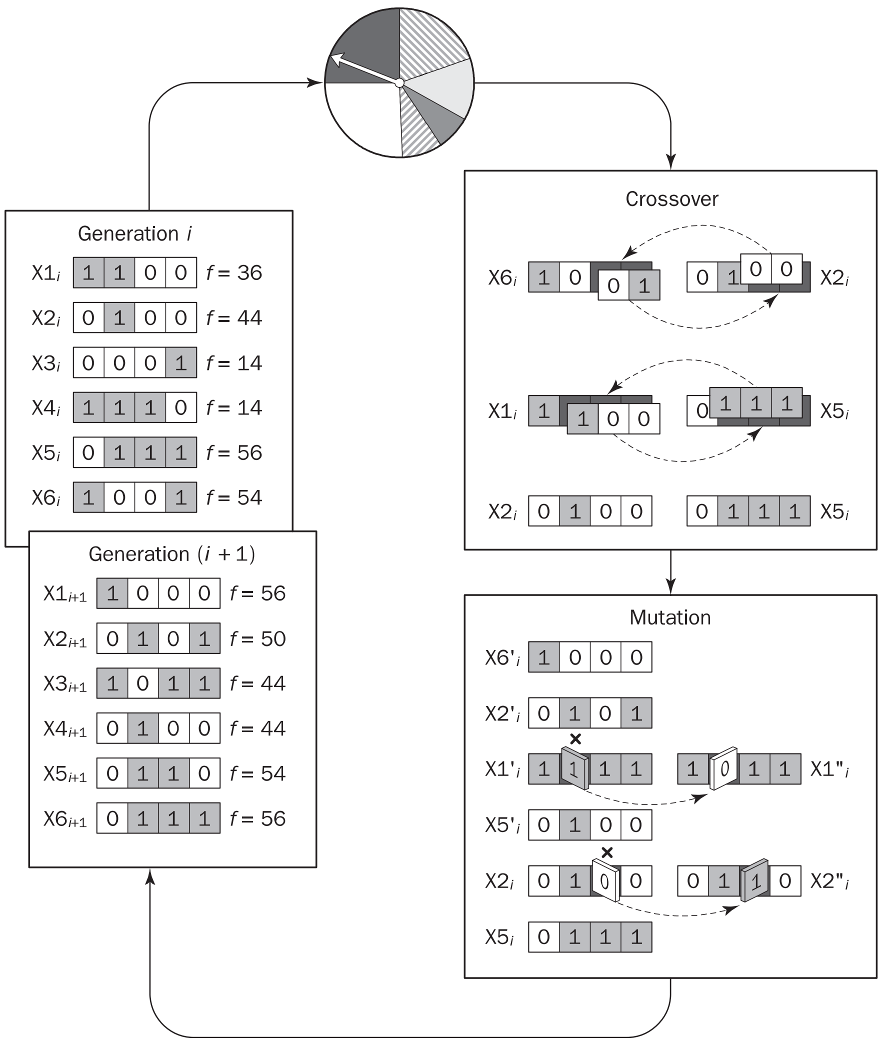


Figure 7.5 The GA cycle

the population evolves to a **near-optimal** solution. In our example, the final population would consist of only chromosomes $\boxed{0|1|1|1}$ and $\boxed{1|0|0|0}$. The chromosome's final locations on the fitness function are illustrated in Figure 7.3(b).

In this example, the problem has only one variable. It is easy to represent. But suppose it is desired to find the maximum of the 'peak' function of two variables:

$$f(x,y) = (1-x)^2 e^{-x^2-(y+1)^2} - (x - x^3 - y^3)e^{-x^2-y^2},$$

where parameters x and y vary between -3 and 3.

The first step is to represent the problem variables as a chromosome. In other words, we represent parameters x and y as a concatenated binary string:

$$\underbrace{\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{0}}_{x}\underbrace{\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{1}}_{y}$$

in which each parameter is represented by eight binary bits.

Then, we choose the size of the chromosome population, for instance 6, and randomly generate an initial population.

The next step is to calculate the fitness of each chromosome. This is done in two stages. First, a chromosome is decoded by converting it into two real numbers, x and y, in the interval between -3 and 3. Then the decoded values of x and y are substituted into the 'peak' function.

How is decoding done?

First, a chromosome, that is a string of 16 bits, is partitioned into two 8-bit strings:

$$\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{0} \text{ and } \boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{1}$$

Then these strings are converted from binary (base 2) to decimal (base 10):

$$\begin{aligned}(10001010)_2 &= 1\times 2^7 + 0\times 2^6 + 0\times 2^5 + 0\times 2^4 + 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 0\times 2^0 \\ &= (138)_{10}\end{aligned}$$

and

$$\begin{aligned}(00111011)_2 &= 0\times 2^7 + 0\times 2^6 + 1\times 2^5 + 1\times 2^4 + 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 1\times 2^0 \\ &= (59)_{10}\end{aligned}$$

Now the range of integers that can be handled by 8-bits, that is the range from 0 to $(2^8 - 1)$, is mapped to the actual range of parameters x and y, that is the range from -3 to 3:

$$\frac{6}{256-1} = 0.0235294$$

To obtain the actual values of x and y, we multiply their decimal values by 0.0235294 and subtract 3 from the results:

$$x = (138)_{10} \times 0.0235294 - 3 = 0.2470588$$

and

$$y = (59)_{10} \times 0.0235294 - 3 = -1.6117647$$

When necessary, we can also apply other decoding techniques, such as **Gray coding** (Caruana and Schaffer, 1988).

Using decoded values of *x* and *y* as inputs in the mathematical function, the GA calculates the fitness of each chromosome.

To find the maximum of the 'peak' function, we will use crossover with the probability equal to 0.7 and mutation with the probability equal to 0.001. As we mentioned earlier, a common practice in GAs is to specify the number of generations. Suppose the desired number of generations is 100. That is, the GA will create 100 generations of 6 chromosomes before stopping.

Figure 7.6(a) shows the initial locations of the chromosomes on the surface and contour plot of the 'peak' function. Each chromosome here is represented by a sphere. The initial population consists of randomly generated individuals that are dissimilar or **heterogeneous**. However, starting from the second generation, crossover begins to recombine features of the best chromosomes, and the population begins to converge on the peak containing the maximum, as shown in Figure 7.6(b). From then until the final generation, the GA is searching around this peak with mutation, resulting in diversity. Figure 7.6(c) shows the final chromosome generation. However, the population has converged on a chromosome lying on a local maximum of the 'peak' function.

But we are looking for the global maximum, so can we be sure the search is for the optimal solution? The most serious problem in the use of GAs is concerned with the quality of the results, in particular whether or not an optimal solution is

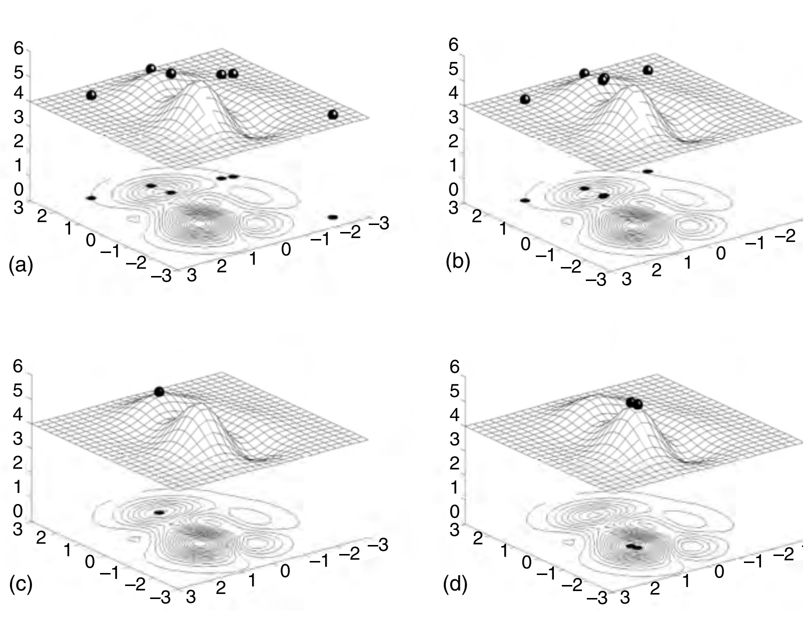
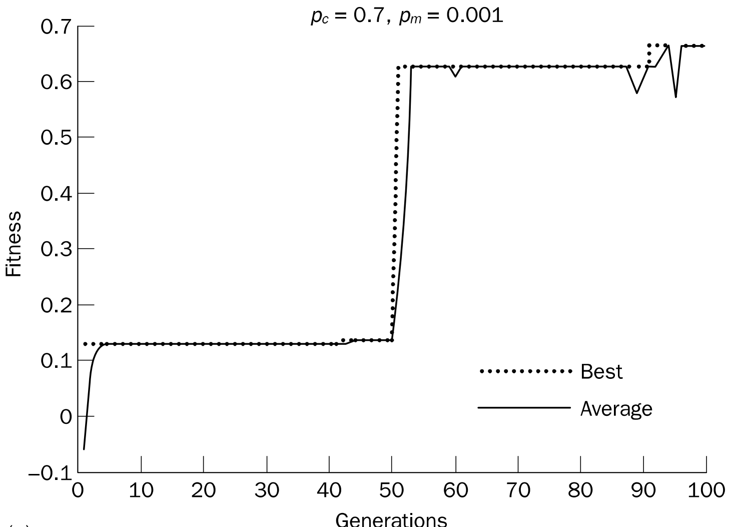
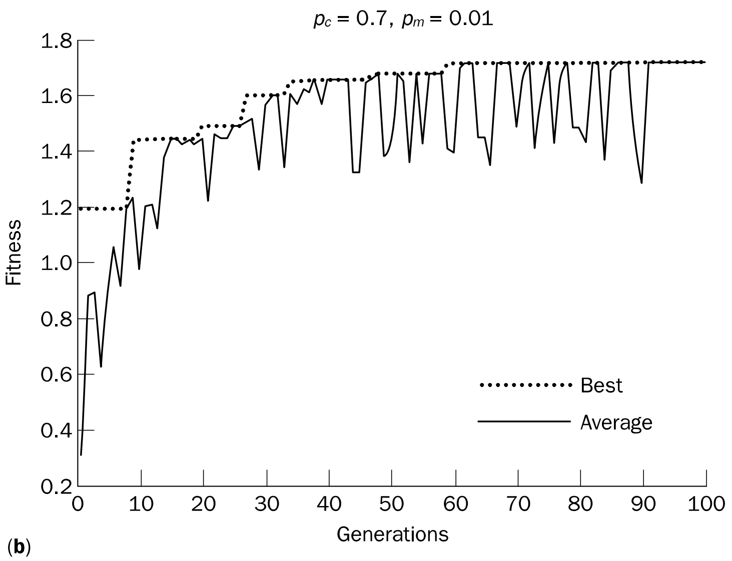


Figure 7.6 Chromosome locations on the surface and contour plot of the 'peak' function: (a) initial population; (b) first generation; (c) local maximum solution; (d) global maximum solution

being reached. One way of providing some degree of insurance is to compare results obtained under different rates of mutation. Let us, for example, increase the mutation rate to 0.01 and rerun the GA. The population might now converge on the chromosomes shown in Figure 7.6(d). However, to be sure of steady results we must increase the size of the chromosome population.



