CHAPTER 2

GENETIC ALGORITHMS FOR OPTIMIZATION PROBLEMS

2.1 INTRODUCTION

Since the idea of genetic algorithms was introduced by John Holland in the early 1970's [27], GAs have been applied to a lot of optimization problems. Because they are not limited by restrictive assumptions about search space which concern continuity, existence of derivatives, unimodality, they are utilized by a lot of researchers who are tackling optimization problems.

Search and optimization techniques can be categorized into three classes: calculus based, enumerative, and random. Calculus based approaches usually require the existence of derivatives and the continuity. Therefore it is difficult to apply them to realistic problems where these assumptions often do not hold. Enumerative methods are straightforward search schemes. They can be applied to optimization problems when the number of feasible solutions are small. Most optimization problems in the real world, however, have countless possible solutions. Therefore they can not be applied to such complex problems. As for random searches, while they search in solution spaces without any kind of information, it may not be efficient. Therefore the search direction should be specified in order to improve their search ability. GAs are one of random searches because they use a random choice as a tool in their searching process. While a random choice performs an important role in GAs, the search in GAs is directed by the environment. That is, they utilize information from the environment in their searching process.

While it is easy to apply GAs to optimization problems, several researchers [22,49,78,82] pointed out that the performance of GAs on some optimization problems was a bit inferior to that of neighborhood search algorithms (*e.g.*, local search, simulated annealing [90], and tabu search [108,119]). The performance of GAs should be improved when it is applied to optimization problems. Hybridization of GAs with other search methods is one way to improve

the performance of GAs. When problem-specific information exists, the performance of GAs can be improved by utilizing the information.

In general, optimization problems to be addressed have several objectives to be optimized. As the number of objectives of the problem increases, the complexity of the problem becomes high because the objectives considered are often contradictory to one another. The researcher who tackles an optimization problem with multiple objectives needs a tool for optimizing their problem. Because objectives to be optimized in the problem are often contradictory to one another, the optimal solution of the problem is not obtained as a single solution. That is, a set of candidate solutions called non-dominated solutions is to be obtained for the problem. Since multiple solutions are to be obtained as candidate solutions of the problem, GAs as a kind of multi-point search potentially have an advantage for optimization problems with multiple objectives. However, GAs have been mainly applied to optimization problems with a single objective. Since Schaffer's work [98], extensions of GAs to multi-objective optimization were proposed in several manners (*e.g.*, see Fonseca & Fleming [14,15], Horn *et al.*[30], Kita *et al.*[60], Kursawe [63], Murata & Ishibuchi [77], and Tamaki *et al.* [111,112]).

In this chapter, we explain the basic scheme of GAs for optimization problems. First, a simple genetic algorithm for single-objective optimization is considered. Next, genetic operators for multi-objective optimization are introduced in order to design a multi-objective genetic algorithm. By using a simple test problem, we compare the multi-objective genetic algorithm (MOGA) with several genetic algorithms for multi-objective optimization. In general, when an algorithm is applied to multi-objective optimization problems, it is important whether the algorithm works well for problems with non-convex feasible regions in objective spaces or not. By using another test problem with a non-convex feasible region, we demonstrate that the MOGA can find non-dominated solutions of such problems.

2.2 GENETIC ALGORITHM FOR SINGLE-OBJECTIVE OPTIMIZATION PROBLEMS

In this section, first we consider a simple genetic algorithm. In order to apply GAs to an optimization problem, each solution of the problem to be searched by GAs should be encoded as a finite-length string over some finite alphabet. We briefly describe the difference between the permutation coding and the binary coding. Next, genetic operators such as selection, crossover, mutation, and elitist strategy are described to construct GAs for optimization problems. These genetic operators should be carefully designed according to the property of the problems. The genetic operators for permutation strings are different from those for binary strings. Before applying GAs to optimization problems, several parameters such as population size, crossover probability, and mutation probability should be specified. After all the genetic operators and the parameters are specified for constructing GAs, we can apply GAs to the optimization problem.

2.2.1 Coding

In GAs, each solution of an optimization problem should be encoded as a finite-length string over some alphabet. The coding techniques can be categorized into the following two methods: a binary coding and a permutation coding. The binary representation is usually used for the coding of solutions. For example, the binary coding is often used for function optimization problems. In such problems, an input parameter vector $\boldsymbol{x}$ on a constraint interval vector [$\boldsymbol{a}$, $\boldsymbol{b}$] is encoded by the binary representation. The parameter vector $\hat{\boldsymbol{x}}$ which optimizes a given function $f(\boldsymbol{x})$ is searched by GAs in the binary coding space. The other coding scheme, the permutation coding, is used for sequencing problems such as scheduling problems and traveling salesman problems. For those problems, permutation strings of a set of numbers are more natural representation than binary strings. Fig. 2.1 shows examples of strings by the binary coding and by the permutation coding. The string by the binary coding consists of "0" and "1". The binary string treated in GAs is often decoded to the parameter value in integer, real number, and so on. The permutation string consists of numerals "1" to "n" where each numeral corresponds to a job in scheduling problems or to a city in traveling salesman problems, and n is the total number of jobs or cities. Then jobs are processed according to their order in the permutation when scheduling problems are considered, or cities are visited according to their

