1. Basic Tools and Techniques

1.1 Moment-generating Function and *z*-transform

Consider a discrete random variable X that can take on a finite number of possible values. The probabilistic distribution of this variable can be represented by the finite vector $\boldsymbol{x} = (x_0, \dots, x_k)$ consisting of the possible values of X and the finite vector $\boldsymbol{p}$ consisting of the corresponding probabilities $p_i = \Pr\{X = x_i\}$.
The mapping $x_i \to p_i$ is usually called the *probability mass function* (p.m.f.)

X must take one of the values x_i. Therefore

$$\sum_{i=0}^{k} p_i = 1 \tag{1.1}$$

Example 1.1

Suppose that one performs k independent trials and each trial can result either in a success (with probability π) or in a failure (with probability $1-\pi$). Let random variable X represent the number of successes that occur in k trials. Such a variable is called a binomial random variable. The p.m.f. of X takes the form

$$x_i = i,\ \ p_i = \binom{k}{i} \pi^i (1-\pi)^{k-i},\ \ 0 \le i \le k$$

According to the binomial theorem it can be seen that

$$\sum_{i=0}^{k} p_i = \sum_{i=0}^{k} \binom{k}{i} \pi^i (1-\pi)^{k-i} = [\pi + (1-\pi)]^k = 1$$

The expected value of X is defined as a weighted average of the possible values that X can take on, where each value is weighted by the probability that X assumes that value:

$$E(X) = \sum_{i=0}^{k} x_i p_i \tag{1.2}$$

Example 1.2

The expected value of a binomial random variable is

$$E(X) = \sum_{i=0}^{k} x_i p_i = \sum_{i=0}^{k} i \binom{k}{i} \pi^i (1-\pi)^{k-i}$$

$$= k\pi \sum_{i=0}^{k-1} \binom{k-1}{i} \pi^i (1-\pi)^{k-i-1} = k\pi[\pi + (1-\pi)]^{k-1} = k\pi$$

The moment-generating function $m(t)$ of the discrete random variable X with p.m.f. $\boldsymbol{x}, \boldsymbol{p}$ is defined for all values of t by

$$m(t) = E(e^{tX}) = \sum_{i=0}^{k} e^{tx_i} p_i \tag{1.3}$$

The function $m(t)$ is called the moment-generating function because all of the moments of random variable X can be obtained by successively differentiating $m(t)$. For example:

$$m'(t) = \frac{\mathrm{d}}{\mathrm{d}t}\left(\sum_{i=0}^{k} e^{tx_i} p_i\right) = \sum_{i=0}^{k} x_i e^{tx_i} p_i. \tag{1.4}$$

Hence

$$m'(0) = \sum_{i=0}^{k} x_i p_i = E(X) \tag{1.5}$$

Then

$$m''(t) = \frac{\mathrm{d}}{\mathrm{d}t}(m'(t)) = \frac{\mathrm{d}}{\mathrm{d}t}\left(\sum_{i=0}^{k} x_i e^{tx_i} p_i\right) = \sum_{i=0}^{k} x_i^2 e^{tx_i} p_i \tag{1.6}$$

and

$$m''(0) = \sum_{i=0}^{k} x_i^2 p_i = E(X^2) \tag{1.7}$$

The nth derivative of $m(t)$ is equal to $E(X^n)$at $t = 0$.

Example 1.3

The moment-generating function of the binomial distribution takes the form

$$m(t) = E(e^{tX}) = \sum_{i=0}^{k} e^{ti} \binom{k}{i} \pi^i (1-\pi)^{k-i}$$

$$= \sum_{i=0}^{k} (\pi\ e^t)^i \binom{k}{i} (1-\pi)^{k-i} = (\pi\ e^t + 1 - \pi)^k$$

Hence

$$m'(t) = k(\pi\ e^t + 1 - \pi)^{k-1} \pi e^t \quad \text{and} \quad E(X) = m'(0) = k\pi .$$

The moment-generating function of a random variable uniquely determines its p.m.f. This means that a one-to-one correspondence exists between the p.m.f. and the moment-generating function.

The following important property of moment-generating function is of special interest for us. The moment-generating function of the sum of the independent random variables is the product of the individual moment-generating functions of these variables. Let $m_X(t)$ and $m_Y(t)$ be the moment-generating functions of random variables X and Y respectively. The p.m.f. of the random variables are represented by the vectors

$$\boldsymbol{x} = (x_0, \ldots, x_{k_X}), \quad \boldsymbol{p}_X = (p_{X0}, \ldots, p_{Xk_X})$$

and

$$\boldsymbol{y} = (y_0, \ldots, y_{k_Y}), \quad \boldsymbol{p}_Y = (p_{Y0}, \ldots, p_{Yk_Y})$$

respectively. Then $m_{X+Y}(t)$, the moment-generating function of $X + Y$, is obtained as

$$m_{X+Y}(t) = m_X(t) m_Y(t) = \sum_{i=0}^{k_X} e^{tx_i} p_{X_i} \sum_{j=0}^{k_Y} e^{ty_j} p_{Y_j}$$

$$= \sum_{i=0}^{k_X} \sum_{j=0}^{k_Y} e^{tx_i} e^{ty_j} p_{X_i} p_{Y_j} = \sum_{i=0}^{k_X} \sum_{j=0}^{k_Y} e^{t(x_i + y_j)} p_{X_i} p_{Y_j} \tag{1.8}$$

The resulting moment-generating function $m_{X+Y}(t)$ relates the probabilities of all the possible combinations of realizations $X = x_i$, $Y = y_j$, for any i and j, with the values that the random function $X + Y$ takes on for these combinations.

In general, for n independent discrete random variables $X_1, \ldots, X_n$

$$m_{\sum_{i=1}^{n} X_i}(t) = \prod_{i=1}^{n} m_{X_i}(t) \tag{1.9}$$

By replacing the function e^t by the variable z in Equation (1.3) we obtain another function related to random variable X that uniquely determines its p.m.f.:

$$\omega(z)=E(z^X)= \sum_{i=0}^{k} z^{x_i} p_i \tag{1.10}$$

This function is usually called the z-transform of discrete random variable X. The z-transform preserves some basic properties of the moment-generating functions. The first derivative of $\omega(z)$ is equal to $E(X)$ at $z = 1$. Indeed:

$$\omega'(z) = \frac{\mathrm{d}}{\mathrm{d}t}\left(\sum_{i=0}^{k} z^{x_i} p_i\right) = \sum_{i=0}^{k} x_i z^{x_i - 1} p_i \tag{1.11}$$

Hence

$$\omega'(1) = \sum_{i=0}^{k} x_i p_i = E(X) \tag{1.12}$$

The z-transform of the sum of independent random variables is the product of the individual z-transforms of these variables:

$$\begin{aligned}\omega_{X+Y}(z) &= \omega_X(z)\omega_Y(z) = \sum_{i=0}^{k_X} z^{x_i} p_{X_i} \sum_{j=0}^{k_Y} z^{y_i} p_{Y_j} \\ &= \sum_{i=0}^{k_X} \sum_{j=0}^{k_Y} z^{x_i} z^{y_j} p_{X_i} p_{Y_j} = \sum_{i=0}^{k_X} \sum_{j=0}^{k_Y} z^{(x_i + y_j)} p_{X_i} p_{Y_j}\end{aligned} \tag{1.13}$$

and in general

$$\omega_{\sum_{i=1}^{n} X_i} = \prod_{i=1}^{n} \omega_{X_i}(z) \tag{1.14}$$

The reader wishing to learn more about the generating function and *z*-transform is referred to the books by Grimmett and Stirzaker [3] and Ross [4].

Example 1.4

Suppose that one performs *k* independent trials and each trial can result either in a success (with probability π) or in a failure (with probability $1 - \pi$). Let random variable X_j represent the number of successes that occur in the *j*th trial.