(**a**)



(**b**)

Figure 7.7 Performance graphs for 100 generations of 6 chromosomes: (a) local maximum solution and (b) global maximum solution of the 'peak' function

A surface of a mathematical function of the sort given in Figure 7.6 is a convenient medium for displaying the GA's performance. However, fitness functions for real world problems cannot be easily represented graphically. Instead, we can use **performance graphs**.

What is a performance graph?

Since genetic algorithms are stochastic, their performance usually varies from generation to generation. As a result, a curve showing the average performance of the entire population of chromosomes as well as a curve showing the performance of the best individual in the population is a useful way of examining the behaviour of a GA over the chosen number of generations.

Figures 7.7(a) and (b) show plots of the best and average values of the fitness function across 100 generations. The *x*-axis of the performance graph indicates how many generations have been created and evaluated at the particular point in the run, and the *y*-axis displays the value of the fitness function at that point.

The erratic behaviour of the average performance curves is due to mutation. The mutation operator allows a GA to explore the landscape in a random manner. Mutation may lead to significant improvement in the population fitness, but more often decreases it. To ensure diversity and at the same time to reduce the harmful effects of mutation, we can increase the size of the chromosome population. Figure 7.8 shows performance graphs for 20 generations of 60 chromosomes. The best and average curves represented here are typical for GAs. As you can see, the average curve rises rapidly at the beginning of the run, but then as the population converges on the nearly optimal solution, it rises more slowly, and finally flattens at the end.

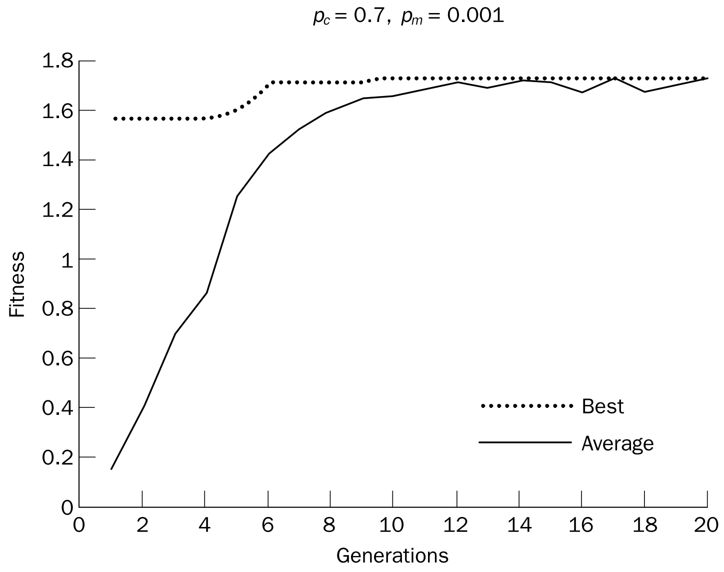


Figure 7.8 Performance graphs for 20 generations of 60 chromosomes

7.4 Why genetic algorithms work

The GA techniques have a solid theoretical foundation (Holland, 1975; Goldberg, 1989; Rawlins, 1991; Whitley, 1993). That foundation is based on the **Schema Theorem**.

John Holland introduced the notation of **schema** (Holland, 1975), which came from the Greek word meaning 'form'. A schema is a set of bit strings of ones, zeros and asterisks, where each asterisk can assume either value 1 or 0. The ones and zeros represent the fixed positions of a schema, while asterisks represent 'wild cards'. For example, the schema $\begin{array}{|c|c|c|c|}\hline 1 & * & * & 0 \\ \hline\end{array}$ stands for a set of 4-bit strings. Each string in this set begins with 1 and ends with 0. These strings are called **instances** of the schema.

What is the relationship between a schema and a chromosome?

It is simple. A chromosome matches a schema when the fixed positions in the schema match the corresponding positions in the chromosome. For example, the schema *H*

$$\begin{array}{|c|c|c|c|}\hline 1 & * & * & 0 \\ \hline\end{array}$$

matches the following set of 4-bit chromosomes:

$$\begin{array}{|c|c|c|c|}\hline 1 & 1 & 1 & 0 \\ \hline\end{array}$$
$$\begin{array}{|c|c|c|c|}\hline 1 & 1 & 0 & 0 \\ \hline\end{array}$$
$$\begin{array}{|c|c|c|c|}\hline 1 & 0 & 1 & 0 \\ \hline\end{array}$$
$$\begin{array}{|c|c|c|c|}\hline 1 & 0 & 0 & 0 \\ \hline\end{array}$$

Each chromosome here begins with 1 and ends with 0. These chromosomes are said to be instances of the schema *H*.

The number of defined bits (non-asterisks) in a schema is called the **order**. The schema *H*, for example, has two defined bits, and thus its order is 2.

In short, genetic algorithms manipulate **schemata** (schemata is the plural of the word schema) when they run. If GAs use a technique that makes the probability of reproduction proportional to chromosome fitness, then according to the Schema Theorem (Holland, 1975), we can predict the presence of a given schema in the next chromosome generation. In other words, we can describe the GA's behaviour in terms of the increase or decrease in the number of instances of a given schema (Goldberg, 1989).

Let us assume that at least one instance of the schema *H* is present in the chromosome initial generation *i*. Now let $m_H(i)$ be the number of instances of the schema *H* in the generation *i*, and $\hat{f}_H(i)$ be the average fitness of these instances. We want to calculate the number of instances in the next generation, $m_H(i+1)$. As the probability of reproduction is proportional to chromosome fitness, we can easily calculate the expected number of offspring of a chromosome *x* in the next generation:

$$m_x(i+1) = \frac{f_x(i)}{\hat{f}(i)}, \tag{7.1}$$

where $f_x(i)$ is the fitness of the chromosome x, and $\hat{f}(i)$ is the average fitness of the chromosome initial generation i.

Then, assuming that the chromosome x is an instance of the schema H, we obtain

$$m_H(i+1) = \frac{\sum_{x=1}^{x=m_H(i)} f_x(i)}{\hat{f}(i)}, \quad x \in H \tag{7.2}$$

Since, by definition,

$$\hat{f}_H(i) = \frac{\sum_{x=1}^{x=m_H(i)} f_x(i)}{m_H(i)},$$

we obtain

$$m_H(i+1) = \frac{\hat{f}_H(i)}{\hat{f}(i)} m_H(i) \tag{7.3}$$

Thus, a schema with above-average fitness will indeed tend to occur more frequently in the next generation of chromosomes, and a schema with below-average fitness will tend to occur less frequently.

How about effects caused by crossover and mutation?

Crossover and mutation can both create and destroy instances of a schema. Here we will consider only destructive effects, that is effects that decrease the number of instances of the schema H. Let us first quantify the destruction caused by the crossover operator. The schema will survive after crossover if at least one of its offspring is also its instance. This is the case when crossover does not occur within the defining length of the schema.

What is the defining length of a schema?

The distance between the outermost defined bits of a schema is called **defining length**. For example, the defining length of $\boxed{*}\boxed{*}\boxed{*}\boxed{*}\boxed{1}\boxed{0}\boxed{1}\boxed{1}$ is 3, of $\boxed{*}\boxed{0}\boxed{*}\boxed{1}\boxed{*}\boxed{1}\boxed{0}\boxed{*}$ is 5 and of $\boxed{1}\boxed{*}\boxed{*}\boxed{*}\boxed{*}\boxed{*}\boxed{*}\boxed{0}$ is 7.

If crossover takes place within the defining length, the schema H can be destroyed and offspring that are not instances of H can be created. (Although the schema H will not be destroyed if two identical chromosomes cross over, even when crossover occurs within the defining length.)

Thus, the probability that the schema H will survive after crossover can be defined as:

$$P_H^{(c)} = 1 - p_c\left(\frac{l_d}{l-1}\right), \tag{7.4}$$

where p_c is the crossover probability, and l and l_d are, respectively, the length and the defining length of the schema H.

It is clear, that the probability of survival under crossover is higher for short schemata rather than for long ones.

Now consider the destructive effects of mutation. Let p_m be the mutation probability for any bit of the schema H, and n be the order of the schema H. Then $(1 - p_m)$ represents the probability that the bit will not be mutated, and thus the probability that the schema H will survive after mutation is determined as:

$$P_H^{(m)} = (1 - p_m)^n \tag{7.5}$$

It is also clear that the probability of survival under mutation is higher for low-order schemata than for high-order ones.

We can now amend Eq. (7.3) to take into account the destructive effects of crossover and mutation:

$$m_H(i+1) = \frac{\hat{f}_H(i)}{\hat{f}(i)} m_H(i)\left[1 - p_c\left(\frac{l_d}{l-1}\right)\right](1 - p_m)^n \tag{7.6}$$

This equation describes the growth of a schema from one generation to the next. It is known as the Schema Theorem. Because Eq. (7.6) considers only the destructive effects of crossover and mutation, it gives us a lower bound on the number of instances of the schema H in the next generation.

