6.1 Genetic Algorithms

Genetic algorithms are computerized search and optimization algorithms based on the mechanics of natural genetics and natural selection. Professor John Holland of the University of Michigan, Ann Arbor envisaged the concept of these algorithms in the mid-sixties and published his seminal work (Holland, 1975). Thereafter, a number of his students and other researchers have contributed to developing this field. To date, most of the GA studies are available through a few books (Davis, 1991; Goldberg, 1989; Holland, 1975; Michalewicz, 1992) and through a number of international conference proceedings (Belew and Booker, 1991; Forrest, 1993; Grefenstette, 1985, 1987; Rawlins, 1991; Schaffer, 1989, Whitley, 1993). An extensive list of GA-related papers is referenced elsewhere (Goldberg, et. al, 1992). GAs are fundamentally different than classical optimization algorithms we have discussed in Chapters 2 through 5. We begin the discussion of GAs by first outlining the working principles of GAs and then highlighting the differences GAs have with the traditional search methods. Thereafter, we show a computer simulation to illustrate the working of GAs.

6.1.1 Working principles

To illustrate the working principles of GAs, we first consider an unconstrained optimization problem. Later, we shall discuss how GAs can be used to solve a constrained optimization problem. Let us consider the following maximization problem:

$$\text{Maximize} \quad f(x), \qquad x_i^{(L)} \leq x_i \leq x_i^{(U)}, \ i = 1, 2, \ldots, N.$$

Although a maximization problem is considered here, a minimization problem can also be handled using GAs. The working of GAs is completed by performing the following tasks:

Coding

In order to use GAs to solve the above problem, variables x_i's are first coded in some string structures. It is important to mention here that the coding of the variables is not absolutely necessary. There exist some studies where GAs are directly used on the variables themselves, but here we shall ignore the exceptions and discuss the working principle of a simple genetic algorithm. Binary-coded strings having 1's and 0's are mostly used. The length of the string is usually determined according to the desired solution accuracy. For example, if four bits are used to code each variable in a two-variable function optimization problem, the strings (0000 0000) and (1111 1111) would represent the points

$$(x_1^{(L)}, x_2^{(L)})^T \qquad (x_1^{(U)}, x_2^{(U)})^T,$$

respectively, because the substrings (0000) and (1111) have the minimum and the maximum decoded values. Any other eight-bit string can be found to represent a point in the search space according to a fixed mapping rule. Usually, the following linear mapping rule is used:

$$x_i = x_i^{(L)} + \frac{x_i^{(U)} - x_i^{(L)}}{2^{\ell_i} - 1} \text{ decoded value } (s_i). \tag{6.1}$$

In the above equation, the variable x_i is coded in a substring s_i of length ℓ_i. The decoded value of a binary substring s_i is calculated as $\sum_{i=0}^{\ell-1} 2^i s_i$, where $s_i \in (0,1)$ and the string s is represented as $(s_{\ell-1}s_{\ell-2}\ldots s_2 s_1 s_0)$. For example, a four-bit string (0111) has a decoded value equal to $((1)2^0 + (1)2^1 + (1)2^2 + (0)2^3)$ or 7. It is worthwhile to mention here that with four bits to code each variable, there are only 2^4 or 16 distinct substrings possible, because each bit-position can take a value either 0 or 1. The accuracy that can be obtained with a four-bit coding is only approximately 1/16th of the search space. But as the string length is increased by one, the obtainable accuracy increases exponentially to 1/32th of the search space. It is not necessary to code all variables in equal substring length. The length of a substring representing a variable depends on the desired accuracy in that variable. Generalizing this concept, we may say that with an ℓ_i-bit coding for a variable, the obtainable accuracy in that variable is approximately $(x_i^{(U)} - x_i^{(L)})/2^{\ell_i}$. Once the coding of the variables has been done, the corresponding point $x = (x_1, x_2, \ldots, x_N)^T$ can be found using Equation (6.1). Thereafter, the function value at the point x can also be calculated by substituting x in the given objective function $f(x)$.

Fitness function

As pointed out earlier, GAs mimic the survival-of-the-fittest principle of nature to make a search process. Therefore, GAs are naturally suitable for solving maximization problems. Minimization problems are usually transformed into maximization problems by some suitable transformation. In general, a *fitness* function $\mathcal{F}(x)$ is first derived from the objective function and used in successive genetic operations. Certain genetic operators require that the fitness function be nonnegative, although certain operators do not have this requirement. For maximization problems, the fitness function can be considered to be the same as the objective function or $\mathcal{F}(x) = f(x)$. For minimization problems, the fitness function is an equivalent maximization problem chosen such that the optimum point remains unchanged. A number of such transformations are possible. The following fitness function is often used:

$$\mathcal{F}(x) = 1/(1 + f(x)). \tag{6.2}$$

This transformation does not alter the location of the minimum, but converts a minimization problem to an equivalent maximization problem. The fitness function value of a string is known as the string's *fitness*.