The p.m.f. of any variable X_j ($1 \le j \le k$) is

$$\Pr\{X_j = 1\} = \pi, \ \Pr\{X_j = 0\} = 1 - \pi.$$

The corresponding *z*-transform takes the form

$$\omega_{Xj}(z) = \pi z^1 + (1-\pi)z^0$$

The random number of successes that occur in *k* trials is equal to the sum of the numbers of successes in each trial

$$X = \sum_{j=1}^{k} X_j$$

Therefore, the corresponding *z*-transform can be obtained as

$$\omega_X(z) = \prod_{j=1}^{n} \omega_{X_j}(z) = [\pi \ z + (1-\pi)z^0]^k$$

$$= \sum_{i=0}^{k} \binom{k}{i} z^i \pi^i (1-\pi)^{k-i} = \sum_{i=0}^{k} z^i \binom{k}{i} \pi^i (1-\pi)^{k-i}$$

This *z*-transform corresponds to the binomial p.m.f:

$$x_i = i, \ p_i = \binom{k}{i} \pi^i (1-\pi)^{k-i}, \ 0 \le i \le k$$

1.2 Mathematical Fundamentals of the Universal Generating Function

1.2.1 Definition of the Universal Generating Function

Consider n independent discrete random variables $X_1, \ldots, X_n$ and assume that each variable X_i has a p.m.f. represented by the vectors $\boldsymbol{x}_i$, $\boldsymbol{p}_i$. In order to evaluate the p.m.f. of an arbitrary function $f(X_1, \ldots, X_n)$, one has to evaluate the vector $\boldsymbol{y}$ of all of the possible values of this function and the vector $\boldsymbol{q}$ of probabilities that the function takes these values.

Each possible value of function f corresponds to a combination of the values of its arguments $X_1, \ldots, X_n$. The total number of possible combinations is

$$K = \prod_{i=1}^{n} (k_i + 1) \tag{1.15}$$

where $k_i + 1$ is the number of different realizations of random variable X_i. Since all of the n variables are statistically independent, the probability of each unique combination is equal to the product of the probabilities of the realizations of arguments composing this combination.

The probability of the jth combination of the realizations of the variables can be obtained as

$$q_j = \prod_{i=1}^{n} p_{ij_i} \tag{1.16}$$

and the corresponding value of the function can be obtained as

$$f_j = f(x_{1j_1}, \ \ldots, \ x_{nj_n}) \tag{1.17}$$

Some different combinations may produce the same values of the function. All of the combinations are mutually exclusive. Therefore, the probability that the function takes on some value is equal to the sum of probabilities of the combinations producing this value. Let A_h be a set of combinations producing the value f_h. If the total number of different realizations of the function $f(X_1, \ldots, X_n)$ is H, then the p.m.f. of the function is

$$\boldsymbol{y} = (f_h : 1 \le h \le H), \quad \boldsymbol{q} = \Big(\sum_{(x_{1j_1}, \ldots, x_{nj_n}) \in A_h} \prod_{i=1}^{n} p_{ij_i} : 1 \le h \le H \Big) \tag{1.18}$$

Example 1.5

Consider two random variables X_1 and X_2 with p.m.f. $\boldsymbol{x_1} = (1, 4)$, $\boldsymbol{p_1} = (0.6, 0.4)$ and $\boldsymbol{x_2} = (0.5, 1, 2)$, $\boldsymbol{p_2} = (0.1, 0.6, 0.3)$. In order to obtain the p.m.f. of the function $Y = X_1^{X_2}$ we have to consider all of the possible combinations of the values taken by the variables. These combinations are presented in Table 1.1.

The values of the function Y corresponding to different combinations of realizations of its random arguments and the probabilities of these combinations can be presented in the form

$$\boldsymbol{y} = (1, 2, 1, 4, 1, 16),\ \ \boldsymbol{q} = (0.06, 0.04, 0.36, 0.24, 0.18, 0.12)$$

Table 1.1. p.m.f. of the function of two variables

No of combination	Combination probability	Value of X_1	Value of X_2	Value of Y
1	0.6×0.1 = 0.06	1	0.5	1
2	0.4×0.1 = 0.04	4	0.5	2
3	0.6×0.6 = 0.36	1	1	1
4	0.4×0.6 = 0.24	4	1	4
5	0.6×0.3 = 0.18	1	2	1
6	0.4×0.3 = 0.12	4	2	16

Note that some different combinations produce the same values of the function Y. Since all of the combinations are mutually exclusive, we can obtain the probability that the function takes some value as being the sum of the probabilities of different combinations of the values of its arguments that produce this value:

$$\Pr\{Y = 1\} = \Pr\{X_1 = 1,\ X_2 = 0.5\} + \Pr\{X_1 = 1,\ X_2 = 1\}$$

$$+ \Pr\{X_1 = 1,\ X_2 = 2\} = 0.06 + 0.36 + 0.18 = 0.6$$

The p.m.f. of the function Y is

$$\boldsymbol{y} = (1, 2, 4, 16),\ \ \boldsymbol{q} = (0.\ 6, 0.04, 0.24, 0.12)$$

The z-transform of each random variable X_i represents its p.m.f. $(x_{i0}, \ldots, x_{ik_i})$, $(p_{i0}, \ldots, p_{ik_i})$ in the polynomial form

$$\sum_{j=0}^{k_i} p_{ij} z^{x_{ij}} \tag{1.19}$$

According to (1.14), the product of the z-transform polynomials corresponding to the variables $X_1, \ldots, X_n$ determines the p.m.f. of the sum of these variables.

In a similar way one can obtain the z-transform representing the p.m.f. of the arbitrary function f by replacing the product of the polynomials by a more general composition operator $\otimes_f$ over z-transform representations of p.m.f. of n independent variables:

$$\underset{f}{\otimes}\left(\sum_{j_i=0}^{k_i} p_{ij_i} z^{x_{ij_i}}\right) = \sum_{j_1=0}^{k_1} \sum_{j_2=0}^{k_2} \dots \sum_{j_n=0}^{k_n} \left(\prod_{i=0}^{n} p_{ij_i} z^{f(x_{ij_1},\dots,x_{nj_n})}\right) \tag{1.20}$$

The technique based on using z-transform and composition operators $\otimes_f$ is named the universal z-transform or universal (moment) generating function (UGF) technique. In the context of this technique, the z-transform of a random variable for which the operator $\otimes_f$ is defined is referred to as its u-function. We refer to the u-function of variable X_i as $u_j(z)$, and to the u-function of the function $f(X_1, \dots, X_n)$ as $U(z)$. According to this notation

$$U(z) = \underset{f}{\otimes}(u_1(z),\ u_2(z),\ \dots,\ u_n(z)) \tag{1.21}$$

where $u_i(z)$ takes the form (1.19) and $U(z)$ takes the form (1.20). For functions of two arguments, two interchangeable notations can be used:

$$U(z) = \underset{f}{\otimes}(u_1(z),\ u_2(z)) = u_1(z) \underset{f}{\otimes} u_2(z) \tag{1.22}$$

Despite the fact that the u-function resembles a polynomial, it is not a polynomial because:

- Its coefficients and exponents are not necessarily scalar variables, but can be other mathematical objects (*e.g.* vectors);
- Operators defined over the u-functions can differ from the operator of the polynomial product (unlike the ordinary z-transform, where only the product of polynomials is defined).

When the u-function $U(z)$ represents the p.m.f. of a random function $f(X_1,\dots, X_n)$, the expected value of this function can be obtained (as an analogy with the regular z-transform) as the first derivative of $U(z)$ at $z = 1$.

In general, the u-functions can be used not just for representing the p.m.f. of random variables. In the following chapters we also use other interpretations. However, in any interpretation the coefficients of the terms in the u-function represent the probabilistic characteristics of some object or state encoded by the exponent in these terms.

The u-functions inherit the essential property of the regular polynomials: they allow for collecting like terms. Indeed, if a u-function representing the p.m.f. of a random variable X contains the terms $p_h z^{x_h}$ and $p_m z^{x_m}$ for which $x_h = x_m$, the two terms can be replaced by a single term $(p_h + p_m)z^{x_m}$, since in this case $\Pr\{X = x_h\} = \Pr\{X = x_m\} = p_h+p_m$.

Example 1.6

Consider the p.m.f. of the function Y from Example 1.5, obtained from Table 1.1. The u-function corresponding to this p.m.f. takes the form:

$$U(z) = 0.06z^1 + 0.04z^2 + 0.36z^1 + 0.24z^4 + 0.18z^1 + 0.12z^{16}$$

By collecting the like terms in this u-function we obtain:

$$U(z) = 0.6z^1 + 0.04z^2 + 0.24z^4 + 0.12z^{16}$$

which corresponds to the final p.m.f. obtained in Example 1.5.