Despite crossover arguably representing a major advantage of GAs, there is as yet no theoretical basis to support the view that a GA will outperform other search and optimisation techniques just because crossover allows the combination of partial solutions.

Genetic algorithms are a very powerful tool, but need to be applied intelligently. For example, coding the problem as a bit string may change the nature of the problem being investigated. In other words, there is a danger that the coded representation becomes a problem that is different from the one we wanted to solve.

To illustrate the ideas discussed above, we consider a simple application of the GA to problems of scheduling resources.

7.5 Case study: maintenance scheduling with genetic algorithms

One of the most successful areas for GA applications includes the problem of scheduling resources. Scheduling problems are complex and difficult to solve. They are usually approached with a combination of search techniques and heuristics.

Why are scheduling problems so difficult?

First, scheduling belongs to NP-complete problems. Such problems are likely to be unmanageable and cannot be solved by combinatorial search techniques. Moreover, heuristics alone cannot guarantee the best solution.

Second, scheduling problems involve a competition for limited resources; as a result, they are complicated by many constraints. The key to the success of the GA lies in defining a fitness function that incorporates all these constraints.

The problem we discuss here is the maintenance scheduling in modern power systems. This task has to be carried out under several constraints and uncertainties, such as failures and forced outages of power equipment and delays in obtaining spare parts. The schedule often has to be revised at short notice. Human experts usually work out the maintenance scheduling by hand, and there is no guarantee that the optimum or even near-optimum schedule is produced.

A typical process of the GA development includes the following steps:

1 Specify the problem, define constraints and optimum criteria.
2 Represent the problem domain as a chromosome.
3 Define a fitness function to evaluate the chromosome's performance.
4 Construct the genetic operators.
5 Run the GA and tune its parameters.

Step 1: ***Specify the problem, define constraints and optimum criteria***

This is probably the most important step in developing a GA, because if it is not correct and complete a viable schedule cannot be obtained.

Power system components are made to operate continuously throughout their life by means of preventive maintenance. The purpose of maintenance scheduling is to find the sequence of outages of power units over a given period of time (normally a year) such that the security of a power system is maximised.

Any outage in a power system is associated with some loss in security. The security margin is determined by the system's net reserve. The net reserve, in turn, is defined as the total installed generating capacity of the system *minus* the power lost due to a scheduled outage and *minus* the maximum load forecast during the maintenance period. For instance, if we assume that the total installed capacity is 150 MW

Table 7.2 Power units and their maintenance requirements

Unit number	Unit capacity, MW	Number of intervals required for unit maintenance during one year
1	20	2
2	15	2
3	35	1
4	40	1
5	15	1
6	15	1
7	10	1

and a unit of 20 MW is scheduled for maintenance during the period when the maximum load is predicted to be 100 MW, then the net reserve will be 30 MW. Maintenance scheduling must ensure that sufficient net reserve is provided for secure power supply during any maintenance period.

Suppose, there are seven power units to be maintained in four equal intervals. The maximum loads expected during these intervals are 80, 90, 65 and 70 MW. The unit capacities and their maintenance requirements are presented in Table 7.2.

The constraints for this problem can be specified as follows:

- Maintenance of any unit starts at the beginning of an interval and finishes at the end of the same or adjacent interval. The maintenance cannot be aborted or finished earlier than scheduled.
- The net reserve of the power system must be greater than or equal to zero at any interval.

The optimum criterion here is that the net reserve must be at the maximum during any maintenance period.

Step 2: ***Represent the problem domain as a chromosome***

Our scheduling problem is essentially an ordering problem, requiring us to list the tasks in a particular order. A complete schedule may consist of a number of overlapping tasks, but not all orderings are legal, since they may violate the constraints. Our job is to represent a complete schedule as a chromosome of a fixed length.

An obvious coding scheme that comes to mind is to assign each unit a binary number and to let the chromosome be a sequence of these binary numbers. However, an ordering of the units in a sequence is not yet a schedule. Some units can be maintained simultaneously, and we must also incorporate the time required for unit maintenance into the schedule. Thus, rather than ordering units in a sequence, we might build a sequence of maintenance schedules of individual units. The unit schedule can be easily represented as a 4-bit string, where each bit is a maintenance interval. If a unit is to be maintained in a particular interval, the corresponding bit assumes value 1, otherwise it is 0. For example, the string $\boxed{0|1|0|0}$ presents a schedule for a unit to be maintained in the second interval. It also shows that the number of intervals required for maintenance of this unit is equal to 1. Thus, a complete maintenance schedule for our problem can be represented as a 28-bit chromosome.

However, crossover and mutation operators could easily create binary strings that call for maintaining some units more than once and others not at all. In addition, we could call for maintenance periods that would exceed the number of intervals really required for unit maintenance.

A better approach is to change the chromosome syntax. As already discussed, a chromosome is a collection of elementary parts called genes. Traditionally, each gene is represented by only one bit and cannot be broken into smaller elements. For our problem, we can adopt the same concept, but represent a gene by four bits. In other words, the smallest indivisible part of our chromosome is a 4-bit string. This representation allows crossover and mutation operators to act according to the theoretical grounding of genetic algorithms. What remains to be done is to produce a pool of genes for each unit:

Unit 1:	1 1 0 0	0 1 1 0	0 0 1 1	
Unit 2:	1 1 0 0	0 1 1 0	0 0 1 1	
Unit 3:	1 0 0 0	0 1 0 0	0 0 1 0	0 0 0 1
Unit 4:	1 0 0 0	0 1 0 0	0 0 1 0	0 0 0 1
Unit 5:	1 0 0 0	0 1 0 0	0 0 1 0	0 0 0 1
Unit 6:	1 0 0 0	0 1 0 0	0 0 1 0	0 0 0 1
Unit 7:	1 0 0 0	0 1 0 0	0 0 1 0	0 0 0 1

The GA can now create an initial population of chromosomes by filling 7-gene chromosomes with genes randomly selected from the corresponding pools. A sample of such a chromosome is shown in Figure 7.9.

Step 3: ***Define a fitness function to evaluate the chromosome performance***

The chromosome evaluation is a crucial part of the GA, because chromosomes are selected for mating based on their fitness. The fitness function must capture what makes a maintenance schedule either good or bad for the user. For our problem we apply a fairly simple function concerned with constraint violations and the net reserve at each interval.

Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6	Unit 7
0 1 1 0	0 0 1 1	0 0 0 1	1 0 0 0	0 1 0 0	0 0 1 0	1 0 0 0

Figure 7.9 A chromosome for the scheduling problem

The evaluation of a chromosome starts with the sum of capacities of the units scheduled for maintenance at each interval. For the chromosome shown in Figure 7.9, we obtain:

$$\begin{aligned}
&\textit{Interval}\ 1: && 0\times 20+0\times 15+0\times 35+1\times 40+0\times 15+0\times 15+1\times 10\\
& && =50\\
&\textit{Interval}\ 2: && 1\times 20+0\times 15+0\times 35+0\times 40+1\times 15+0\times 15+0\times 10\\
& && =35\\
&\textit{Interval}\ 3: && 1\times 20+1\times 15+0\times 35+0\times 40+0\times 15+1\times 15+0\times 10\\
& && =50\\
&\textit{Interval}\ 4: && 0\times 20+1\times 15+1\times 35+0\times 40+0\times 15+0\times 15+0\times 10\\
& && =50
\end{aligned}$$

Then these values are subtracted from the total installed capacity of the power system (in our case, 150 MW):

$$\begin{aligned}
&\textit{Interval}\ 1: && 150-50=100\\
&\textit{Interval}\ 2: && 150-35=115\\
&\textit{Interval}\ 3: && 150-50=100\\
&\textit{Interval}\ 4: && 150-50=100
\end{aligned}$$

And finally, by subtracting the maximum loads expected at each interval, we obtain the respective net reserves:

$$\begin{aligned}
&\textit{Interval}\ 1: && 100-80=20\\
&\textit{Interval}\ 2: && 115-90=25\\
&\textit{Interval}\ 3: && 100-65=35\\
&\textit{Interval}\ 4: && 100-70=30
\end{aligned}$$

Since all the results are positive, this particular chromosome does not violate any constraint, and thus represents a legal schedule. The chromosome's fitness is determined as the lowest of the net reserves; in our case it is 20.

If, however, the net reserve at any interval is negative, the schedule is illegal, and the fitness function returns zero.

At the beginning of a run, a randomly built initial population might consist of all illegal schedules. In this case, chromosome fitness values remain unchanged, and selection takes place in accordance with the actual fitness values.

Step 4: ***Construct the genetic operators***

Constructing genetic operators is challenging and we must experiment to make crossover and mutation work correctly. The chromosome has to be broken up in a way that is legal for our problem. Since we have already changed the chromosome syntax for this, we can use the GA operators in their classical forms. Each gene in a chromosome is represented by a 4-bit indivisible string, which consists of a possible maintenance schedule for a particular unit. Thus, any random mutation

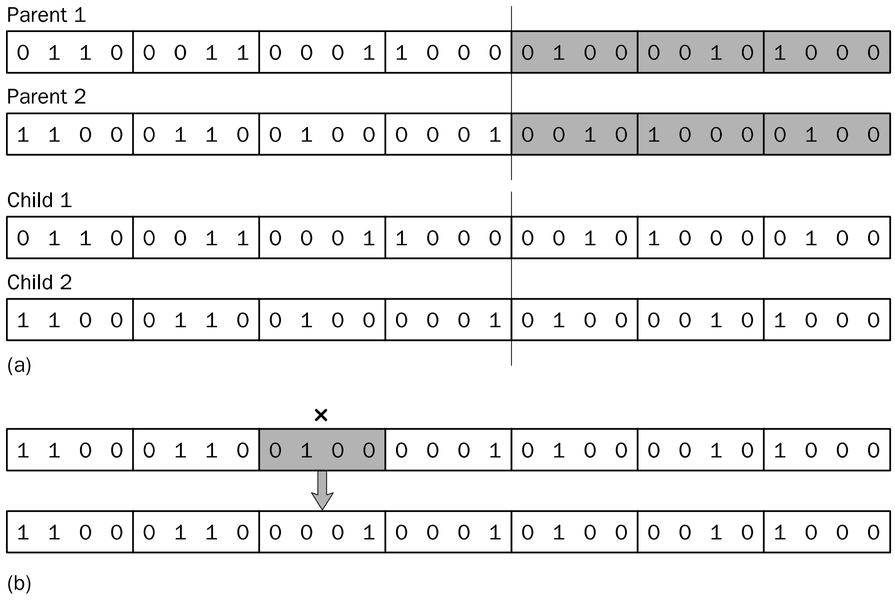


Figure 7.10 Genetic operators for the scheduling problem: (a) the crossover operator; (b) the mutation operator

of a gene or recombination of several genes from two parent chromosomes may result only in changes of the maintenance schedules for individual units, but cannot create 'unnatural' chromosomes.