The operation of GAs begins with a population of random strings representing design or decision variables. Thereafter, each string is evaluated to find the fitness value. The population is then operated by three main operators—*reproduction*, *crossover*, and *mutation*—to create a new population of points. The new population is further evaluated and tested for termination. If the termination criterion is not met, the population is iteratively operated by the above three operators and evaluated. This procedure is continued until the termination criterion is met. One cycle of these operations and the subsequent evaluation procedure is known as a *generation* in GA's terminology. The operators are described next.

GA operators

Reproduction is usually the first operator applied on a population. Reproduction selects good strings in a population and forms a mating pool. That is why the reproduction operator is sometimes known as the selection operator. There exist a number of reproduction operators in GA literature, but the essential idea in all of them is that the above-average strings are picked from the current population and their multiple copies are inserted in the mating pool in a probabilistic manner. The commonly-used reproduction operator is the proportionate reproduction operator where a string is selected for the mating pool with a probability proportional to its fitness. Thus, the i-th string in the population is selected with a probability proportional to $\mathcal{F}_i$. Since the population size is usually kept fixed in a simple GA, the sum of the probability of each string being selected for the mating pool must be one. Therefore, the probability for selecting the i-th string is

$$p_i = \frac{\mathcal{F}_i}{\sum\limits_{j=1}^{n} \mathcal{F}_j},$$

where n is the population size. One way to implement this selection scheme is to imagine a roulette-wheel with it's circumference marked for each string proportionate to the string's fitness. The roulette-wheel is spun n times, each time selecting an instance of the string chosen by the roulette-wheel pointer. Since the circumference of the wheel is marked according to a string's fitness, this roulette-wheel mechanism is expected to make $\mathcal{F}_i/\overline{\mathcal{F}}$ copies of the i-th string in the mating pool. The average fitness of the population is calculated as

$$\overline{\mathcal{F}} = \sum_{i=1}^{n} \mathcal{F}_i/n.$$

Figure 6.1 shows a roulette-wheel for five individuals having different fitness values. Since the third individual has a higher fitness value

Point	*Fitness*
1	25.0
2	5.0
3	40.0
4	10.0
5	20.0

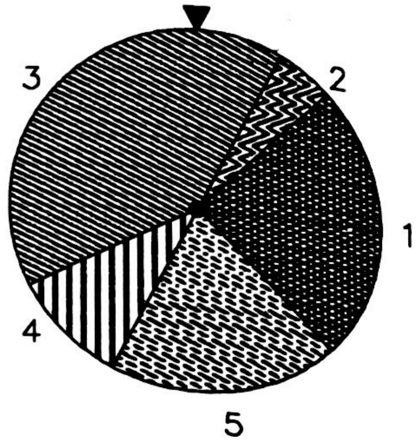

Figure 6.1 A roulette-wheel marked for five individuals according to their fitness values. The third individual has a higher probability of selection than any other.

than any other, it is expected that the roulette-wheel selection will choose the third individual more than any other individual. This roulette-wheel selection scheme can be simulated easily. Using the fitness value $\mathcal{F}_i$ of all strings, the probability of selecting a string p_i can be calculated. Thereafter, the cumulative probability (P_i) of each string being copied can be calculated by adding the individual probabilities from the top of the list. Thus, the bottom-most string in the population should have a cumulative probability (P_n) equal to 1. The roulette-wheel concept can be simulated by realizing that the i-th string in the population represents the cumulative probability values from P_{i-1} to P_i. The first string represents the cumulative values from zero to P_1. Thus, the cumulative probability of any string lies between 0 to 1. In order to choose n strings, n random numbers between zero to one are created at random. Thus, a string that represents the chosen random number in the cumulative probability range (calculated from the fitness values) for the string is copied to the mating pool. This way, the string with a higher fitness value will represent a larger range in the cumulative probability values and therefore has a higher probability of being copied into the mating pool. On the other hand, a string with a smaller fitness value represents a smaller range in cumulative probability values and has a smaller probability of being copied into the mating pool. We illustrate the working of this roulette-wheel simulation later through a computer simulation of GAs.

In reproduction, good strings in a population are probabilistically assigned a larger number of copies and a mating pool is formed. It is important to note that no new strings are formed in the reproduction phase. In the crossover operator, new strings are created by exchanging information among strings of the mating pool. Many crossover operators exist in the GA literature. In most crossover operators, two strings are picked from the mating pool at random and some portions of the strings are exchanged between the strings. A single-point crossover operator is performed by randomly choosing a crossing site along the string and by exchanging all bits on the right side of the crossing site as shown:

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

The two strings participating in the crossover operation are known as parent strings and the resulting strings are known as children strings. It is intuitive from this construction that good substrings from parent strings can be combined to form a better child string, if an appropriate site is chosen. Since the knowledge of an appropriate site is usually not known beforehand, a random site is often chosen. With a random site, the children strings produced may or may not have a combination of good substrings from parent strings, depending on whether or not the crossing site falls in the appropriate place. But we do not worry about this too much, because if good strings are created by crossover, there will be more copies of them in the next mating pool generated by the reproduction operator. But if good strings are not created by crossover, they will not survive too long, because reproduction will select against those strings in subsequent generations.