The expected value of Y can be obtained as

$$E(Y) = U'(1) = 0.6\times1 + 0.04\times2 + 0.24\times4 + 0.12\times16 = 3.56$$

The described technique of determining the p.m.f. of functions is based on an enumerative approach. This approach is extremely resource consuming. Indeed, the resulting u-function $U(z)$ contains K terms (see Equation (1.15)), which requires excessive storage space. In order to obtain $U(z)$ one has to perform $(n - 1)K$ procedures of probabilities multiplication and K procedures of function evaluation. Fortunately, many functions used in reliability engineering produce the same values for different combinations of the values of their arguments. The combination of recursive determination of the functions with simplification techniques based on the like terms collection allows one to reduce considerably the computational burden associated with evaluating the p.m.f. of complex functions.

Example 1.7

Consider the function

$$Y = f(X_1, \ldots, X_5) = (\max(X_1, X_2) + \min(X_3, X_4))\, X_5$$

of five independent random variables $X_1, \ldots, X_5$. The probability mass functions of these variables are determined by pairs of vectors $\boldsymbol{x}_i$, $\boldsymbol{p}_i$ $(0 \le i \le 5)$ and are presented in Table 1.2.

These p.m.f. can be represented in the form of u-functions as follows:

$$u_1(z) = p_{10}z^{x_{10}} + p_{11}z^{x_{11}} + p_{12}z^{x_{12}} = 0.6z^5 + 0.3z^8 + 0.1z^{12}$$

$$u_2(z) = p_{20}z^{x_{20}} + p_{21}z^{x_{21}} = 0.7z^8 + 0.3z^{10}$$

$$u_3(z) = p_{30}z^{x_{30}} + p_{31}z^{x_{31}} = 0.6z^0 + 0.4z^1$$

$$u_4(z) = p_{40}z^{x_{40}} + p_{41}z^{x_{41}} + p_{42}z^{x_{42}} = 0.1z^0 + 0.5z^8 + 0.4z^{10}$$

$$u_5(z) = p_{50}z^{x_{50}} + p_{51}z^{x_{51}} = 0.5z^1 + 0.5z^{1.5}$$

Using the straightforward approach one can obtain the p.m.f. of the random variable Y applying the operator (1.20) over these u-functions. Since $k_1 + 1 = 3$, $k_2 + 1 = 2$, $k_3 + 1 = 2$, $k_4 + 1 = 3$, $k_5 + 1 = 2$, the total number of term multiplication procedures that one has to perform using this equation is 3×2×2×3×2 = 72.

Table 1.2. p.m.f. of random variables

X_1	p_1	0.6	0.3	0.1
	x_1	5	8	12
X_2	p_2	0.7	0.3	-
	x_2	8	10	-
X_3	p_3	0.6	0.4	-
	x_3	0	1	-
X_4	p_4	0.1	0.5	0.4
	x_4	0	8	10
X_5	p_5	0.5	0.5	-
	x_5	1	1.5	-

Now let us introduce three auxiliary random variables X_6, X_7 and X_8, and define the same function recursively:

$$X_6 = \max\{X_1, X_2\}$$

$$X_7 = \min\{X_3, X_4\}$$

$$X_8 = X_6 + X_7$$

$$Y = X_8 X_5$$

We can obtain the p.m.f. of variable Y using composition operators over pairs of u-functions as follows:

$$u_6(z) = u_1(z) \underset{\max}{\otimes} u_1(z) = (0.6z^5 + 0.3z^8 + 0.1z^{12}) \underset{\max}{\otimes} (0.7z^8 + 0.3z^{10})$$

$$=0.42z^{\max\{5,8\}} + 0.21z^{\max\{8,8\}} + 0.07z^{\max\{12,8\}} + 0.18z^{\max\{5,10\}} + 0.09z^{\max\{8,10\}}$$

$$+ 0.03z^{\max\{12,10\}} = 0.63z^8 + 0.27z^{10} + 0.1z^{12}$$

$$u_7(z) = u_3(z) \underset{\min}{\otimes} u_4(z) = (0.6z^0 + 0.4z^2) \underset{\min}{\otimes} (0.1z^0 + 0.5z^3 + 0.4z^5)$$

$$= 0.06z^{\min\{0,0\}} + 0.04z^{\min\{2,0\}} + 0.3z^{\min\{0,3\}} + 0.2z^{\min\{2,3\}}$$

$$+0.24z^{\min\{0,5\}} + 0.16z^{\min\{2,5\}} = 0.64z^0 + 0.36z^2$$

$$u_8(z) = u_6(z) \underset{+}{\otimes} u_7(z) = (0.63z^8 + 0.27z^{10} + 0.1z^{12}) \underset{+}{\otimes} (0.64z^0 + 0.36z^2)$$

$$= 0.4032z^{8+0} + 0.1728z^{10+0} + 0.064z^{12+0} + 0.2268z^{8+2} + 0.0972z^{10+2}$$
$$+ 0.036z^{12+2} = 0.4032z^{8} + 0.3996z^{10} + 0.1612z^{12} + 0.036z^{14}$$
$$U(z) = u_8(z) \underset{\times}{\otimes} u_5(z)$$
$$= (0.4032z^{8} + 0.3996z^{10} + 0.1612z^{12} + 0.036z^{14})(0.5z^{1} + 0.5z^{1.5})$$
$$= 0.2016z^{8\times1} + 0.1998z^{10\times1} + 0.0806z^{12\times1} + 0.018z^{14\times1} + 0.2016z^{8\times1.5}$$
$$+ 0.1998z^{10\times1.5} + 0.0806z^{12\times1.5} + 0.018z^{14\times1.5} = 0.2016z^{8} + 0.1998z^{10}$$
$$+ 0.2822z^{12} + 0.018z^{14} + 0.1998z^{15} + 0.0806z^{18} + 0.018z^{21}$$

The final u-function $U(z)$ represents the p.m.f. of Y, which takes the form

$$\boldsymbol{y} = (8, 10, 12, 14, 15, 18, 21)$$
$$\boldsymbol{q} = (0.2016, 0.1998, 0.2822, 0.018, 0.1998, 0.0806, 0.018)$$

Observe that during the recursive derivation of this p.m.f. we used only 26 term multiplication procedures. This considerable computational complexity reduction is possible because of the like term collection in intermediate u-functions.

The problem of system reliability analysis usually includes evaluation of the p.m.f. of some random values characterizing the system's behaviour. These values can be very complex functions of a large number of random variables. The explicit derivation of such functions is an extremely complicated task. Fortunately, the UGF method for many types of system allows one to obtain the system u-function recursively. This property of the UGF method is based on the associative property of many functions used in reliability engineering. The recursive approach presumes obtaining u-functions of subsystems containing several basic elements and then treating the subsystem as a single element with the u-function obtained when computing the u-function of a higher level subsystem. Combining the recursive approach with the simplification technique reduces the number of terms in the intermediate u-functions and provides a drastic reduction of the computational burden.

1.2.2 Properties of Composition Operators

The properties of composition operator $\otimes_f$ strictly depend on the properties of the function $f(X_1, ..., X_n)$. Since the procedure of the multiplication of the probabilities in this operator is commutative and associative, the entire operator can also possess these properties if the function possesses them.

If

$$f(X_1, X_2, \dots, X_n) = f(f(X_1, X_2, \dots, X_{n-1}), X_n), \tag{1.23}$$

then:

$$\begin{aligned}U(z) &= \underset{f}{\otimes}(u_1(z), u_2(z), \ \ldots, \ u_n(z)) \\ &= \underset{f}{\otimes}(\underset{f}{\otimes}(u_1(z), u_2(z), \ \ldots, \ u_{n-1}(z)), \ u_n(z)).\end{aligned} \tag{1.24}$$

Therefore, one can obtain the u-function $U(z)$ assigning $U_1(z) = u_1(z)$ and applying operator $\underset{f}{\otimes}$ consecutively:

$$U_j(z) = \underset{f}{\otimes} \ (U_{j-1}(z), \ u_j(z)) \ \text{ for } 2 \le j \le n, \tag{1.25}$$

such that finally $U(z) = U_n(z)$.