Figure 7.10(a) shows an example of the crossover application during a run of the GA. The children are made by cutting the parents at the randomly selected point denoted by the vertical line and exchanging parental genes after the cut. Figure 7.10(b) demonstrates an example of mutation. The mutation operator randomly selects a 4-bit gene in a chromosome and replaces it by a gene randomly selected from the corresponding pool. In the example shown in Figure 7.10(b), the chromosome is mutated in its third gene, which is replaced by the gene $\boxed{0}\boxed{0}\boxed{0}\boxed{1}$ chosen from the pool of genes for the Unit 3.

Step 5: ***Run the GA and tune its parameters***

It is time to run the GA. First, we must choose the population size and the number of generations to be run. Common sense suggests that a larger population can achieve better solutions than a smaller one, but will work more slowly. In fact, however, the most effective population size depends on the problem being solved, particularly on the problem coding scheme (Goldberg, 1989). The GA can run only a finite number of generations to obtain a solution. Perhaps we could choose a very large population and run it only once, or we could choose a smaller population and run it several times. In any case, only experimentation can give us the answer.

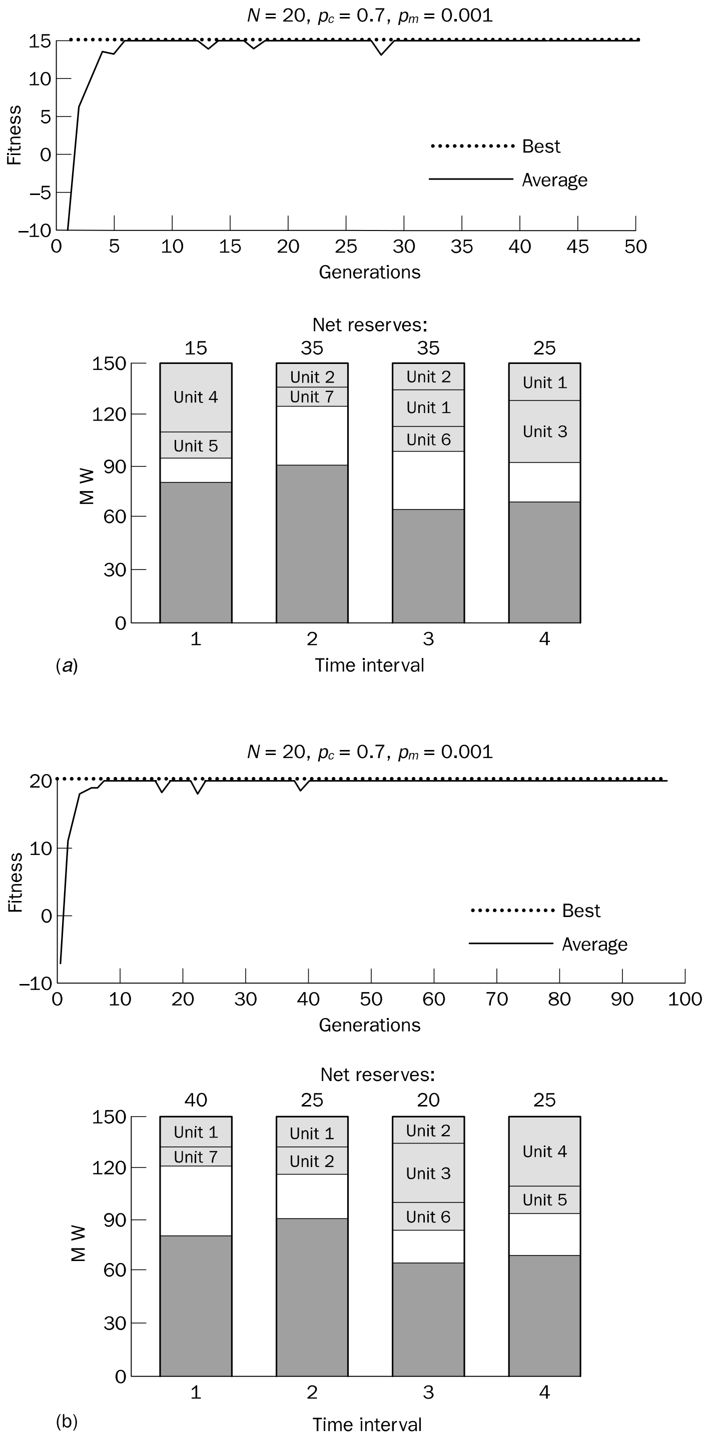
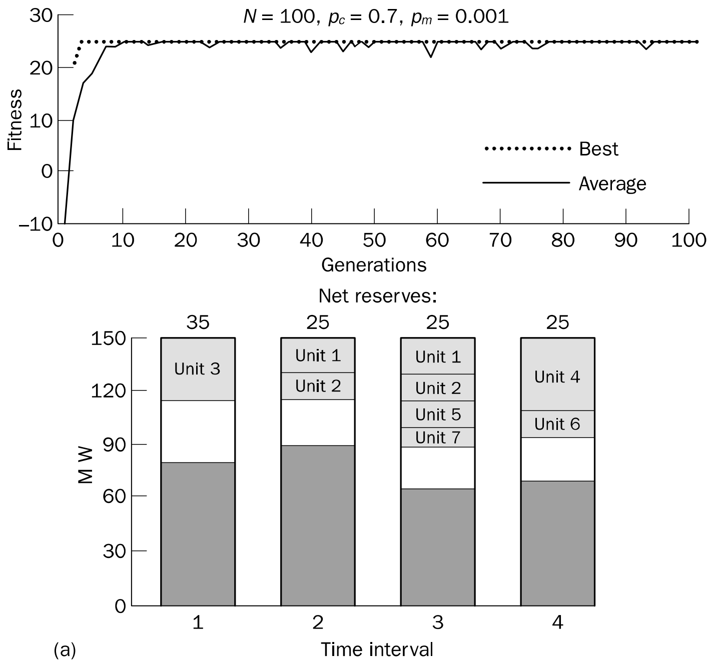


Figure 7.11 Performance graphs and the best maintenance schedules created in a population of 20 chromosomes: (a) 50 generations; (b) 100 generations



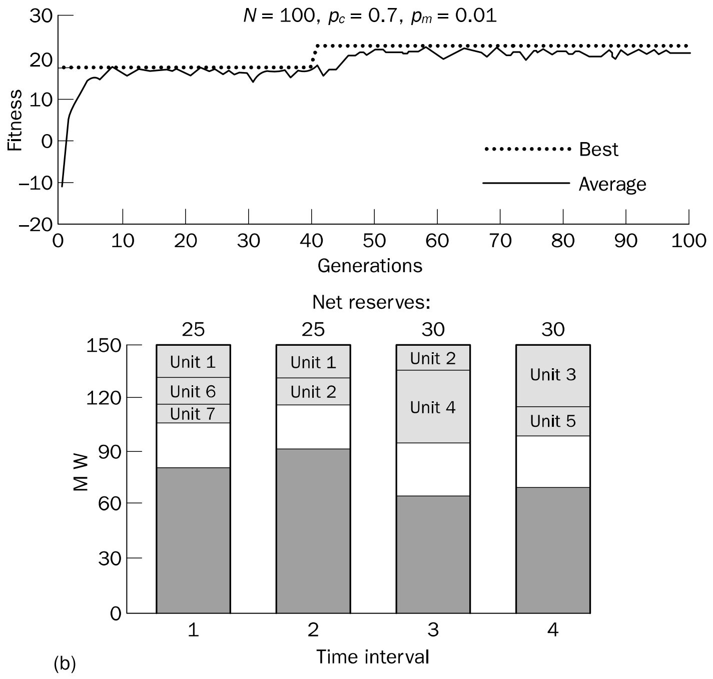


Figure 7.12 Performance graphs and the best maintenance schedules created in a population of 100 chromosomes: (a) mutation rate is 0.001; (b) mutation rate is 0.01

Figure 7.11(a) presents performance graphs and the best schedule created by 50 generations of 20 chromosomes. As you can see, the minimum of the net reserves for the best schedule is 15 MW. Let us increase the number of generations to 100 and compare the best schedules. Figure 7.11(b) presents the results. The best schedule now provides the minimum net reserve of 20 MW. However, in both cases, the best individuals appeared in the initial generation, and the increasing number of generations did not affect the final solution. It indicates that we should try increasing the population size.

Figure 7.12(a) shows fitness function values across 100 generations, and the best schedule so far. The minimum net reserve has increased to 25 MW. To make sure of the quality of the best-so-far schedule, we must compare results obtained under different rates of mutation. Thus, let us increase the mutation rate to 0.01 and rerun the GA once more. Figure 7.12(b) presents the results. The minimum net reserve is still 25 MW. Now we can confidently argue that the optimum solution has been found.

7.6 Evolution strategies

Another approach to simulating natural evolution was proposed in Germany in the early 1960s. Unlike genetic algorithms, this approach – called an **evolution strategy** – was designed to solve technical optimisation problems.