It is clear from this discussion that the effect of crossover may be detrimental or beneficial. Thus, in order to preserve some of the good strings that are already present in the mating pool, not all strings in the mating pool are used in crossover. When a crossover probability of p_c is used, only $100p_c$ per cent strings in the population are used in the crossover operation and $100(1-p_c)$ per cent of the population remains as they are in the current population[1].

A crossover operator is mainly responsible for the search of new strings, even though a mutation operator is also used for this purpose sparingly. The mutation operator changes 1 to 0 and vice versa with a small mutation probability, p_m. The bit-wise mutation is performed bit by bit by flipping a coin[2] with a probability p_m. If at any bit the outcome is true then the bit is altered; otherwise the bit is kept unchanged. The need for mutation is to create a point in the neighbourhood of the current point, thereby achieving a local search around the current solution. The mutation is also used to maintain diversity in the population. For example, consider the following population having four eight-bit strings:

```
0110 1011
0011 1101
0001 0110
0111 1100
```

Notice that all four strings have a 0 in the left-most bit position. If the true optimum solution requires 1 in that position, then neither reproduction nor crossover operator described above will be able to create 1 in that position. The inclusion of mutation introduces some probability (Np_m) of turning 0 into 1.

These three operators are simple and straightforward. The reproduction operator selects good strings and the crossover operator recombines good substrings from good strings together to hopefully create a better substring. The mutation operator alters a string locally to hopefully create a better string. Even though none of these claims are guaranteed and/or tested while creating a string, it is expected that if bad strings are created they will be eliminated by the reproduction operator in the next generation and if good strings are created, they will be increasingly emphasized. Interested readers may refer to Goldberg (1989) and other GA literature given in the references for further insight and some mathematical foundations of genetic algorithms.

Here, we outline some differences and similarities of GAs with traditional optimization methods.

[1] Even though the best $(1 - p_c)100\%$ of the current population can be copied deterministically to the new population, this is usually performed at random.

[2] Flipping of a coin with a probability p is simulated as follows. A number between 0 to 1 is chosen at random. If the random number is smaller than p, the outcome of coin-flipping is true, otherwise the outcome is false.

EXERCISE 6.1.1

The objective is to minimize the function

$$f(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$$

in the interval $0 \leq x_1, x_2 \leq 6$. Recall that the true solution to this problem is $(3, 2)^T$ having a function value equal to zero.

Step 1 In order to solve this problem using genetic algorithms, we choose binary coding to represent variables x_1 and x_2. In the calculations here, 10-bits are chosen for each variable, thereby making the total string length equal to 20. With 10 bits, we can get a solution accuracy of $(6-0)/(2^{10}-1)$ or 0.006 in the interval $(0, 6)$. We choose roulette-wheel selection, a single-point crossover, and a bit-wise mutation operator. The crossover and mutation probabilities are assigned to be 0.8 and 0.05, respectively. We decide to have 20 points in the population. The random population created using Knuth's (1981) random number generator[3] with a random seed equal to 0.760 is shown in Table 6.1. We set $t_{\max} = 30$ and initialize the generation counter $t = 0$.

Step 2 The next step is to evaluate each string in the population. We calculate the fitness of the first string. The first substring (1100100000) decodes to a value equal to $(2^9 + 2^8 + 2^5)$ or 800. Thus, the corresponding parameter value is equal to $0 + (6 - 0) \times 800/1023$ or 4.692. The second substring (1110010000) decodes to a value equal to $(2^9 + 2^8 + 2^7 + 2^4)$ or 912. Thus, the corresponding parameter value is equal to $0 + (6 - 0) \times 912/1023$ or 5.349. Thus, the first string corresponds to the point $x^{(1)} = (4.692, 5.349)^T$. These values can now be substituted in the objective function expression to obtain the function value. It is found that the function value at this point is equal to $f(x^{(1)}) = 959.680$. We now calculate the fitness function value at this point using the transformation rule: $\mathcal{F}(x^{(1)}) = 1.0/(1.0 + 959.680) = 0.001$. This value is used in the reproduction operation. Similarly, other strings in the population are evaluated and fitness values are calculated. Table 6.1 shows the objective function value and the fitness value for all 20 strings in the initial population.

Step 3 Since $t = 0 < t_{\max} = 30$, we proceed to Step 4.

---

[3]A FORTRAN code implementing the random number generator appears in the GA code presented at the end of this chapter.

