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

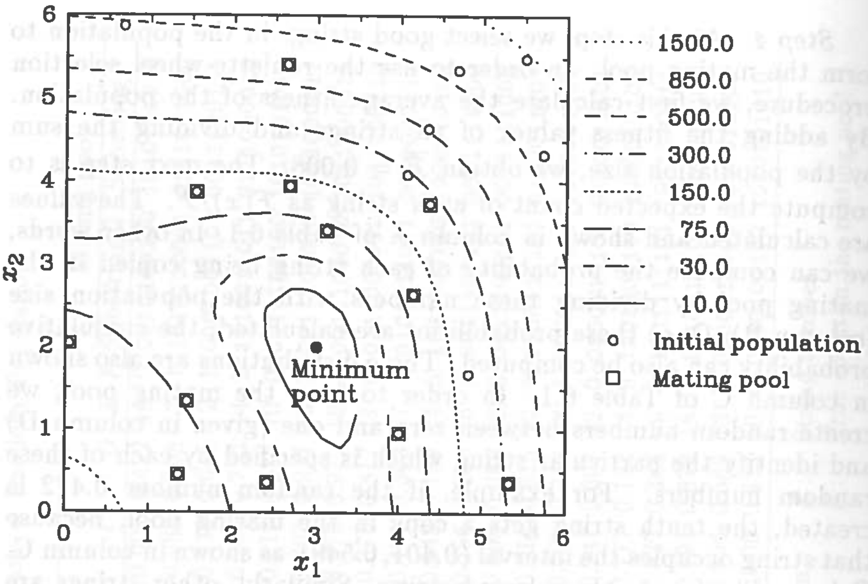

Figure 6.5 The initial population (marked with empty circles) and the mating pool (marked with boxes) on a contour plot of the objective function. The best point in the population has a function value 39.849 and the average function value of the initial population is 360.540.

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

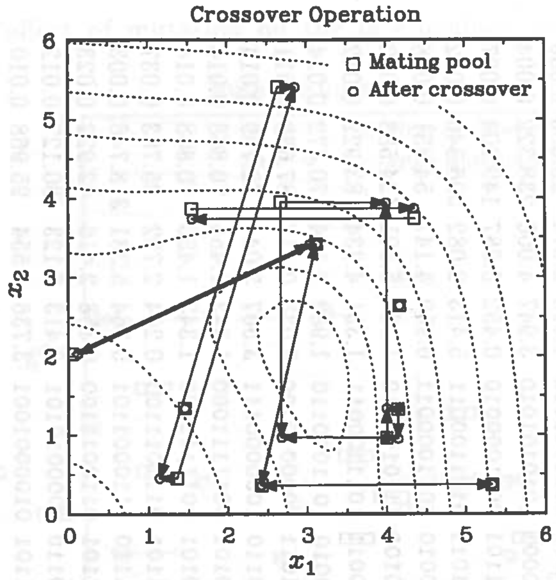

Figure 6.6 The population after the crossover operation. Two points are crossed over to form two new points. Of ten pairs of strings, seven pairs are crossed.

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

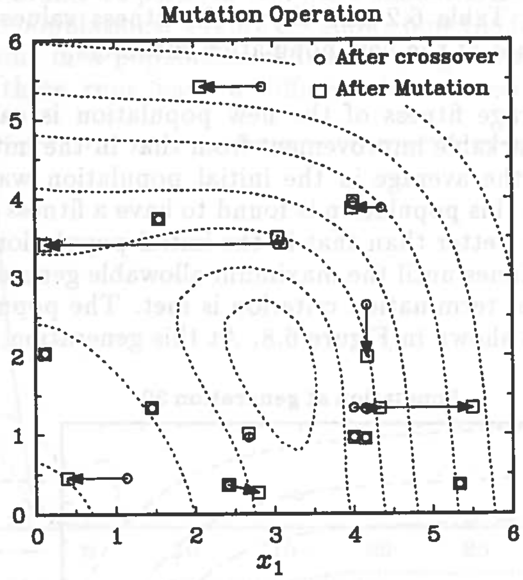

Figure 6.7 The population after mutation operation. Some points do not get mutated and remain unaltered. The best point in the population has a function value 18.886 and the average function value of the population is 140.210, an improvement of over 60 per cent.