In 1963 two students of the Technical University of Berlin, Ingo Rechenberg and Hans-Paul Schwefel, were working on the search for the optimal shapes of bodies in a flow. In their work, they used the wind tunnel of the Institute of Flow Engineering. Because it was then a matter of laborious intuitive experimentation, they decided to try random changes in the parameters defining the shape following the example of natural mutation. As a result, the evolution strategy was born (Rechenberg, 1965; Schwefel, 1981).

Evolution strategies were developed as an alternative to the engineer's intuition. Until recently, evolution strategies were used in technical optimisation problems when no analytical objective function was available, and no conventional optimisation method existed, thus engineers had to rely only on their intuition.

Unlike GAs, evolution strategies use only a mutation operator.

How do we implement an evolution strategy?

In its simplest form, termed as a $(1 + 1)$-evolution strategy, one parent generates one offspring per generation by applying **normally distributed** mutation. The $(1 + 1)$-evolution strategy can be implemented as follows:

Step 1: Choose the number of parameters N to represent the problem, and then determine a feasible range for each parameter:

$$\{x_{1min}, x_{1max}\}, \{x_{2min}, x_{2max}\}, \ldots, \{x_{Nmin}, x_{Nmax}\},$$

Define a standard deviation for each parameter and the function to be optimised.

Step 2: Randomly select an initial value for each parameter from the respective feasible range. The set of these parameters will constitute the initial population of parent parameters:

$$x_1, x_2, \ldots, x_N$$

Step 3: Calculate the solution associated with the parent parameters:

$$X = f(x_1, x_2, \ldots, x_N)$$

Step 4: Create a new (offspring) parameter by adding a normally distributed random variable a with mean zero and pre-selected deviation δ to each parent parameter:

$$x'_i = x_i + a(0, \delta), \qquad i = 1, 2, \ldots, N \tag{7.7}$$

Normally distributed mutations with mean zero reflect the natural process of evolution where smaller changes occur more frequently than larger ones.

Step 5: Calculate the solution associated with the offspring parameters:

$$X' = f(x'_1, x'_2, \ldots, x'_N)$$

Step 6: Compare the solution associated with the offspring parameters with the one associated with the parent parameters. If the solution for the offspring is better than that for the parents, replace the parent population with the offspring population. Otherwise, keep the parent parameters.

Step 7: Go to Step 4, and repeat the process until a satisfactory solution is reached, or a specified number of generations is considered.

The $(1 + 1)$-evolution strategy can be represented as a block-diagram shown in Figure 7.13.

Why do we vary all the parameters simultaneously when generating a new solution?

An evolution strategy here reflects the nature of a chromosome. In fact, a single gene may simultaneously affect several characteristics of the living organism. On the other hand, a single characteristic of an individual may be determined by the simultaneous interactions of several genes. The natural selection acts on a collection of genes, not on a single gene in isolation.

Evolution strategies can solve a wide range of constrained and unconstrained non-linear optimisation problems and produce better results than many conventional, highly complex, non-linear optimisation techniques (Schwefel, 1995). Experiments also suggest that the simplest version of evolution strategies that uses a single parent – single offspring search works best.

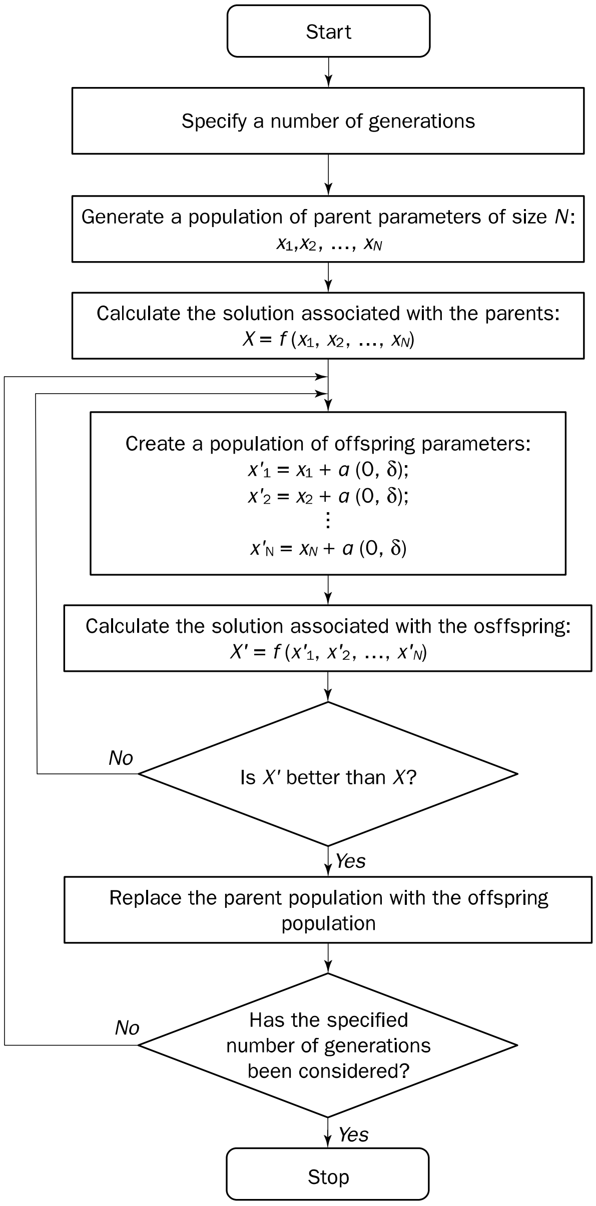

Start
Specify a number of generations
Generate a population of parent parameters of size N: $x_1, x_2, \ldots, x_N$
Calculate the solution associated with the parents: $X = f(x_1, x_2, \ldots, x_N)$
Create a population of offspring parameters:
$x'_1 = x_1 + a(0, \delta)$;
$x'_2 = x_2 + a(0, \delta)$;
$\vdots$
$x'_N = x_N + a(0, \delta)$
Calculate the solution associated with the osffspring: $X' = f(x'_1, x'_2, \ldots, x'_N)$
Is X' better than X?
No
Yes
Replace the parent population with the offspring population
Has the specified number of generations been considered?
No
Yes
Stop

Figure 7.13 Block-diagram of the $(1 + 1)$-evolution strategy

What are the differences between genetic algorithms and evolution strategies?

The principal difference between a GA and an evolution strategy is that the former uses both crossover and mutation whereas the latter uses only mutation. In addition, when we use an evolution strategy we do not need to represent the problem in a coded form.

Which method works best?

An evolution strategy uses a purely numerical optimisation procedure, similar to a focused Monte Carlo search. GAs are capable of more general applications, but the hardest part of applying a GA is coding the problem. In general, to answer the question as to which method works best, we have to experiment to find out. It is application-dependent.

7.7 Genetic programming

One of the central problems in computer science is how to make computers solve problems without being explicitly programmed to do so. Genetic programming offers a solution through the evolution of computer programs by methods of natural selection. In fact, genetic programming is an extension of the conventional genetic algorithm, but the goal of genetic programming is not just to evolve a bit-string representation of some problem but the computer code that solves the problem. In other words, genetic programming creates computer programs as the solution, while GAs create a string of binary numbers that represent the solution.

Genetic programming is a recent development in the area of evolutionary computation. It was greatly stimulated in the 1990s by John Koza (Koza, 1992, 1994).

How does genetic programming work?

According to Koza, genetic programming searches the space of possible computer programs for a program that is highly fit for solving the problem at hand (Koza, 1992).

Any computer program is a sequence of operations (functions) applied to values (arguments), but different programming languages may include different types of statements and operations, and have different syntactic restrictions. Since genetic programming manipulates programs by applying genetic operators, a programming language should permit a computer program to be manipulated as data and the newly created data to be executed as a program. For these reasons, LISP was chosen as the main language for genetic programming (Koza, 1992).

What is LISP?

LISP, or **List Processor**, is one of the oldest high-level programming languages (FORTRAN is just two years older than LISP). LISP, which was written by John McCarthy in the late 1950s, has become one of the standard languages for artificial intelligence.

LISP has a highly symbol-oriented structure. Its basic data structures are **atoms** and **lists**. An atom is the smallest indivisible element of the LISP syntax. The number *21*, the symbol *X* and the string *'This is a string'* are examples of LISP atoms. A list is an object composed of atoms and/or other lists. LISP lists are written as an ordered collection of items inside a pair of parentheses. For example, the list

(−(∗ A B) C)

calls for the application of the subtraction function (−) to two arguments, namely the list (∗ A B) and the atom C. First, LISP applies the multiplication function (∗) to the atoms A and B. Once the list (∗ A B) is evaluated, LISP applies the subtraction function (−) to the two arguments, and thus evaluates the entire list (−(∗ A B) C).

Both atoms and lists are called symbolic expressions or **S-expressions**. In LISP, all data and all programs are S-expressions. This gives LISP the ability to operate on programs as if they were data. In other words, LISP programs can modify themselves or even write other LISP programs. This remarkable property of LISP makes it very attractive for genetic programming.

Any LISP S-expression can be depicted as a rooted point-labelled tree with ordered branches. Figure 7.14 shows the tree corresponding to the S-expression (−(∗ A B) C). This tree has five points, each of which represents either a function or a terminal. The two internal points of the tree are labelled with functions (−) and (∗). Note that the root of the tree is the function appearing just inside the leftmost opening parenthesis of the S-expression. The three external points of the tree, also called leaves, are labelled with terminals A, B and C. In the graphical representation, the branches are ordered because the order of the arguments in many functions directly affects the results.

How do we apply genetic programming to a problem?