Table 6.1 Evaluation and Reproduction Phases on a Random Population

	String												Mating pool	
	Substring-2	Substring-1	x_2	x_1	$f(x)$	$\mathcal{F}(x)$	A	B	C	D	E	F	Substring-2	Substring-1
1	1110010000	1100100000	5.349	4.692	959.680	0.001	0.13	0.007	0.007	0.472	10	0	0010100100	1010101010
2	0001001101	0011100111	0.452	1.355	105.520	0.009	1.10	0.055	0.062	0.108	3	1	1010100001	0111001000
3	1010100001	0111001000	3.947	2.674	126.685	0.008	0.98	0.049	0.111	0.045	2	1	0001001101	0011100111
4	1001000110	1000010100	3.413	3.120	65.026	0.015	1.85	0.093	0.204	0.723	14	2	1110011011	0111000010
5	1100011000	1011100011	4.645	4.334	512.197	0.002	0.25	0.013	0.217	0.536	10	0	0010100100	1010101010
6	0011100101	0011111000	1.343	1.455	70.868	0.014	1.71	0.086	0.303	0.931	19	2	0011100010	1011000011
7	0101011011	0000000111	2.035	0.041	88.273	0.011	1.34	0.067	0.370	0.972	19	1	0011100010	1011000011
8	1110101000	1110101011	5.490	5.507	1436.563	0.001	0.12	0.006	0.376	0.817	17	0	0111000010	1011000110
9	1001111101	1011100111	3.736	4.358	265.556	0.004	0.49	0.025	0.401	0.363	7	1	0101011011	0000000111
10	0010100100	1010101010	0.962	4.000	39.849	0.024	2.96	0.148	0.549	0.189	4	3	1001000110	1000010100
11	1111101001	0001110100	5.871	0.680	814.117	0.001	0.14	0.007	0.556	0.220	6	0	0011100101	0011111000
12	0000111101	0110011101	0.358	2.422	42.598	0.023	2.84	0.142	0.698	0.288	6	3	0011100101	0011111000
13	0000111110	1110001101	0.364	5.331	318.746	0.003	0.36	0.018	0.716	0.615	12	1	0000111101	0110011101
14	1110011011	0111000010	5.413	2.639	624.164	0.002	0.24	0.012	0.728	0.712	13	1	0000111110	1110001101
15	1010111010	1010111000	4.094	4.082	286.800	0.003	0.37	0.019	0.747	0.607	12	0	0000111101	0110011101
16	0100011111	1100111000	1.683	4.833	197.556	0.005	0.61	0.030	0.777	0.192	4	0	1001000110	1000010100
17	0111000010	1011000110	2.639	4.164	97.699	0.010	1.22	0.060	0.837	0.386	9	1	1001111101	1011100111
18	1010010100	0100001001	3.871	1.554	113.201	0.009	1.09	0.054	0.891	0.872	18	1	1010010100	0100001001
19	0011100010	1011000011	1.326	4.147	57.753	0.017	2.08	0.103	0.994	0.589	12	2	0000111101	0110011101
20	1011100011	1111010000	4.334	5.724	987.955	0.001	0.13	0.006	1.000	0.413	10	0	0010100100	1010101010

A : Expected count
B : Probability of selection
C : Cumulative probability of selection
D : Random number between 0 and 1
E : String number
F : True count in the mating pool

Step 4 At this step, we select good strings in the population to form the mating pool. In order to use the roulette-wheel selection procedure, we first calculate the average fitness of the population. By adding the fitness values of all strings and dividing the sum by the population size, we obtain $\overline{\mathcal{F}} = 0.008$. The next step is to compute the expected count of each string as $\mathcal{F}(x)/\overline{\mathcal{F}}$. The values are calculated and shown in column A of Table 6.1. In other words, we can compute the probability of each string being copied in the mating pool by dividing these numbers with the population size (column B). Once these probabilities are calculated, the cumulative probability can also be computed. These distributions are also shown in column C of Table 6.1. In order to form the mating pool, we create random numbers between zero and one (given in column D) and identify the particular string which is specified by each of these random numbers. For example, if the random number 0.472 is created, the tenth string gets a copy in the mating pool, because that string occupies the interval (0.401, 0.549), as shown in column C. Column E refers to the selected string. Similarly, other strings are selected according to the random numbers shown in column D. After this selection procedure is repeated n times (n is the population size), the number of selected copies for each string is counted. This number is shown in column F. The complete mating pool is also shown in the table. Columns A and F reveal that the theoretical expected count and the true count of each string more or less agree with each other. Figure 6.5 shows the initial random population and the mating pool after reproduction. The points marked with an enclosed box are the points in the mating pool. The action of the reproduction operator is clear from this plot. The inferior points have been probabilistically eliminated from further consideration. Notice that not all selected points are better than all rejected points. For example, the 14th individual (with a fitness value 0.002) is selected but the 16th individual (with a function value 0.005) is not selected.

Although the above roulette-wheel selection is easier to implement, it is noisy. A more stable version of this selection operator is sometimes used. After the expected count for each individual string is calculated, the strings are first assigned copies exactly equal to the mantissa of the expected count. Thereafter, the regular roulette-wheel selection is implemented using the decimal part of the expected count as the probability of selection. This selection method is less noisy and is known as the *stochastic remainder* selection.