If the function f possesses the associative property

$$f(X_1, \ldots, X_j, X_{j+1}, \ldots, X_n) = f(\ f(X_1, \ldots, X_j), \ f(X_{j+1}, \ldots, X_n)) \tag{1.26}$$

for any j, then the $\underset{f}{\otimes}$ operator also possesses this property:

$$\begin{aligned}&\underset{f}{\otimes}(u_1(z), \ \ldots, \ u_n(z)) \\ &= \underset{f}{\otimes}(\underset{f}{\otimes}(u_1(z), \ \ldots, \ u_{j-1}(z)), \underset{f}{\otimes}(u_j(z), \ \ldots, \ u_n(z))\end{aligned} \tag{1.27}$$

If, in addition to the property (1.24), the function f is also commutative:

$$f(X_1, \ldots, X_j, X_{j+1}, \ldots, X_n) = f(X_1, \ldots, X_{j+1}, X_j, \ldots, X_n) \tag{1.28}$$

then for any j, which provides the commutative property for the $\underset{f}{\otimes}$ operator:

$$\begin{aligned}&\underset{f}{\otimes}(u_1(z), \ldots, u_j(z), u_{j+1}(z), \ldots, u_n(z)) \\ &= \underset{f}{\otimes}(u_1(z), \ldots, u_{j+1}(z), u_j(z), \ldots, u_n(z))\end{aligned} \tag{1.29}$$

the order of arguments in the function $f(X_1, \ldots, X_n)$ is inessential and the u-function $U(z)$ can be obtained using recursive procedures (1.23) and (1.25) over any permutation of u-functions of random arguments $X_1, \ldots, X_n$.

If a function takes the recursive form

$$f(f_1(X_1, \ldots, X_j), \ f_2(X_{j+1}, \ldots, X_h), \ \ldots, \ f_m(X_l, \ldots, X_n)) \tag{1.30}$$

then the corresponding u-function $U(z)$ can also be obtained recursively:

$$\underset{f}{\otimes}(\underset{f_1}{\otimes}(u_1(z),...,u_j(z)),\ \underset{f_2}{\otimes}(u_{j+1}(z),...,u_h(z)),...,\ \underset{f_m}{\otimes}(u_l(z),...,u_n(z)). \quad (1.31)$$

Example 1.8

Consider the variables X_1, X_2, X_3 with p.m.f. presented in Table 1.1. The u-functions of these variables are:

$$u_1(z) = 0.6z^5 + 0.3z^8 + 0.1z^{12}$$

$$u_2(z) = 0.7z^8 + 0.3z^{10}$$

$$u_3(z) = 0.6z^0 + 0.4z^1$$

The function $Y = \min(X_1, X_2, X_3)$ possesses both commutative and associative properties. Therefore

$$\min(\min(X_1, X_2), X_3) = \min(\min(X_2, X_1), X_3) = \min(\min(X_1, X_3), X_2)$$

$$= \min(\min(X_3, X_1), X_2) = \min(\min(X_2, X_3), X_1) = \min(\min(X_3, X_2), X_1).$$

The u-function of Y can be obtained using the recursive procedure

$$u_4(z) = u_1(z) \underset{\min}{\otimes} u_2(z) = (0.6z^5 + 0.3z^8 + 0.1z^{12}) \underset{\min}{\otimes} (0.7z^8 + 0.3z^{10})$$

$$= 0.42z^{\min\{5,8\}} + 0.21z^{\min\{8,8\}} + 0.07z^{\min\{12,8\}}$$

$$+ 0.18z^{\min\{5,10\}} + 0.09z^{\min\{8,10\}} + 0.03z^{\min\{12,10\}} = 0.6z^5 + 0.37z^8 + 0.03z^{10}$$

$$U(z) = u_4(z) \underset{\min}{\otimes} u_3(z) = (0.6z^5 + 0.37z^8 + 0.03z^{10}) \underset{\min}{\otimes} (0.6z^0 + 0.4z^1)$$

$$= 0.36z^{\min\{5,0\}} + 0.222z^{\min\{8,0\}} + 0.018z^{\min\{12,0\}}$$

$$+ 0.24z^{\min\{5,1\}} + 0.148z^{\min\{8,1\}} + 0.012z^{\min\{12,1\}} = 0.6z^0 + 0.4z^1$$

The same u-function can also be obtained using another recursive procedure

$$u_4(z) = u_1(z) \underset{\min}{\otimes} u_3(z) = (0.6z^5 + 0.3z^8 + 0.1z^{12}) \underset{\min}{\otimes} (0.6z^0 + 0.4z^1)$$

$$= 0.36z^{\min\{5,0\}} + 0.18z^{\min\{8,0\}} + 0.06z^{\min\{12,0\}}$$

$$+0.24z^{\min\{5,1\}} + 0.12z^{\min\{8,1\}} + 0.04z^{\min\{12,1\}} = 0.6z^0 + 0.4z^1;$$

$$U(z) = u_3(z) \underset{\min}{\otimes} u_2(z) = (0.6z^0 + 0.4z^1) \underset{\min}{\otimes} (0.7z^8 + 0.3z^{10})$$

$$= 0.42z^{\min\{0,8\}} + 0.28z^{\min\{1,8\}} + 0.18z^{\min\{0,10\}} + 0.12z^{\min\{1,10\}} = 0.6z^0 + 0.4z^1$$

Observe that while both recursive procedures produce the same u-function, their computational complexity differs . In the first case, 12 term multiplication operations have been performed; in the second case, only 10 operations have been performed.

Consider a random variable X with p.m.f. represented by u-function $u_X(z)=\sum_{j=0}^{k} p_j z^{x_j}$. In order to obtain the u-function representing the p.m.f. of function $f(X, c)$ of the variable X and a constant c one can apply the following simplified operator:

$$U(z) = u_X(z) \underset{f}{\otimes} c = (\sum_{j=0}^{k} p_j z^{x_j}) \underset{f}{\otimes} c = \sum_{j=0}^{k} p_j z^{f(x_j,c)} \tag{1.32}$$

This can be easily proved if we represent the constant c as the random variable C that can take the value of c with a probability of 1. The u-function of such a variable takes the form

$$u_c(z) = z^{c} \tag{1.33}$$

Applying the operator $\otimes_f$ over the two u-functions $u_X(z)$ and $u_c(z)$ we obtain Equation (1.32).

1.3 Introduction to Genetic Algorithms

An abundance of optimization methods have been used to solve various reliability optimization problems. The algorithms applied are either heuristics or exact procedures based mainly on modifications of dynamic programming and nonlinear programming. Most of these methods are strongly problem oriented. This means that, since they are designed for solving certain optimization problems, they cannot be easily adapted for solving other problems. In recent years, many studies on reliability optimization use a universal optimization approach based on metaheuristics. These metaheuristics hardly depend on the specific nature of the problem that is solved and, therefore, can be easily applied to solve a wide range of optimization problems. The metaheuristics are based on artificial reasoning rather than on classical mathematical programming. Their important advantage is that they do not require any information about the objective function besides its values corresponding to the points visited in the solution space. All metaheuristics use the idea of randomness when performing a search, but they also use past knowledge in order to direct the search. Such search algorithms are known as randomized search techniques.

Genetic algorithms (GAs) are one of the most widely used metaheuristics. They were inspired by the optimization procedure that exists in nature, the biological phenomenon of evolution. A GA maintains a population of different solutions allowing them to mate, produce offspring, mutate, and fight for survival. The principle of survival of the fittest ensures the population's drive towards optimization. The GAs have become the popular universal tool for solving various optimization problems, as they have the following advantages:

- they can be easily implemented and adapted;
- they usually converge rapidly on solutions of good quality;
- they can easily handle constrained optimization problems;
- they produce variety of good quality solutions simultaneously, which is important in the decision-making process.