Before applying genetic programming to a problem, we must accomplish **five preparatory steps** (Koza, 1994):

1 Determine the set of terminals.

2 Select the set of primitive functions.

3 Define the fitness function.

4 Decide on the parameters for controlling the run.

5 Choose the method for designating a result of the run.

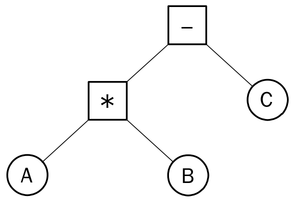


Figure 7.14 Graphical representation of the LISP S-expression (−(∗ A B) C)

Table 7.3 Ten fitness cases for the Pythagorean Theorem

Side *a*	**Side** *b*	**Hypotenuse** *c*	**Side** *a*	**Side** *b*	**Hypotenuse** *c*
3	5	5.830952	12	10	15.620499
8	14	16.124515	21	6	21.840330
18	2	18.110770	7	4	8.062258
32	11	33.837849	16	24	28.844410
4	3	5.000000	2	9	9.219545

The Pythagorean Theorem helps us to illustrate these preparatory steps and demonstrate the potential of genetic programming. The theorem says that the hypotenuse, c, of a right triangle with short sides a and b is given by

$$c = \sqrt{a^2 + b^2}.$$

The aim of genetic programming is to discover a program that matches this function. To measure the performance of the as-yet-undiscovered computer program, we will use a number of different **fitness cases**. The fitness cases for the Pythagorean Theorem are represented by the samples of right triangles in Table 7.3. These fitness cases are chosen at random over a range of values of variables a and b.

Step 1: ***Determine the set of terminals***
The terminals correspond to the inputs of the computer program to be discovered. Our program takes two inputs, a and b.

Step 2: ***Select the set of primitive functions***
The functions can be presented by standard arithmetic operations, standard programming operations, standard mathematical functions, logical functions or domain-specific functions. Our program will use four standard arithmetic operations $+$, $-$, $*$ and /, and one mathematical function *sqrt*.

Terminals and primitive functions together constitute the building blocks from which genetic programming constructs a computer program to solve the problem.

Step 3: ***Define the fitness function***
A fitness function evaluates how well a particular computer program can solve the problem. The choice of the fitness function depends on the problem, and may vary greatly from one problem to the next. For our problem, the fitness of the computer program can be measured by the error between the actual result produced by the program and the correct result given by the fitness case. Typically, the error is not measured over just one fitness case, but instead calculated as a sum of the absolute errors over a number of fitness cases. The closer this sum is to zero, the better the computer program.

Step 4: ***Decide on the parameters for controlling the run***
For controlling a run, genetic programming uses the same primary parameters as those used for GAs. They include the population size and the maximum number of generations to be run.

Step 5: ***Choose the method for designating a result of the run***
It is common practice in genetic programming to designate the best-so-far generated program as the result of a run.

Once these five steps are complete, a run can be made. The run of genetic programming starts with a random generation of an initial population of computer programs. Each program is composed of functions $+$, $-$, $*$, / and *sqrt*, and terminals a and b.

In the initial population, all computer programs usually have poor fitness, but some individuals are more fit than others. Just as a fitter chromosome is more likely to be selected for reproduction, so a fitter computer program is more likely to survive by copying itself into the next generation.

Is the crossover operator capable of operating on computer programs?

In genetic programming, the crossover operator operates on two computer programs which are selected on the basis of their fitness. These programs can have different sizes and shapes. The two offspring programs are composed by recombining randomly chosen parts of their parents. For example, consider the following two LISP S-expressions:

$$(/\ (-\ (sqrt\ (+\ (*\ a\ a)\ (-\ a\ b)))\ a)\ (*\ a\ b)),$$

which is equivalent to

$$\frac{\sqrt{a^2 + (a - b)} - a}{ab},$$

and

$$(+\ (-\ (sqrt\ (-\ (*\ b\ b)\ a))\ b)\ (sqrt\ (/\ a\ b))),$$

which is equivalent to

$$\left(\sqrt{b^2 - a} - b\right) + \sqrt{\frac{a}{b}}.$$

These two S-expressions can be presented as rooted, point-labelled trees with ordered branches as shown in Figure 7.15(a). Internal points of the trees correspond to functions and external points correspond to terminals.

Any point, internal or external, can be chosen as a crossover point. Suppose that the crossover point for the first parent is the function $(*)$, and the crossover

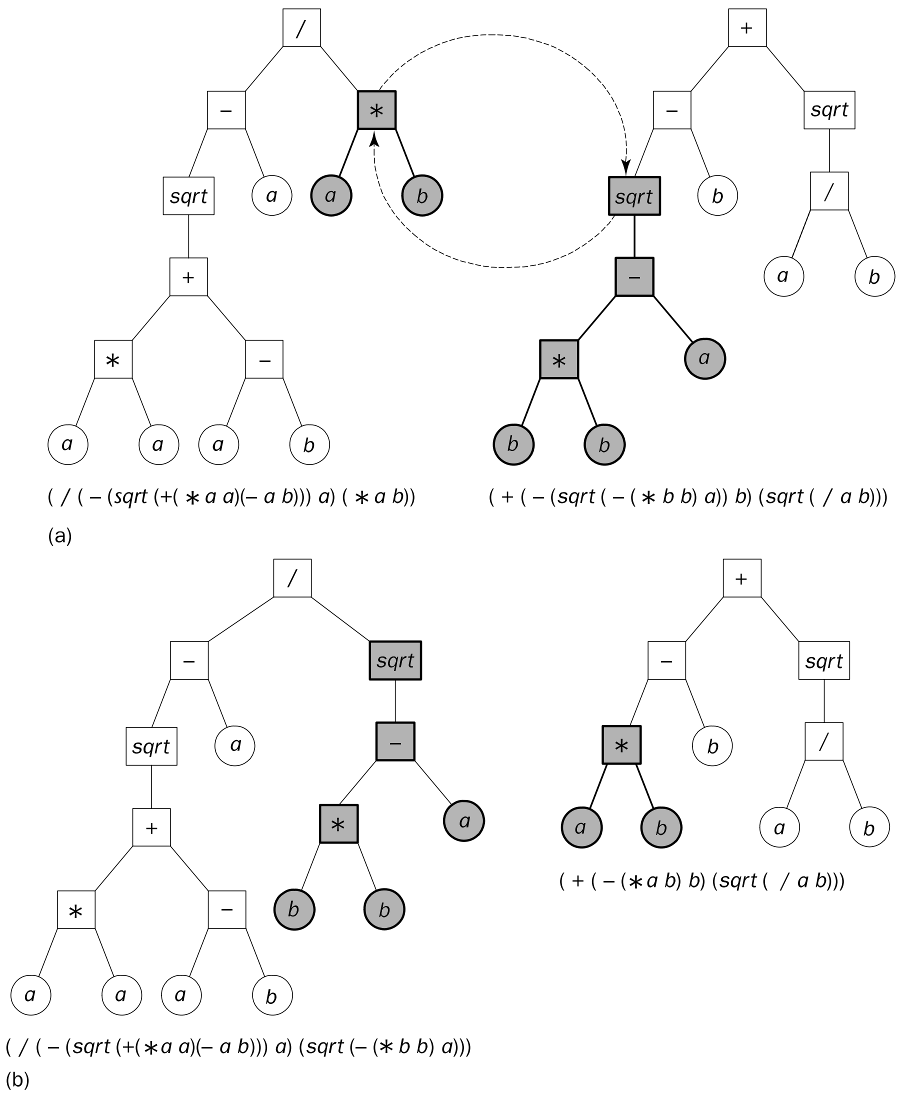

(/ (– (sqrt (+(∗ a a)(– a b))) a) (∗ a b))
(+ (– (sqrt (– (∗ b b) a)) b) (sqrt (/ a b)))
(a)
(/ (– (sqrt (+(∗a a)(– a b))) a) (sqrt (– (∗ b b) a)))
(+ (– (∗a b) b) (sqrt (/ a b)))
(b)

Figure 7.15 Crossover in genetic programming: (a) two parental S-expressions; (b) two offspring S-expressions

point for the second parent is the function *sqrt*. As a result, we obtain the two **crossover fragments** rooted at the chosen crossover points as shown in Figure 7.15(a). The crossover operator creates two offspring by exchanging the crossover fragments of two parents. Thus, the first offspring is created by inserting the crossover fragment of the second parent into the place of the crossover fragment of the first parent. Similarly, the second offspring is created by inserting the crossover fragment of the first parent into the place of the crossover fragment of the second parent. The two offspring resulting from crossover of the two parents are shown in Figure 7.15(b). These offspring are equivalent to

$$\frac{\sqrt{a^2 + (a - b)} - a}{\sqrt{b^2 - a}} \quad \text{and} \quad (ab - b) + \sqrt{\frac{a}{b}}.$$

The crossover operator produces valid offspring computer programs regardless of the choice of crossover points.

Is mutation used in genetic programming?

A mutation operator can randomly change any function or any terminal in the LISP S-expression. Under mutation, a function can only be replaced by a function and a terminal can only be replaced by a terminal. Figure 7.16 explains the basic concept of mutation in genetic programming.

In summary, genetic programming creates computer programs by executing the following steps (Koza, 1994):

Step 1: Assign the maximum number of generations to be run and probabilities for cloning, crossover and mutation. Note that the sum of the probability of cloning, the probability of crossover and the probability of mutation must be equal to one.

Step 2: Generate an initial population of computer programs of size *N* by combining randomly selected functions and terminals.

Step 3: Execute each computer program in the population and calculate its fitness with an appropriate fitness function. Designate the best-so-far individual as the result of the run.

Step 4: With the assigned probabilities, select a genetic operator to perform cloning, crossover or mutation.

Step 5: If the cloning operator is chosen, select one computer program from the current population of programs and copy it into a new population.

If the crossover operator is chosen, select a pair of computer programs from the current population, create a pair of offspring programs and place them into the new population.

If the mutation operator is chosen, select one computer program from the current population, perform mutation and place the mutant into the new population.

All programs are selected with a probability based on their fitness (i.e., the higher the fitness, the more likely the program is to be selected).

Step 6: Repeat Step 4 until the size of the new population of computer programs becomes equal to the size of the initial population, *N*.