Step 5 At this step, the strings in the mating pool are used in the crossover operation. In a single-point crossover, two strings are selected at random and crossed at a random site. Since the mating

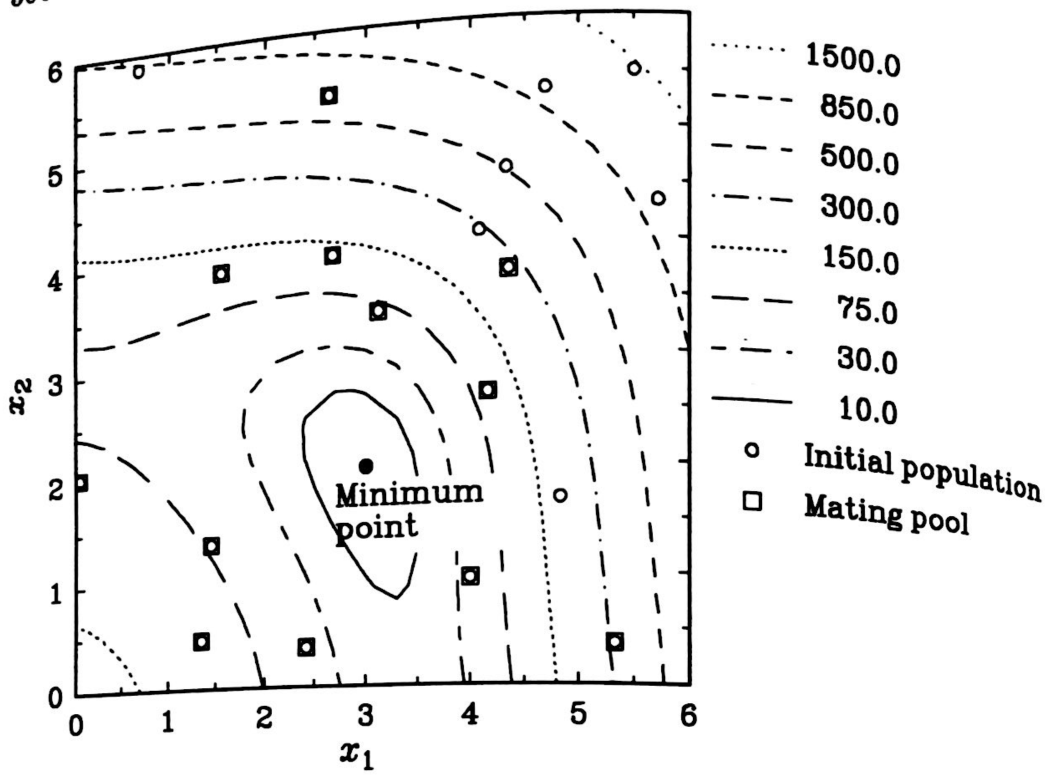

Figure 6.5 The initial population (marked with empty circles) and the mating pool (marked with boxes) on a contour plot of the objective function. The best point in the population has a function value 39.849 and the average function value of the initial population is 360.540.

pool contains strings at random, we pick pairs of strings from the top of the list. Thus, strings 3 and 10 participate in the first crossover operation. When two strings are chosen for crossover, first a coin is flipped with a probability $p_c = 0.8$ to check whether a crossover is desired or not. If the outcome of the coin-flipping is true, the crossing over is performed, otherwise the strings are directly placed in an intermediate population for subsequent genetic operation. It turns out that the outcome of the first coin-flipping is true, meaning that a crossover is required to be performed. The next step is to find a cross-site at random. We choose a site by creating a random number between $(0, \ell - 1)$ or $(0, 19)$. It turns out that the obtained random number is 11. Thus, we cross the strings at the site 11 and create two new strings. After crossover, the children strings are placed in the intermediate population. Then, strings 14 and 2 (selected at random) are used in the crossover operation. This time the coin-flipping comes true again and we perform the crossover at the site 8 found at random. The new children strings are put into the intermediate population. Figure 6.6 shows how points cross over and form new points. The points marked with a small box are the points in the mating pool and the points marked with a small circle are children points created after crossover operation. Notice that not