The GA concept was developed by John Holland at the University of Michigan and first described in his book [5]. Holland was impressed by the ease with which biological organisms could perform tasks, which eluded even the most powerful computers. He also noted that very few artificial systems have the most remarkable characteristics of biological systems: robustness and flexibility. Unlike technical systems, biological ones have methods for self-guidance, self-repair and reproducing these features. Holland's biologically inspired approach to optimization is based on the following analogies:

- As in nature, where there are many organisms, there are many possible solutions to a given problem.
- As in nature, where an organism contains many genes defining its properties, each solution is defined by many interacting variables (parameters).
- As in nature, where groups of organisms live together in a population and some organisms in the population are more fit than others, a group of possible solutions can be stored together in computer memory and some of them are closer to the optimum than others.
- As in nature, where organisms that are fitter have more chances of mating and having offspring, solutions that are closer to the optimum can be selected more often to combine their parameters to form new solutions.
- As in nature, where organisms produced by good parents are more likely to be better adapted than the average organism because they received good genes, offspring of good solutions are more likely to be better than a random guess, since they are composed of better parameters.
- As in nature, where survival of the fittest ensures that the successful traits continue to get passed along to subsequent generations, and are refined as the population evolves, the survival-of-the-fittest rule ensures that the composition of the parameters corresponding to the best guesses continually get refined.

GAs maintain a population of individual solutions, each one represented by a finite string of symbols, known as the *genome*, encoding a possible solution within a given problem space. This space, referred to as the *search space*, comprises all of the possible solutions to the problem at hand. Generally speaking, a GA is applied to spaces, which are too large to be searched exhaustively.

GAs exploit the idea of the survival of the fittest and an interbreeding population to create a novel and innovative search strategy. They iteratively create new populations from the old ones by ranking the strings and interbreeding the fittest to create new strings, which are (hopefully) closer to the optimum solution for the problem at hand. In each generation, a GA creates a set of strings from pieces of the previous strings, occasionally adding random new data to keep the population from stagnating. The result is a search strategy that is tailored for vast, complex, multimodal search spaces.

The idea of survival of the fittest is of great importance to genetic algorithms. GAs use what is termed as the fitness function in order to select the fittest string to be used to create new, and conceivably better, populations of strings. The fitness function takes a string and assigns it a relative fitness value. The method by which it does this and the nature of the fitness value do not matter. The only thing that the fitness function must do is rank the strings in some way by producing their fitness values. These values are then used to select the fittest strings.

GAs use the idea of randomness when performing a search. However, it must be clearly understood that the GAs are not simply random search algorithms. Random search algorithms can be inherently inefficient due to the directionless nature of their search. GAs are not directionless. They utilize knowledge from previous generations of strings in order to construct new strings that will approach the optimal solution. GAs are a form of a randomized search, and the way that the strings are chosen and combined comprise a stochastic process.

The essential differences between GAs and other forms of optimization, according to Goldberg [6], are as follows.

GAs usually use a coded form of the solution parameters rather than their actual values. Solution encoding in a form of strings of symbols (an analogy to chromosomes containing genes) provides the possibility of crossover and mutation. The symbolic alphabet that was used was initially binary, due to certain computational advantages purported in [5]. This has been extended in recent years to include character-based encodings, integer and real-valued encodings, and tree representations [7].

GAs do not just use a single point on the problem space, rather they use a set, or population, of points (solutions) to conduct a search. This gives the GAs the power to search noisy spaces littered with local optimum points. Instead of relying on a single point to search through the space, GAs look at many different areas of the problem space at once, and use all of this information as a guide.

GAs use only payoff information to guide themselves through the problem space. Many search techniques need a range of information to guide themselves. For example, gradient methods require derivatives. The only information a GA needs to continue searching for the optimum is some measure of fitness about a point in the space.

GAs are probabilistic in nature, not deterministic. This is a direct result of the randomization techniques used by GAs.

GAs are inherently parallel. Herein lies one of their most powerful features. GAs, by their nature, are very parallel, dealing with a large number of solutions simultaneously. Using schemata theory, Holland has estimated that a GA, processing n strings at each generation, in reality processes n^3 useful substrings [6].

Two of the most common GA implementations are “generational” and “steady state”, although recently the steady-state technique has received increased attention [8]. This interest is partly attributed to the fact that steady-state techniques can offer a substantial reduction in the memory requirements of a system: the technique abolishes the need to maintain more than one population during the evolutionary process, which is necessary in the generational GA. In this way, genetic systems have greater portability for a variety of computer environments because of the reduced memory overhead. Another reason for the increased interest in steady-state techniques is that, in many cases, a steady-state GA has been shown to be more effective than a generational GA [9, 10]. This improved performance can be attributed to factors such as the diversity of the population and the immediate availability of superior individuals.

A comprehensive description of a generational GA can be found in [6]. Here, we present the structure of a steady-state GA.

1.3.1 Structure of Steady-state Genetic Algorithms

The steady-state GA (see Figure 1.1) proceeds as follows [11]: an initial population of solutions is generated randomly or heuristically. Within this population, new solutions are obtained during the genetic cycle by using the *crossover* operator. This operator produces an offspring from a randomly selected pair of parent solutions (the parent solutions are selected with a probability proportional to their relative fitness), facilitating the inheritance of some basic properties from the parents to the offspring. The newly obtained offspring undergoes *mutation* with the probability p_{mut}.

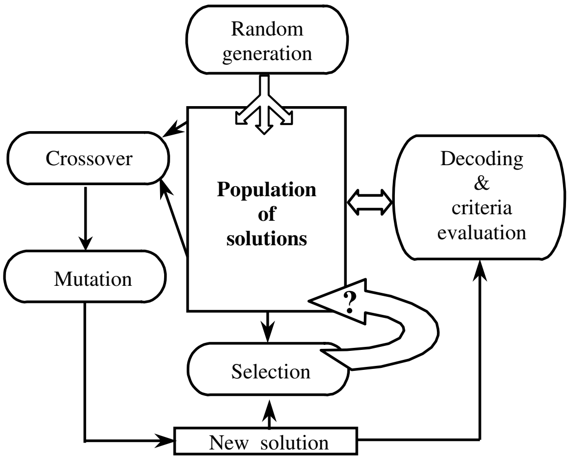

Figure 1.1. Structure of a steady-state GA

Each new solution is decoded and its objective function (fitness) values are estimated. These values, which are a measure of quality, are used to compare different solutions. The comparison is accomplished by a selection procedure that determines which solution is better: the newly obtained solution or the worst solution in the population. The better solution joins the population, while the other is discarded. If the population contains equivalent solutions following selection, then redundancies are eliminated and the population size decreases as a result.

A genetic cycle terminates when N_{rep} new solutions are produced or when the number of solutions in the population reaches a specified level. Then, new randomly constructed solutions are generated to replenish the shrunken population, and a new genetic cycle begins. The whole GA is terminated when its termination condition is satisfied. This condition can be specified in the same way as in a generational GA. The following is the steady-state GA in pseudo-code format.

```
begin STEADY STATE GA
 Initialize population $\Pi$
 Evaluate population $\Pi$ {compute fitness values}
 while GA termination criterion is not satisfied do
       {GENETIC CYCLE}
    while genetic cycle termination criterion is not satisfied do
         Select at random Parent Solutions $S_1$, $S_2$ from $\Pi$
         Crossover: ($S_1$, $S_2$) $\rightarrow$ $S_O$ {offspring}
         Mutate offspring $S_O \rightarrow S^*_O$ with probability $p_{\mathrm{mut}}$
         Evaluate $S^*_O$
              Replace $S_W$ {the worst solution in $\Pi$ with $S^*_O$ } if $S^*_O$ is
              better than $S_W$
         Eliminate identical solutions in $\Pi$
    end while
    Replenish $\Pi$ with new randomly generated solutions
 end while
end GA
```

Example 1.9

In this example we present several initial stages of a steady-state GA, that maximizes the function of six integer variables $x_1, \ldots, x_6$ taking the form

$$f(x_1,\ldots,x_6)=1000\ \ [(x_1-3.4)^2+(x_2-1.8)^2+(x_3-7.7)^2$$
$$+(x_4-3.1)^2+(x_5-2.8)^2+(x_6-8.8)^2]^{-1}$$

The variables can take values from 1 to 9. The initial population, consisting of five solutions ordered according to their fitness (value of function f), is:

No.	x_1	x_2	x_3	x_4	x_5	x_6	$f(x_1,\ldots,x_6)$
1	4	2	4	1	2	5	297.8
2	3	7	7	7	2	7	213.8
3	7	5	3	5	3	9	204.2
4	2	7	4	2	1	4	142.5
5	8	2	3	1	1	4	135.2