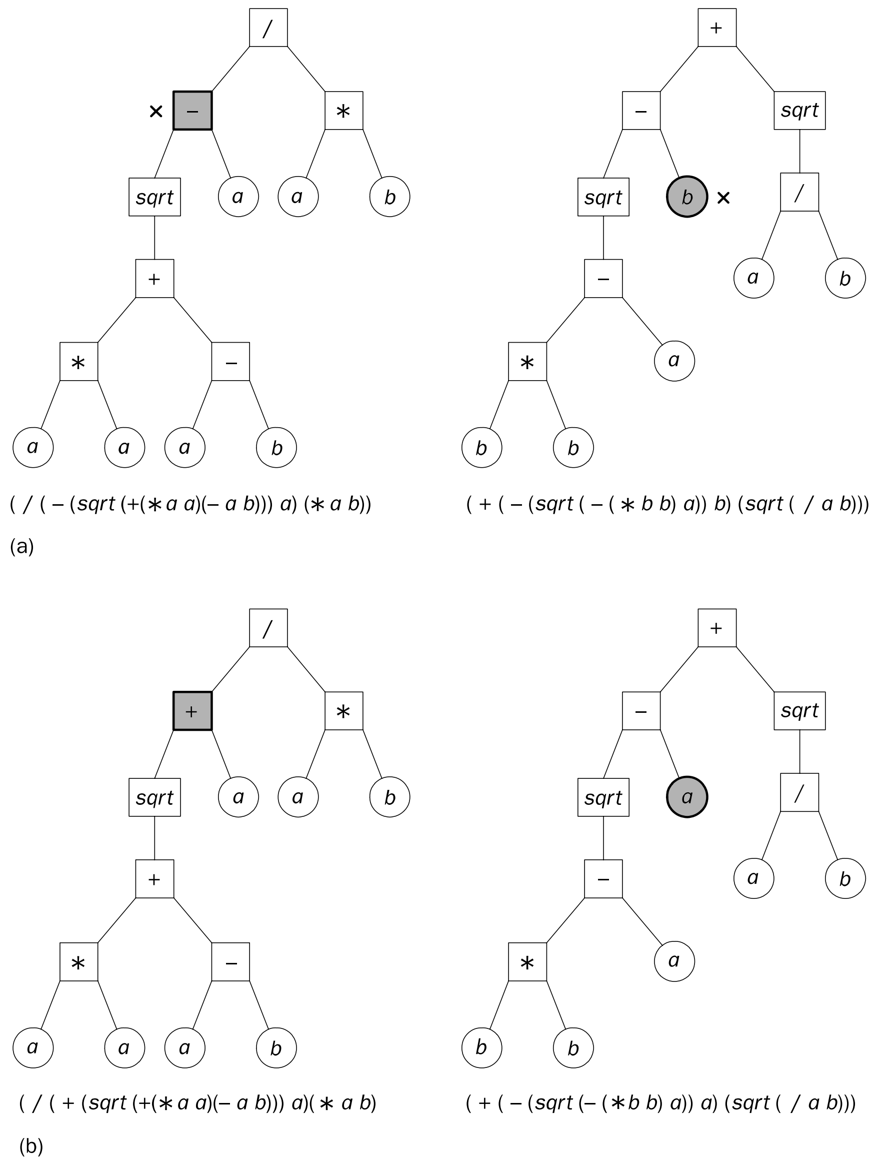

(/ (− (sqrt (+(∗ a a)(− a b))) a) (∗ a b))
(a)
(+ (− (sqrt (− (∗ b b) a)) b) (sqrt (/ a b)))
(/ (+ (sqrt (+(∗ a a)(− a b))) a)(∗ a b)
(b)
(+ (− (sqrt (− (∗b b) a)) a) (sqrt (/ a b)))

Figure 7.16 Mutation in genetic programming: (a) original S-expressions; (b) mutated S-expressions

Step 7: Replace the current (parent) population with the new (offspring) population.

Step 8: Go to Step 3 and repeat the process until the termination criterion is satisfied.

Figure 7.17 is a flowchart representing the above steps of genetic programming.

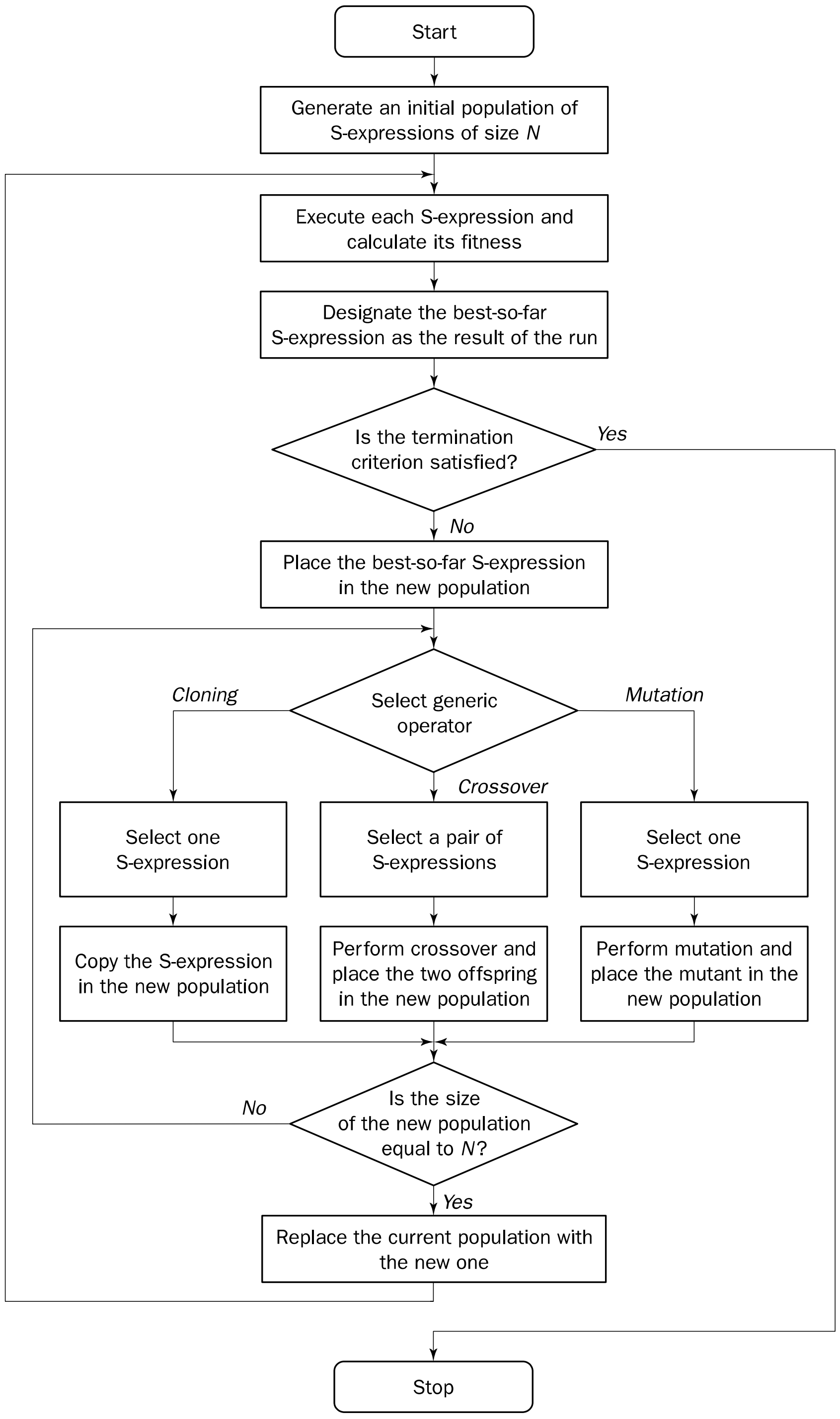

Start
Generate an initial population of S-expressions of size *N*
Execute each S-expression and calculate its fitness
Designate the best-so-far S-expression as the result of the run
Is the termination criterion satisfied?
Yes
No
Place the best-so-far S-expression in the new population
Select generic operator
Cloning
Crossover
Mutation
Select one S-expression
Select a pair of S-expressions
Select one S-expression
Copy the S-expression in the new population
Perform crossover and place the two offspring in the new population
Perform mutation and place the mutant in the new population
Is the size of the new population equal to *N*?
No
Yes
Replace the current population with the new one
Stop

Figure 7.17 Flowchart for genetic programming

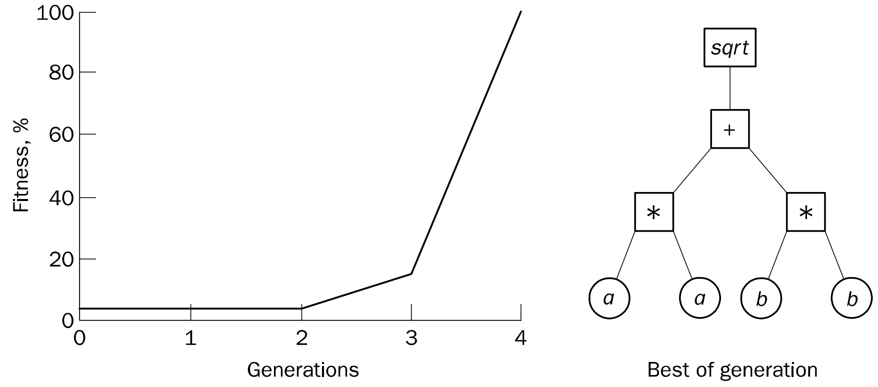


Figure 7.18 Fitness history of the best S-expression

Let us now return to the Pythagorean Theorem. Figure 7.18 shows the fitness history of the best S-expression in a population of 500 computer programs. As you can see, in the randomly generated initial population, even the best S-expression has very poor fitness. But fitness improves very rapidly, and at the fourth generation the correct S-expression is reproduced. This simple example demonstrates that genetic programming offers a general and robust method of evolving computer programs.

In the Pythagorean Theorem example, we used LISP S-expressions but there is no reason to restrict genetic programming only to LISP S-expressions. It can also be implemented in C, C++, Pascal, FORTRAN, Mathematica, Smalltalk and other programming languages, and can be applied more generally.

What are the main advantages of genetic programming compared to genetic algorithms?