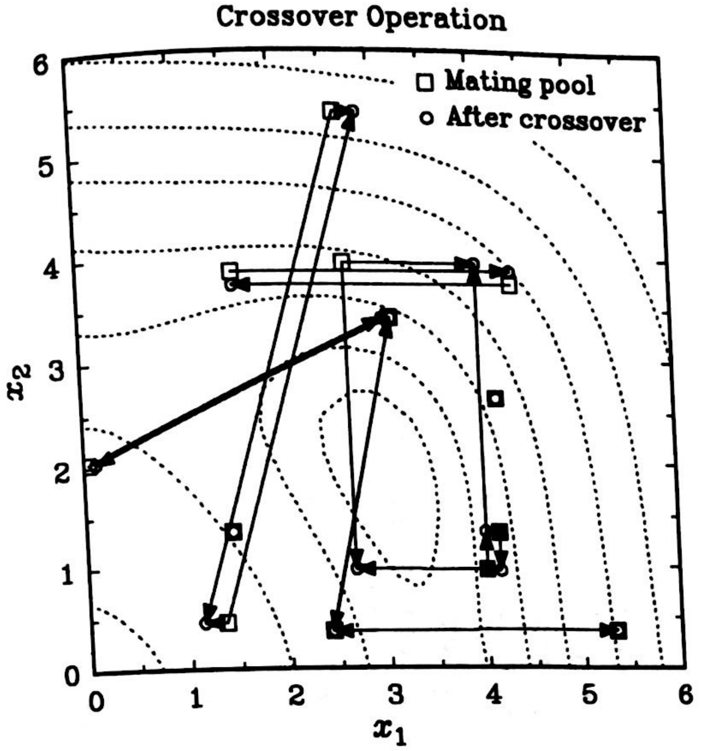

Figure 6.6 The population after the crossover operation. Two points are crossed over to form two new points. Of ten pairs of strings, seven pairs are crossed.

all 10 pairs of points in the mating pool cross with each other. With the flipping of a coin with a probability $p_c = 0.8$, it turns out that fourth, seventh, and tenth crossovers come out to be false. Thus, in these cases, the strings are copied directly into the intermediate population. The complete population at the end of the crossover operation is shown in Table 6.2. It is interesting to note that with $p_c = 0.8$, the expected number of crossover in a population of size 20 is $0.8 \times 20/2$ or 8. In this exercise problem, we performed seven crossovers and in three cases we simply copied the strings to the intermediate population. Figure 6.6 shows that some good points and some not-so-good points are created after crossover. In some cases, points far away from the parent points are created and in some cases points close to the parent points are created.

Step 6 The next step is to perform mutation on strings in the intermediate population. For bit-wise mutation, we flip a coin with a probability $p_m = 0.05$ for every bit. If the outcome is true, we alter the bit to 1 or 0 depending on the bit value. With a probability of 0.05, a population size 20, and a string length 20, we can expect to alter a total of about $0.05 \times 20 \times 20$ or 20 bits in the population. Table 6.2 shows the mutated bits in bold characters in the table. As counted from the table, we have actually altered 16 bits. Figure 6.7

Table 6.2 Crossover and Mutation Operators

Mating pool				Intermediate population		Mutation					
Substring-2	*Substring-1*	*G*	*H*	Substring-2	Substring-1	Substring-2	Substring-1	x_1	x_2	$f(x)$	$\mathcal{F}(x)$
0010100100	1010101010	Y	9	0010100101	0111001000	0010101101	0111001000	1.015	2.674	18.886	0.050
1010100001	0111001000	Y	9	1010100000	1010101010	1010100001	1010101010	3.947	4.000	238.322	0.004
0001001101	0011100111	Y	12	0001001101	0011000010	0001001101	0001000010	0.452	0.387	149.204	0.007
1110011011	0111000010	Y	12	1110011011	0111100111	1110011011	0101100011	5.413	2.082	596.340	0.002
0010100100	1010101010	Y	5	0010100010	1011000011	0010100010	1011000011	0.950	4.147	54.851	0.018
0011100010	1011000011	Y	5	0011100100	1010101010	0011100100	1110101010	1.337	5.501	424.583	0.002
0011100010	1011000011	N		0011100010	1011000011	0011100011	1011100011	1.331	4.334	83.929	0.012
0111000010	1011000110	N		0111000010	1011000110	0101010010	1011000110	1.982	4.164	70.472	0.014
0101011011	0000000111	Y	14	0101011011	0000010100	0101011011	0000010100	2.035	0.117	87.633	0.011
1001000110	1000010100	Y	14	1001000110	1000000111	1001010110	1000000111	3.507	3.044	72.789	0.014
0011100101	0011111000	Y	1	0011100101	0011111000	0011100101	0011111000	1.343	1.455	70.868	0.014
0011100101	0011111000	Y	1	0011100101	0011111000	0011100101	0011111000	1.343	1.455	70.868	0.014
0000111101	0110011101	N		0000111101	0110011101	0000101101	0111011100	0.264	2.792	25.783	0.037
0000111110	1110001101	N		0000111110	1110001101	0000111110	1110001101	0.364	5.331	318.746	0.003
0000111101	0110011101	Y	18	0000111101	0110011100	0000111101	0110011100	0.358	2.416	42.922	0.023
1001000110	1000010100	Y	18	1001000110	1000010101	1001000110	0000010101	3.413	0.123	80.127	0.012
1001111101	1011100111	Y	10	1001111101	0100001001	1001111101	0100001001	3.736	1.554	95.968	0.010
1010010100	0100001001	Y	10	1010010100	1011100111	1010010100	1010100111	3.871	3.982	219.426	0.005
0000111101	0110011101	N		0000111101	0110011101	0000111101	0110011101	0.358	2.422	42.598	0.023
0010100100	1010101010	N		0010100100	1010101010	0010100100	1010101010	0.962	4.000	39.849	0.024