Using the random generator that produces the numbers of the solutions, the GA chooses the first and third strings, *i.e.* (4 2 4 1 2 5) and (7 5 3 5 3 9) respectively. From these strings, it produces a new one by applying a crossover procedure that takes the three first numbers from the better parent string and the last three numbers from the inferior parent string. The resulting string is (4 2 4 5 3 9). The fitness of this new solution is $f(x_1, ..., x_6) = 562.4$. The new solution enters the population, replacing the one with the lowest fitness. The new population is now

No.	x_1	x_2	x_3	x_4	x_5	x_6	$f(x_1,\ldots,x_6)$
1	4	2	4	5	3	9	562.4
2	4	2	4	1	2	5	297.8
3	3	7	7	7	2	7	213.8
4	7	5	3	5	3	9	204.2
5	2	7	4	2	1	4	142.5

Choosing at random the third and fourth strings, (3 7 7 7 2 7) and (7 5 3 5 3 9) respectively, the GA produces the new string (3 7 7 5 3 9) using the crossover operator. This string undergoes a mutation that changes one of its numbers by one (here, the fourth element of the string changes from 5 to 4). The resulting string (3 7 7 4 3 9) has a fitness of $f(x_1, ..., x_6) = 349.9$. This solution is better than the inferior one in the population; therefore, the new solution replaces the inferior one. Now the population takes the form

No.	x_1	x_2	x_3	x_4	x_5	x_6	$f(x_1,\ldots,x_6)$
1	4	2	4	5	3	9	562.4
2	3	7	7	4	3	9	349.9
3	4	2	4	1	2	5	297.8
4	3	7	7	7	2	7	213.8
5	7	5	3	5	3	9	204.2

A new solution (4 2 4 4 3 9) is obtained by the crossover operator over the randomly chosen first and second solutions, *i.e.* (4 2 4 5 3 9) and (3 7 7 4 3 9) respectively. After the mutation this solution takes the form (4 2 4 5 3 9) and has the fitness $f(x_1,..., x_6) = 1165.5$. The population obtained after the new solution joins it is

No.	x_1	x_2	x_3	x_4	x_5	x_6	$f(x_1,\ldots,x_6)$
1	4	2	5	4	3	9	1165.5
2	4	2	4	5	3	9	562.4
3	3	7	7	4	3	9	349.9
4	4	2	4	1	2	5	297.8
5	3	7	7	7	2	7	213.8

Note that the mutation procedure is not applied to all the solutions obtained by the crossover. This procedure is used with some prespecified probability p_{mut}. In our example, only the second and the third newly obtained solutions underwent the mutation.

The actual GAs operate with much larger populations and produce thousands of new solutions using the crossover and mutation procedures. The steady-state GA with a population size of 100 obtained the optimal solution for the problem presented after producing about 3000 new solutions. Note that the total number of possible solutions is $9^6 = 531441$. The GA managed to find the optimal solution by exploring less than 0.6% of the entire solution space.

Both types of GA are based on the crossover and mutation procedures, which depend strongly on the solution encoding technique. These procedures should preserve the feasibility of the solutions and provide the inheritance of their essential properties.

1.3.2 Adaptation of Genetic Algorithms to Specific Optimization Problems

There are three basic steps in applying a GA to a specific problem.

In the first step, one defines the solution representation (encoding in a form of a string of symbols) and determines the decoding procedure, which evaluates the fitness of the solution represented by the arbitrary string.

In the second step, one has to adapt the crossover and mutation procedures to the given representation in order to provide feasibility for the new solutions produced by these procedures as well as inheriting the basic properties of the parent solutions by their offspring.

In the third step, one has to choose the basic GA parameters, such as the population size, the mutation probability, the crossover probability (generational GA) or the number of crossovers per genetic cycle (in the steady-state GA), and formulate the termination condition in order to provide the greatest possible GA efficiency (convergence speed).

The strings representing GA solutions are randomly generated by the population generation procedure, modified by the crossover and mutation procedures, and decoded by the fitness evaluation procedure. Therefore, the solution representation in the GA should meet the following requirements:

- It should be easily generated (the sophisticated complex solution generation procedures reduce the GA speed).
- It should be as compact as possible (using very long strings requires excessive computational resources and slows the GA convergence).
- It should be unambiguous (i.e. different solutions should be represented by different strings).
- It should represent feasible solutions (if not any randomly generated string represents a feasible solution, then the feasibility should be provided by simple string transformation).
- It should provide feasibility inheritance of new solutions obtained from feasible ones by the crossover and mutation operators.

The field of reliability optimization includes the problems of finding optimal parameters, optimal allocation and assignment of different elements into a system, and optimal sequencing of the elements. Many of these problems are combinatorial by their nature. The most suitable symbol alphabet for this class of problems is integer numbers. The finite string of integer numbers can be easily generated and stored. The random generator produces integer numbers for each element of the string in a specified range. This range should be the same for each element in order to make the string generation procedure simple and fast. If for some reason different string elements should belong to different ranges, then the string should be transformed to provide solution feasibility.

In the following sections we show how integer strings can be interpreted for solving different kinds of optimization problems.

1.3.2.1 Parameter Determination Problems

When the problem lies in determining a vector of H parameters (x_1, x_2, ..., x_H) that maximizes an objective function $f(x_1, x_2, ..., x_H)$ one always has to specify the ranges of the parameter variation:

$$x_j^{\min} \le x_j \le x_j^{\max} \text{ for } 1 \le j \le H \tag{1.34}$$

In order to facilitate the search in the solution space determined by inequalities (1.34), integer strings $\boldsymbol{a} = (a_1\ a_2\ ...\ a_H)$ should be generated with elements ranging from 0 to N and the values of parameters should be obtained for each string as

$$x_j = x_j^{\min} + a_j(x_j^{\max} - x_j^{\min})/N. \tag{1.35}$$

Note that the space of the integer strings just approximately maps the space of the real-valued parameters. The number N determines the precision of the search. The search resolution for the jth parameter is $(x_j^{\max} - x_j^{\min})/N$. Therefore, the increase of N provides a more precise search. On the other hand, the size of the search space of integer strings grows drastically with the increase of N, which slows the GA convergence. A reasonable compromise can be found by using a multistage GA search. In this method, a moderate value of N is chosen and the GA is run to obtain a "crude" solution. Then the ranges of all the parameters are corrected to accomplish the search in a small vicinity of the vector of parameters obtained and the GA is started again. The desired search precision can be obtained by a few iterations.

Example 1.10

Consider a problem in which one has to minimize a function of seven parameters. Assume that following a preliminary decision the ranges of the possible variations of the parameters are different.

Let the random generator provide the generation of integer numbers in the range of 0 -100 (N = 100). The random integer string and the corresponding values of the parameters obtained according to (1.35) are presented in Table 1.3.

Table 1.3. Example of parameters encoding

No. of variable	1	2	3	4	5	6	7
$x_j^{\min}$	0.0	0.0	1.0	1.0	1.0	0.0	0.0
$x_j^{\max}$	3.0	3.0	5.0	5.0	5.0	5.0	5.0
Random integer string	21	4	0	100	72	98	0
Decoded variable	0.63	0.12	1.0	5.0	3.88	4.9	0.0

1.3.2.2 Partition and Allocation Problems

The partition problem can be considered as a problem of allocating Y items belonging to a set Φ in K mutually disjoint subsets Φ_i, *i.e.* such that

$$\bigcup_{i=1}^{K} \Phi_i = \Phi, \quad \Phi_i \bigcap \Phi_j = \varnothing, \; i \neq j \tag{1.36}$$

Each set can contain from 0 to Y items. The partition of the set Φ can be represented by the Y-length string $\boldsymbol{a} = (a_1\ a_2\ ...\ a_{Y-1}\ a_Y)$ in which a_j is a number of the set to which item j belongs. Note that, in the strings representing feasible solutions of the partition problem, each element can take a value in the range (1, K).