Genetic programming applies the same evolutionary approach as a GA does. However, genetic programming is no longer breeding bit strings that represent coded solutions but complete computer programs that solve a particular problem. The fundamental difficulty of GAs lies in the problem representation, that is, in the fixed-length coding. A poor representation limits the power of a GA, and even worse, may lead to a false solution.

A fixed-length coding is rather artificial. As it cannot provide a dynamic variability in length, such a coding often causes considerable redundancy and reduces the efficiency of genetic search. In contrast, genetic programming uses high-level building blocks of variable length. Their size and complexity can change during breeding. Genetic programming works well in a large number of different cases (Koza, 1994) and has many potential applications.

Are there any difficulties?

Despite many successful applications, there is still no proof that genetic programming will scale up to more complex problems that require larger computer programs. And even if it scales up, extensive computer run times may be needed.

7.8 Summary

In this chapter, we presented an overview of evolutionary computation. We considered genetic algorithms, evolution strategies and genetic programming. We introduced the main steps in developing a genetic algorithm, discussed why genetic algorithms work, and illustrated the theory through actual applications of genetic algorithms. Then we presented a basic concept of evolutionary strategies and determined the differences between evolutionary strategies and genetic algorithms. Finally, we considered genetic programming and its application to real problems.

The most important lessons learned in this chapter are:

- The evolutionary approach to artificial intelligence is based on the computational models of natural selection and genetics known as evolutionary computation. Evolutionary computation combines genetic algorithms, evolution strategies and genetic programming.
- All methods of evolutionary computation work as follows: create a population of individuals, evaluate their fitness, generate a new population by applying genetic operators, and repeat this process a number of times.
- Genetic algorithms were invented by John Holland in the early 1970s. Holland's genetic algorithm is a sequence of procedural steps for moving from one generation of artificial 'chromosomes' to another. It uses 'natural' selection and genetics-inspired techniques known as crossover and mutation. Each chromosome consists of a number of 'genes', and each gene is represented by 0 or 1.
- Genetic algorithms use fitness values of individual chromosomes to carry out reproduction. As reproduction takes place, the crossover operator exchanges parts of two single chromosomes, and the mutation operator changes the gene value in some randomly chosen location of the chromosome. After a number of successive reproductions, the less fit chromosomes become extinct, while those best fit gradually come to dominate the population.
- Genetic algorithms work by discovering and recombining schemata – good 'building blocks' of candidate solutions. The genetic algorithm does not need knowledge of the problem domain, but it requires the fitness function to evaluate the fitness of a solution.
- Solving a problem using genetic algorithms involves defining constraints and optimum criteria, encoding the problem solutions as chromosomes, defining a fitness function to evaluate a chromosome's performance, and creating appropriate crossover and mutation operators.
- Genetic algorithms are a very powerful tool. However, coding the problem as a bit string may change the nature of the problem being investigated. There is always a danger that the coded representation represents a problem that is different from the one we want to solve.

- Evolution strategies were developed by Ingo Rechenberg and Hans-Paul Schwefel in the early 1960s as an alternative to the engineer's intuition. Evolution strategies are used in technical optimisation problems when no analytical objective function is available, and no conventional optimisation method exists – only the engineer's intuition.
- An evolution strategy is a purely numerical optimisation procedure that is similar to a focused Monte Carlo search. Unlike genetic algorithms, evolution strategies use only a mutation operator. In addition, the representation of a problem in a coded form is not required.
- Genetic programming is a recent development in the area of evolutionary computation. It was greatly stimulated in the 1990s by John Koza. Genetic programming applies the same evolutionary approach as genetic algorithms. However, genetic programming is no longer breeding bit strings that represent coded solutions but complete computer programs that solve a problem at hand.
- Solving a problem by genetic programming involves determining the set of arguments, selecting the set of functions, defining a fitness function to evaluate the performance of created computer programs, and choosing the method for designating a result of the run.
- Since genetic programming manipulates programs by applying genetic operators, a programming language should permit a computer program to be manipulated as data and the newly created data to be executed as a program. For these reasons, LISP was chosen as the main language for genetic programming.

Questions for review

1 Why are genetic algorithms called genetic? Who was the 'father' of genetic algorithms?

2 What are the main steps of a genetic algorithm? Draw a flowchart that implements these steps. What are termination criteria used in genetic algorithms?

3 What is the roulette wheel selection technique? How does it work? Give an example.

4 How does the crossover operator work? Give an example using fixed-length bit strings. Give another example using LISP S-expressions.

5 What is mutation? Why is it needed? How does the mutation operator work? Give an example using fixed-length bit strings. Give another example using LISP S-expressions.

6 Why do genetic algorithms work? What is a schema? Give an example of a schema and its instances. Explain the relationship between a schema and a chromosome. What is the Schema Theorem?

7 Describe a typical process of the development of a genetic algorithm for solving a real problem. What is the fundamental difficulty of genetic algorithms?

8 What is an evolution strategy? How is it implemented? What are the differences between evolution strategies and genetic algorithms?

9 Draw a block-diagram of the $(1+1)$ evolution strategy. Why do we vary all the parameters simultaneously when generating a new solution?

10 What is genetic programming? How does it work? Why has LISP become the main language for genetic programming?

11 What is a LISP S-expression? Give an example and represent it as a rooted point-labelled tree with ordered branches. Show terminals and functions on the tree.

12 What are the main steps in genetic programming? Draw a flowchart that implements these steps. What are advantages of genetic programming?

References

Atmar, W. (1994). Notes on the simulation of evolution, *IEEE Transactions on Neural Networks*, 5(1), 130–148.

Caruana, R.A. and Schaffer, J.D. (1988). Representation and hidden bias: gray vs. binary coding for genetic algorithms, *Proceedings of the Fifth International Conference on Machine Learning*, J. Laird, ed., Morgan Kaufmann, San Mateo, CA.

Davis, L. (1991). *Handbook on Genetic Algorithms*. Van Nostrand Reinhold, New York.

Fogel, L.J., Owens, A.J. and Walsh, M.J. (1966). *Artificial Intelligence Through Simulated Evolution*. Morgan Kaufmann, Los Altos, CA.

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison-Wesley, Reading, MA.

Hartl, D.L. and Clark, A.G. (1989). *Principles of Population Genetics*, 2nd edn. Sinauer, Sunderland, MA.

Hoffman, A. (1989). *Arguments on Evolution: A Paleontologist's Perspective*. Oxford University Press, New York.

Holland, J.H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor.

Keeton, W.T. (1980). *Biological Science*, 3rd edn. W.W. Norton, New York.

Koza, J.R. (1992). *Genetic Programming: On the Programming of the Computers by Means of Natural Selection*. MIT Press, Cambridge, MA.

Koza, J.R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA.

Mayr, E. (1988). *Towards a New Philosophy of Biology: Observations of an Evolutionist*. Belknap Press, Cambridge, MA.

Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolutionary Programs*, 3rd edn. Springer-Verlag, New York.

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA.

Rawlins, G. (1991). *Foundations of Genetic Algorithms*. Morgan Kaufmann, San Francisco, CA.

Rechenberg, I. (1965). *Cybernetic Solution Path of an Experimental Problem*. Ministry of Aviation, Royal Aircraft Establishment, Library Translation No. 1122, August.

Schwefel, H.-P. (1981). *Numerical Optimization of Computer Models*. John Wiley, Chichester.

Schwefel, H.-P. (1995). *Evolution and Optimum Seeking*. John Wiley, New York.

Turing, A.M. (1950). Computing machinery and intelligence, *Mind*, 59, 433–460.

Whitley, L.D. (1993). *Foundations of Genetic Algorithms 2*. Morgan Kaufmann, San Francisco, CA.

Wright, S. (1932). The roles of mutation, inbreeding, crossbreeding, and selection in evolution, *Proceedings of the 6th International Congress on Genetics*, Ithaca, NY, vol. 1, pp. 356–366.

Bibliography

Arnold, D.V. and Beyer, H.-G. (2002). *Noisy Optimization with Evolution Strategies*. Kluwer Academic Publishers, Boston.

Beyer, H.-G. (2001). *The Theory of Evolution Strategies*. Springer-Verlag, Heidelberg.

Cantu-Paz, E. (2000). *Designing Efficient Parallel Genetic Algorithms*. Kluwer Academic Publishers, Boston.

Christian, J. (2001). *Illustrating Evolutionary Computation with Mathematica*. Morgan Kaufmann, San Francisco, CA.

Coley, D.A. (1999). *An Introduction to Genetic Algorithms for Scientists and Engineers*. World Scientific, Singapore.

Davidor, Y. (1990). *Genetic Algorithms and Robotics*. World Scientific, Singapore.

Gen, M. and Cheng, R. (1997). *Genetic Algorithms and Engineering Design*. John Wiley, New York.

Gen, M. and Cheng, R. (1999). *Genetic Algorithms and Engineering Optimization*. John Wiley, New York.

Goldberg, D.E. (2002). *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Kluwer Academic Publishers, Boston.

Haupt, R.L. and Haupt, S.E. (1998). *Practical Genetic Algorithms*. John Wiley, New York.

Koza, J.R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA.

Koza, J.R., Bennett III, F.H., Andre, D. and Keane, M.A. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann, San Francisco, CA.

Koza, J.R., Keane, M.A., Streeter, M.J., Mydlowec, W., Yu, J. and Lanza, G. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, Boston.

Langdon, W.B. (1998). *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming*! Kluwer Academic Publishers, Amsterdam.

Langdon, W.B. and Poli, R. (2002). *Foundations of Genetic Programming*. Springer-Verlag, Berlin.

Man, K.F., Tang, K.S., Kwong, S. and Halang, W.A. (1997). *Genetic Algorithms for Control and Signal Processing*. Springer-Verlag, London.

Man, K.F., Tang, K.S. and Kwong, S. (1999). *Genetic Algorithms: Concepts and Designs*. Springer-Verlag, London.

O'Neill, M. and Ryan, C. (2003). *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers, Boston.

Vose, M.D. (1999). *The Simple Genetic Algorithm: Foundations and Theory*. MIT Press, Cambridge, MA.