G : Whether crossover (Y yes, N no), *H* : Crossing site shows the effect of mutation on the intermediate population. In

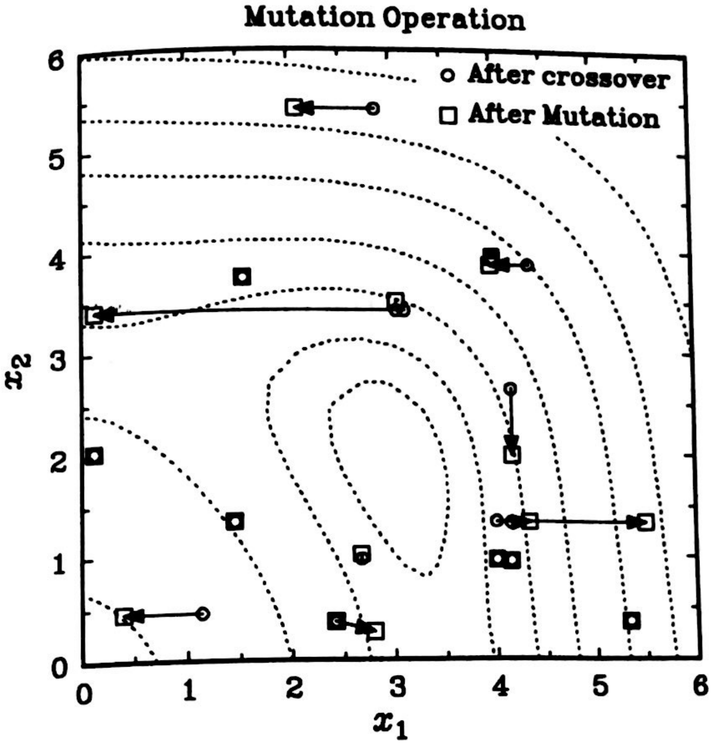

Figure 6.7 The population after mutation operation. Some points do not get mutated and remain unaltered. The best point in the population has a function value 18.886 and the average function value of the population is 140.210, an improvement of over 60 per cent.

some cases, the mutation operator changes a point locally and in some other it can bring a large change. The points marked with a small circle are points in the intermediate population. The points marked with a small box constitute the new population (obtained after reproduction, crossover, and mutation). It is interesting to note that if only one bit is mutated in a string, the point is moved along a particular variable only. Like the crossover operator, the mutation operator has created some points better and some points worse than the original points. This flexibility enables GA operators to explore the search space properly before converging to a region prematurely. Although this requires some extra computation, this flexibility is essential to solve global optimization problems.

Step 7 The resulting population becomes the new population. We now evaluate each string as before by first identifying the substrings for each variable and mapping the decoded values of the substrings in the chosen intervals. This completes one iteration of genetic algorithms. We increment the generation counter to $t = 1$ and proceed to Step 3 for the next iteration. The new population after one iteration of GAs is shown in Figure 6.7 (marked with empty boxes). The figure shows that in one iteration, some good points have been found. Table 6.2 also shows the fitness values and objective function values of the new population members.

The average fitness of the new population is calculated to be 0.015, a remarkable improvement from that in the initial population (recall that the average in the initial population was 0.008). The best point in this population is found to have a fitness equal to 0.050, which is also better than that in the initial population (0.024). This process continues until the maximum allowable generation is reached or some other termination criterion is met. The population after 25 generation is shown in Figure 6.8. At this generation, the best point

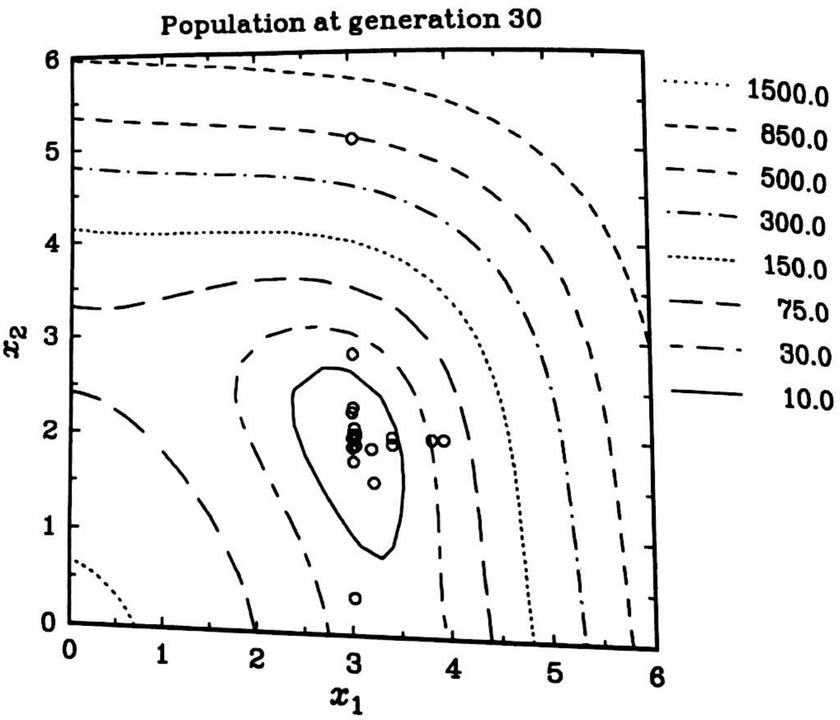

Figure 6.8 All 20 points in the population at generation 25 shown on the contour plot of the objective function. The figure shows that most points are clustered around the true minimum.