Now consider a more complicated allocation problem in which the number of items is not specified. Assume that there are H types of different items with an unlimited number of items for each type h. The number of items of each type allocated in each subset can vary. To represent an allocation of the variable number of items in K subsets one can use the following string encoding $\boldsymbol{a} = (a_{11}\ a_{12}\ ...a_{1K}\ a_{21}\ a_{22}\ ...\ a_{2K}...\ a_{H1}\ a_{H2}...\ a_{HK})$, in which a_{ij} corresponds to the number of items of type i belonging to subset j. Observe that the different subsets can contain identical elements.

Example 1.11

Consider the problem of allocating items of three different types in two disjoint subsets. In this problem, H=3 and K=2. Any possible allocation can be represented by an integer string using the encoding described above. For example, the string (2 1 0 1 1 1) encodes the solution in which two type 1 items are allocated in the first subset and one in the second subset, one item of type 2 is allocated in the second subset, one item of type 3 is allocated in each of the two subsets.

When K = 1, one has an assignment problem in which a number of different items should be chosen from a list containing an unlimited number of items of K different types. Any solution of the assignment problem can be represented by the string $\boldsymbol{a} = (a_1\ a_2\ ...\ a_K)$, in which a_j corresponds to the number of chosen items of type j.

The range of variance of string elements for both allocation and assignment problems can be specified based on the preliminary estimation of the characteristics of the optimal solution (maximal possible number of elements of the same type included into the single subset). The greater the range, the greater the solution space to be explored (note that the minimal possible value of the string element is always zero in order to provide the possibility of not choosing any element of the given type to the given subset). In many practical applications, the total number of items belonging to each subset is also limited. In this case, any string representing a solution in which this constraint is not met should be transformed in the following way:

$$a^*_{ij} = \begin{cases} \left\lfloor a_{ij} N_j / \sum_{h=1}^{H} a_{hj} \right\rfloor, & \text{if } N_j < \sum_{h=1}^{H} a_{hj} \\ a_{ij}, & \text{otherwise} \end{cases} \quad \text{for } 1 \le i \le H, 1 \le j \le K \tag{1.37}$$

where N_j is the maximal allowed number of items in subset j.

Example 1.12

Consider the case in which the items of three types should be allocated into two subsets. Assume that it is prohibited to allocate more than five items of each type to the same subset. The GA should produce strings with elements ranging from 0 to 5. An example of such a string is (4 2 5 1 0 2).

Assume that for some reason the total numbers of items in the first and in the second subsets are restricted to seven and six respectively. In order to obtain a feasible solution, one has to apply the transform (1.37) in which $N_1 = 7$, $N_2 = 6$:

$$\sum_{h=1}^{3} a_{h1} = 4+5+0=9, \quad \sum_{h=1}^{3} a_{h2} = 2+1+2=5$$

The string elements take the values

$$a_{11} = \lfloor 4\times 7/9 \rfloor = 3,\ a_{21} = \lfloor 5\times 7/9 \rfloor = 3,\ a_{31} = \lfloor 0\times 7/9 \rfloor = 0$$

$$a_{12} = \lfloor 2\times 6/5 \rfloor = 2,\ a_{22} = \lfloor 1\times 6/5 \rfloor = 1,\ a_{32} = \lfloor 2\times 6/5 \rfloor = 2$$

After the transformation, one obtains the following string: (3 2 3 1 0 2).

When the number of item types and subsets is large, the solution representation described above results in an enormous growth of the length of the string. Besides, to represent a reasonable solution (especially when the number of items belonging to each subset is limited), such a string should contain a large fraction of zeros because only a few items should be included in each subset. This redundancy causes an increase in the need of computational resources and lowers the efficiency of the GA. To reduce the redundancy of the solution representation, each inclusion of m items of type h into subset k is represented by a triplet (m h k). In order to preserve the constant length of the strings, one has to specify in advance a maximal reasonable number of such inclusions I. The string representing up to I inclusions takes the form (m_1 h_1 k_1 m_2 h_2 k_2 ... m_I h_I k_I). The range of string elements should be (0, max{M, H, K}), where M is the maximal possible number of elements of the same type included into a single subset. An arbitrary string generated in this range can still produce infeasible solutions. In order to provide the feasibility, one has to apply the transform $a^*_j = \mathrm{mod}_{x+1}\, a_j$, where x is equal to M, H and K for the string elements corresponding to m, h and k respectively. If one of the elements of the triplet is equal to zero, then this means that no inclusion is made.

For example, the string (*3 1* 2 1 2 *3 2 1* 1 2 2 *2 3 2*) represents the same allocation as string (3 2 3 1 0 2) in Example 1.12. Note that the permutation of triplets, as well as an addition or reduction of triplets containing zeros, does not change the solution. For example, the string (4 0 1 *2 3 2* 2 1 2 *3 1 1* 1 2 2 *3 2 1*) also represents the same allocation as that of the previous string.

1.3.2.3 Mixed Partition and Parameter Determination Problems

Consider a problem in which Y items should be allocated in K subsets and a value of a certain parameter should be assigned to each item. The first option of representing solutions of such a problem in the GA is by using a $2Y$-length string which takes the form $\boldsymbol{a} = (a_{11}\ a_{12}\ a_{21}\ a_{22}\ ...\ a_{Y1}\ a_{Y2})$. In this string, a_{j1} and a_{j2} correspond respectively to the number of the set the item j belongs to and to the value of the parameter associated with this item. The elements of the string should be generated in the range (0, max$\{K, N\}$), where N is chosen as described in Section 1.3.2.1. The solution decoding procedure should transform the odd elements of the string as follows:

$$a^*_{j1} = 1 + \mathrm{mod}_K\, a_{j1} \tag{1.38}$$

in order to obtain the class number in the range from 1 to K. The even elements of the string should be transformed as follows:

$$a^*_{j2} = \mathrm{mod}_{N+1}\, a_{j2} \tag{1.39}$$

in order to obtain the parameter value encoded by the integer number in the range from 0 to N. The value of the parameter is then obtained using Equation (1.35).

Example 1.13

Consider a problem in which seven items (N = 7) should be allocated to three separated subsets (K = 3) and a value of a parameter associated with each item should be chosen. The solution should encode both items' distribution among the subsets and the parameters. Let the range of the string elements be (0, 100) (N = 100). The string

(*99* 21 *22* 4 *75* 0 *14* 100 *29* 72 *60* 98 *1* 0)

(in which elements corresponding to the numbers of the subsets are marked in italics) represents the solution presented in Table 1.4. The values corresponding to the numbers of the groups are obtained using Equation (1.38) as

$$a^*_{11} = 1 + \mathrm{mod}_K\, a_{11} = 1 + \mathrm{mod}_3\, 99 = 1$$

$$a^*_{21} = 1 + \mathrm{mod}_K\, a_{21} = 1 + \mathrm{mod}_3\, 22 = 2$$

and so on. The numbers that determine the units' weights are obtained using Equation (1.39) as

$$a_{12}^* = \mathrm{mod}_{101}\, a_{12} = \mathrm{mod}_{101}\, 21 = 21$$

$$a_{22}^* = \mathrm{mod}_{101}\, a_{22} = \mathrm{mod}_{101}\, 4 = 4$$

and so on. Observe that, in this solution, items 1, 3, and 6 belong to the first subset, items 2 and 7 belong to the second subset, and items 4 and 5 belong to the third subset. The parameters are identical to those in Example 1.10.

Table 1.4. Example of the solution encoding for the mixed partition and parameter determination problem

No. of unit	1	2	3	4	5	6	7
No. of subset	1	2	1	3	3	1	2
Integer code parameter value	21	4	0	100	72	98	0

This encoding scheme has two disadvantages:

- A large number of different strings can represent an identical solution. Indeed, when K is much smaller than N, many different values of a_{ji} produce the same value of $1+\mathrm{mod}_K\, a_{ji}$ (actually, this transform maps any value $mK+n$ for $n<K$ and $m = 1, 2, \ldots, \lfloor (N-n)/K \rfloor$ into the same number $n+1$). Note for example that the string

($\mathit{3}$ 21 $\mathit{76}$ 4 $\mathit{27}$ 0 $\mathit{29}$ 100 $\mathit{89}$ 72 $\mathit{18}$ 98 $\mathit{70}$)

represents the same solution as the string presented above. This causes a situation where the GA population is overwhelmed with different strings corresponding to the same solution, which misleads the search process.

- The string is quite long, which slows the GA process and increases need for computational resources.