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

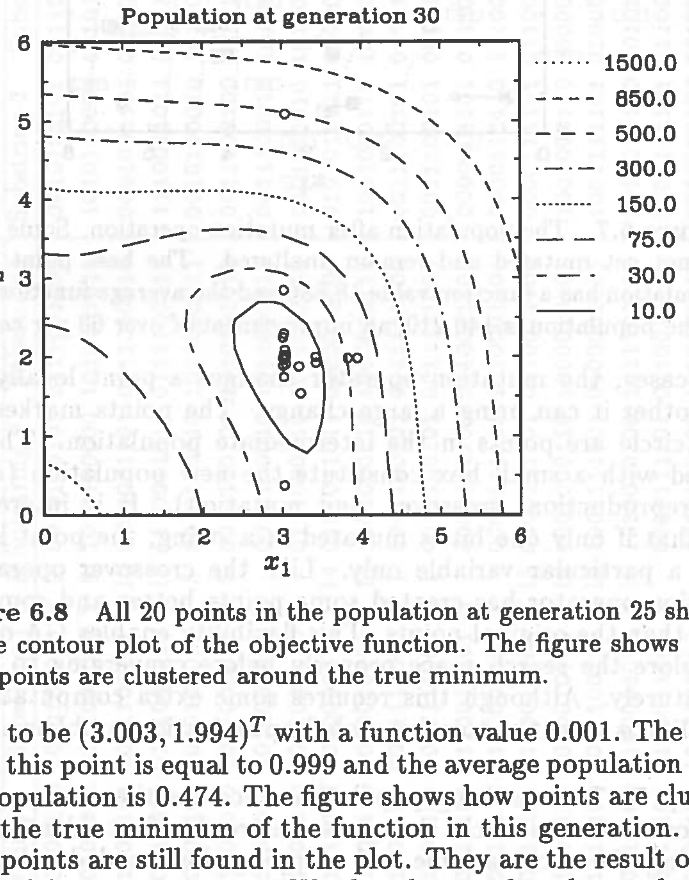

Figure 6.8 All 20 points in the population at generation 25 shown on the contour plot of the objective function. The figure shows that most points are clustered around the true minimum.

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 $0.8 \times 20 \times 26$ or 416 (including the evaluations of the initial population).

In order to show the efficiency of GAs in arriving at a point close to the true optimum, we perform two more simulations starting with different initial populations. Figure 6.9 shows how the function value of the best point in a population reduces with generation number. Although all three runs have a different initial best point, they

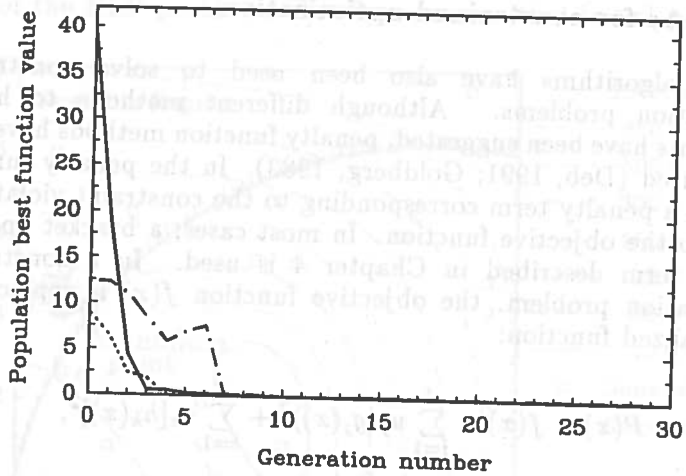

Figure 6.9 The function value of the best point in the population for three independent GA runs. All runs quickly converge to a point near the true optimum.

quickly converge to a solution close to the true optimum (recall that the optimum point has a function value equal to zero).

In order to illustrate the schema processing through genetic operators, we investigate the growth of a particular schema $H = (0 * \ldots * \; * \ldots *)$. This schema represents all points in the range $0 \leq x_2 < 3$. The optimum point lies in this region. With reference to Equation (6.3), we observe that the order, defining length, and the fitness of the schema are such that it is a building block. This schema contains more good points than it's competitor $H^c = (1 * \ldots * \; * \ldots *)$ which represents the range $3 \leq x_2 \leq 6$. According to Equation (6.3), the schema H must increase exponentially due to the action of genetic operators. We observe that in the random initial population the schema H has nine strings and the schema H^c has 11 strings. At the end of one generation, the population has 14 strings representing the schema H and only six strings representing the schema H^c. We may also investigate other interesting regions in the search space and observe their growth in terms of the number of representative points in the population. Other low-order and above-average schemata are also processed similarly and are combined to form higher-order and good schemata. This processing of several schemata happens in parallel without any extra book-keeping (Goldberg, 1989). Eventually, this processing forms the optimum or a near-optimum point.

6.1.4 GAs for constrained optimization

Genetic algorithms have also been used to solve constrained optimization problems. Although different methods to handle constraints have been suggested, penalty function methods have been mostly used (Deb, 1991; Goldberg, 1983). In the penalty function method, a penalty term corresponding to the constraint violation is added to the objective function. In most cases, a bracket operator penalty term described in Chapter 4 is used. In a constrained minimization problem, the objective function $f(x)$ is replaced by the penalized function:

$$P(x) = f(x) + \sum_{j=1}^{J} u_j \langle g_j(x) \rangle^2 + \sum_{k=1}^{K} v_k [h_k(x)]^2, \tag{6.4}$$

where u_j and v_k are penalty coefficients, which are usually kept constant throughout GA simulation. The fitness function is formed by the usual transformation: $\mathcal{F}(x) = 1/(1 + P(x))$. Since GAs are population based search techniques, the final population converges to a region, rather than a point as depicted in the simulation on Himmelblau's function in Exercise 6.1.1. Unlike the penalty function method described in Chapter 4, the update of penalty parameters in successive sequences is not necessary with GAs. Recall that in the traditional penalty function method (described in Chapter 4), the increase of penalty parameter R in successive sequences distorted the penalized function. In some occasions, this distortion may create some artificial local optima. This causes the traditional penalty function method difficult to solve a constrained optimization problem in a single sequence. Since GAs can handle multimodal functions better than the traditional methods, a large penalty parameter R can be used. Since the exact optimum point can only be obtained with an infinite value of penalty parameter R, in most cases the GA solution would be close to the true optimum. With a solution close to the true optimum, an arbitrary large value of R can be used with the steepest descent method to find the optimum point in only one sequence of the penalty function method. In order to illustrate this procedure, we reconsider the NLP problem described in Exercise 4.2.1.