is found to be $(3.003, 1.994)^T$ with a function value 0.001. The fitness value at this point is equal to 0.999 and the average population fitness of the population is 0.474. The figure shows how points are clustered around the true minimum of the function in this generation. A few inferior points are still found in the plot. They are the result of some unsuccessful crossover events. We also observe that the total number of function evaluations required to obtain this solution is $0.8\times20\times26$ or 416 (including the evaluations of the initial population).

evaluations. At the end of 30 generations, the total function evaluations required were 837 (including the evaluation of the initial population). It is worth mentioning here that no effort is made to optimally set the GA parameters to obtain the above solution. It is anticipated that with a proper choice of GA parameters, a better result may have been obtained. Nevertheless, for a comparable number of function evaluations, GAs have found a population near the true optimum.

6.1.5 Other GA operators

There exist a number of variations to GA operators. In most cases, the variants are developed to suit particular applications. Nevertheless, there are some variants which are developed in order to achieve some fundamental change in the working of GAs. Here we discuss two such variants for reproduction and crossover operators.

It has been discussed earlier that the roulette-wheel selection operator has inherent noise in selecting good individuals. Although this noise can be somewhat reduced by using stochastic remainder selection, there are two other difficulties with these selection operators. If a population contains an exceptionally good individual early on in the simulation[4], the expected number of copies in the mating pool ($\mathcal{F}_i/\overline{\mathcal{F}}$) may be so large that the individual occupies most of the mating pool. This reduces the diversity in the mating pool and causes GAs to prematurely converge to a wrong solution. On the other hand, the whole population usually contains equally good points later in the simulation. This may cause each individual to have a copy in the mating pool, thereby making a directionless search. Both these difficulties can be eliminated by transforming the fitness function $\mathcal{F}(x)$ to a scaled fitness function $\mathcal{S}(x)$ at every generation. The transformation could be a simple linear transformation

$$\mathcal{S}(x) = a\mathcal{F}(x) + b.$$

The parameters a and b should be defined to allocate the best individual in the population a predefined number of copies in the mating pool and to allocate an average individual one copy in the mating pool (Goldberg, 1989). Since this transformation is performed at every iteration, both difficulties can be eliminated by using this scaling procedure. Another way to overcome the above difficulty is to use a different selection algorithm altogether.

[4]This may happen in constrained optimization problems where the population may primarily contain infeasible points except a few feasible points.

Tournament selection works by first picking s individuals (with or without replacement) from the population and then selecting the best of the chosen s individuals. If performed without replacement in a systematic way[5], this selection scheme can assign exactly s copies of the best individual to the mating pool at every generation. The control of this selection *pressure* in tournament selection is making it popular in recent GA applications. In most GA applications, a binary tournament selection with $s = 2$ is used.

In trying to solve problems with many variables, the single-point crossover operator described earlier may not provide adequate search. Moreover, the single-point crossover operator has some bias of exchange for the right-most bits. They have a higher probability of getting exchanged than the left-most bits in the string. Thus, if ten variables are coded left to right with the first variable being at the left-most position and the tenth variable at the right-most position, the effective search on the tenth variable is more compared to the first variable. In order to overcome this difficulty, a multipoint crossover is often used. The operation of a two-point crossover operator is shown below:

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

Two random sites are chosen along the string length and bits inside the cross-sites are swapped between the parents. An extreme of the above crossover operator is to have a *uniform* crossover operator where a bit at any location is chosen from either parent with a probability 0.5. In the following, we show the working of a uniform crossover operator, where the first and the fourth bit positions have been exchanged.

$$\begin{array}{c} 0\,0\,0\,0\,0 \\ 1\,1\,1\,1\,1 \end{array} \Rightarrow \begin{array}{c} 1\,0\,0\,1\,0 \\ 0\,1\,1\,0\,1 \end{array}$$

This operator has the maximum search power among all of the above crossover operators. Simultaneously, this crossover has the minimum

---

[5]First the population is shuffled. Thereafter, the first s copies are picked from the top of the shuffled list and the best is chosen for the mating pool. Then, the next s individuals (numbered $(s+1)$ to $2s$ in the shuffled list) are picked and the best is chosen for the mating pool. This process is continued until all population members are considered once. The whole population is shuffled again and the same procedure is repeated. This is continued until the complete mating pool is formed.