In order to avoid these problems, another solution representation can be suggested that lies in using a Y-length string in which element a_j represents both the number of the set and the value of the parameter corresponding to item j. To obtain such a compound representation, the string elements should be generated in the range (0, $K(N+1)-1$). The number of the subset that element j belongs to should be obtained as

$$1+\lfloor a_j/(N+1) \rfloor \tag{1.40}$$

and the number corresponding to the value of jth parameter should be obtained as

$$\mathrm{mod}_{N+1}\, a_j \tag{1.41}$$

Consider the example presented above with $K = 3$ and $N = 100$. The range of the string elements should be (0, 302). The string

(21 105 0 302 274 98 101)

corresponds to the same solution as the strings in the previous example (Table 1.4).

1.3.2.4 Sequencing Problems

The sequencing problem lies in ordering a group of unique items. It can be considered as a special case of the partition problem in which the number of items Y is equal to the number of subsets K and each subset should not be empty. As in the partition problem, the sequences of items can be represented by Y-length strings ($a_1\ a_2\ ...\ a_{Y-1}\ a_Y$) in which a_j is a number of a set to which item j belongs. However, in the case of the sequencing problem, the string representing a feasible solution should be a permutation of Y integer numbers, *i.e.* it should contain all the numbers from 1 to Y and each number in the string should be unique. While the decoding of such strings is very simple (it just explicitly represents the order of item numbers), the generation procedure should be more sophisticated to satisfy the above-mentioned constraints.

The simplest procedure for generating a random string permutation is as follows:

1. Fill the entire string with zeros.
2. For i from 1 to Y in the sequence:
   2.1. Generate a random number j in the range (1, Y).
   2.2. If $a_j = 0$ assign $a_j = i$ or else find the closest zero element to the right of a_j and assign i to this element (treat the string as a circle, *i.e.* consider a_0 to be the closest element to the right of a_Y).

Like the generation procedures for the partition problem, this one also requires the generation of Y random numbers.

1.3.2.5 Determination of Solution Fitness

Having a solution represented in the GA by an integer string $\boldsymbol{a}$ one then has to estimate the quality of this solution (or, in terms of the evolution process, the fitness of the individual). The GA seeks solutions with the greatest possible fitness. Therefore, the fitness should be defined in such a way that its greatest values correspond to the best solutions.

For example, when optimizing the system reliability R (which is a function of some of the parameters represented by $\boldsymbol{a}$) one can define the solution fitness equal to this index, since one wants to maximize it. On the contrary, when minimizing the system cost C, one has to define the solution fitness as $M - C$, where M is a constant number. In this case, the maximal solution fitness corresponds to its minimal cost.

In the majority of optimization problems, the optimal solution should satisfy some constraints. There are three different approaches to handling the constraints in GA [7]. One of these uses penalty functions as an adjustment to the fitness function; two other approaches use "decoder" or "repair" algorithms to avoid building illegal solutions or repair them respectively. The "decoder" and "repair" approaches suffer from the disadvantage of being tailored to the specific problems and thus are not sufficiently general to handle a variety of problems. On the other hand, the penalty approach based on generating potential solutions without considering the constraints and on decreasing the fitness of solutions, violating the constraints, is suitable for problems with a relatively small number of constraints. For heavily constrained problems, the penalty approach causes the GA to spend most of its time evaluating solutions violating the constraints. Fortunately, the reliability optimization problems usually deal with few constraints.

Using the penalty approach one transforms a constrained problem into an unconstrained one by associating a penalty with all constraint violations. The penalty is incorporated into the fitness function. Thus, the original problem of maximizing a function $f(\boldsymbol{a})$ is transformed into the maximization of the function

$$f(\boldsymbol{a})-\sum_{j=1}^{J}\pi_j\eta_j \tag{1.42}$$

where J is the total number of constraints, π_j is a penalty coefficient related to the jth constraint ($j = 1, \ldots, J$) and η_j is a measure of the constraint violation. Note that the penalty coefficient should be chosen in such a way as to allow the solution with the smallest value of $f(\boldsymbol{a})$ that meets all of the constraints to have a fitness greater than the solution with the greatest value of $f(\boldsymbol{a})$ but violating at least one constraint.

Consider, for example, a typical problem of maximizing the system reliability subject to cost constraint: $R(\boldsymbol{a}) \rightarrow \max$ subject to $C(\boldsymbol{a})\leq C^*$.

The system cost and reliability are functions of parameters encoded by a string $\boldsymbol{a}$: $C(\boldsymbol{a})$ and $R(\boldsymbol{a})$ respectively. The system cost should not be greater than C^*. The fitness of any solution $\boldsymbol{a}$ can be defined as

$$\begin{gathered} M+R(\boldsymbol{a})-\pi\eta(C^*,\boldsymbol{a}) \\ \text{where} \\ \eta(C^*,\boldsymbol{a})=(1+C(\boldsymbol{a})-C^*)1(C(\boldsymbol{a})>C^*) \end{gathered} \tag{1.43}$$

The coefficient π should be greater than one. In this case the fitness of any solution violating the constraint is smaller than M (the smallest violation of the constraint $C(\boldsymbol{a})\leq C^*$ produces a penalty greater than π) while the fitness of any solution meeting the constraint is greater than M. In order to keep the fitness of the solutions positive, one can choose $M>\pi(1+C_{\max}-C^*)$, where $C_{\max}$ is the maximal possible system cost.

Another typical optimization problem is minimizing the system cost subject to the reliability constraint: $C(\boldsymbol{a}) \rightarrow \min$ subject to $R(\boldsymbol{a})\geq R^*$.

The fitness of any solution $\boldsymbol{a}$ of this problem can be defined as

$$\begin{gathered} M-C(\boldsymbol{a})-\pi\eta(R^*,\boldsymbol{a}) \\ \text{where} \\ \eta(A^*,\boldsymbol{a})=(1+R^*-R(\boldsymbol{a}))1(R(\boldsymbol{a})<R^*) \end{gathered} \tag{1.44}$$

The coefficient π should be greater than $C_{\max}$. In this case, the fitness of any solution violating the constraint is smaller than $M-C_{\max}$ whereas the fitness of any solution meeting the constraint is greater than $M - C_{\max}$. In order to keep the fitness of the solutions positive, one can choose $M > C_{\max} + 2\pi$.

1.3.2.6 Basic Genetic Algorithm Procedures and Parameters

The crossover procedures create a new solution as the offspring of a pair of existing ones (parent solutions). The offspring should inherit some useful properties of both parents in order to facilitate their propagation throughout the population. The mutation procedure is applied to the offspring solution. It introduces slight changes into the solution encoding string by modifying some of the string elements. Both of these procedures should be developed in such a way as to provide the feasibility of the offspring solutions given that parent solutions are feasible.

When applied to parameter determination, partition, and assignment problems, the solution feasibility means that the values of all of the string elements belong to a specified range. The most commonly used crossover procedures for these problems generate offspring in which every position is occupied by a corresponding element from one of the parents. This property of the offspring solution provides its feasibility. For example, in the *uniform crossover* each string element is copied either from the first or second parent string with equal probability.

The commonly used mutation procedure changes the value of a randomly selected string element by 1 (increasing or decreasing this value with equal probability). If after the mutation the element is out of the specified range, it takes the minimal or maximal allowed value.

When applied to the sequencing problems, the crossover and mutation operators should produce the offspring that preserve the form of permutations. This means that the offspring string should contain all of the elements that appear in the initial strings and each element should appear in the offspring only once. Any omission or duplication of the element constitutes an error. For example, in the *fragment crossover* operator all of the elements from the first parent string are copied to the same positions of the offspring. Then, all of the elements belonging to a randomly chosen set of adjacent positions in the offspring are reallocated within this set in the order that they appear in the second parent string. It can be seen that this operator provides the feasibility of the permutation solutions.

The widely used mutation procedure that preserves the permutation feasibility swaps two string elements initially located in two randomly chosen positions.

There are no general rules in order to choose the values of basic GA parameters for solving specific optimization problems. The best way to determine the proper combination of these values is by experimental comparison between GAs with different parameters.

A detailed description of a variety of different crossover and mutation operators and recommendations concerning the choice of GA parameters can be found in the GA literature.