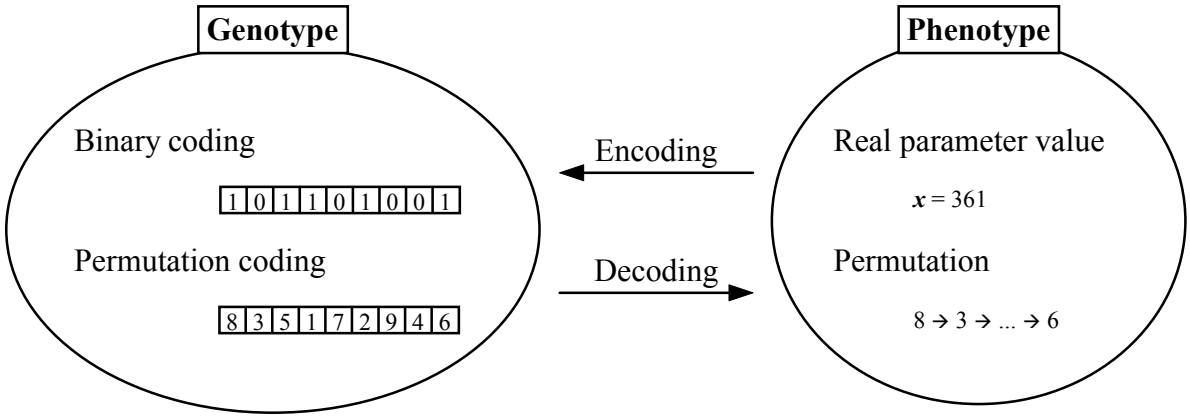

Fig. 2.1 Genotype and phenotype.

order in the permutation in traveling salesman problems.

As shown in Fig. 2.1, strings which consist of binary or numeral elements are called genotype, and solutions which are decoded from strings are called phenotype. GAs search over the genotype world, and strings which are obtained by GAs are decoded into solutions in the phenotype world. That is, the users of GAs can get solutions of their optimization problems after the strings obtained by GAs are decoded into the solutions in the phenotype world.

2.2.2 Evaluation

Each of solutions which are decoded from the strings obtained by GAs is evaluated for optimization problems. GAs search a string with a better fitness value in the genotype world. In the case of function optimization problems, the function value $f(\boldsymbol{x})$ is calculated using a solution $\boldsymbol{x}$ decoded from the corresponding binary string obtained by GAs. When the function value $f(\boldsymbol{x})$ is better, the string in the genotype world which corresponds to the solution $\boldsymbol{x}$ gets a better fitness value. Then the function value of the solution $\boldsymbol{x}$ is transformed to the fitness value in the genotype world. In the genotype world, it is easy for a string with a high fitness value to survive. For function optimization problems, if the function is to be maximized, the function value itself can be used directly for the fitness value. Otherwise, if the function is to be minimized, the fitness function should be defined as an increasing function by transforming the function in the phenotype world. For permutation problems, the same thing can be said. Scheduling problems have many evaluation functions such as the makespan, the total flowtime,

the tardiness penalty, and so on. Traveling salesman problems also have evaluation functions such as the total travel distance. Because permutations found by GAs are evaluated by the evaluation functions in the permutation problems, the function values can be transformed to the fitness values in the same way of function optimization problems. In this way, a fitness value is assigned to each string in the genotype world.

2.2.3 Selection

Selection is an operator to select two parent strings for generating new strings (*i.e.*, offspring). In the selection, a string with a high fitness value has more chance to be selected as one of parents than a string with a low fitness value. In GAs, parent strings are selected by random choice. The parent strings, however, are not selected by a sheer random choice. The fitness value of each string is utilized for selecting parent strings. We describe two ways of selection schemes which are often employed: the roulette wheel selection scheme and the rank-based selection scheme.

One way is the roulette wheel selection. The roulette wheel selection scheme is often used as a selection operator. Let N_{pop} be the number of strings in each population in GAs, that is, N_{pop} is the population size. We denote N_{pop} strings in the current generation by $\Psi = \{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_{N_{\mathrm{pop}}}\}$. Each solution $\boldsymbol{x}_i$ is selected as a parent string according to the selection probability $P_{\mathrm{s}}(\boldsymbol{x}_i)$. In the roulette wheel selection scheme, the selection probability $P_{\mathrm{s}}(\boldsymbol{x}_i)$ is defined as follows:

$$P_{\mathrm{s}}(\boldsymbol{x}_i) = \frac{f(\boldsymbol{x}_i)}{\sum_{j=1}^{N_{\mathrm{pop}}} f(\boldsymbol{x}_j)}, \quad \text{for} \quad i = 1, 2, ..., N_{\mathrm{pop}}, \tag{2.1}$$

where $f(\cdot)$ is the fitness value of the solution $\boldsymbol{x}$.

The other way is the rank-based selection scheme. In this selection scheme, the population is sorted according to the fitness value. Then, the string which has the best fitness value in Ψ is ranked as the first string, and the string which has the second best fitness value is ranked as the second string, and so on. The number of selected strings is decided according to their own rank. For example, Table 2.1 shows that the number of strings selected as parent strings is defined as a percentage of N_{pop}.

Table 2.1 The number of selected strings in the rank-based selection scheme.

Rank	1	2	...	$N_{\text{pop}} - 1$	N_{pop}
% of N_{pop}	20 %	10 %	...	0.1 %	0 %

2.2.4 Crossover

Crossover is an operator to generate new strings (*i.e.*, offspring) from parent strings. Various crossover operators have been proposed for GAs. The crossover operators for permutation strings are different from those for binary strings because permutation problems usually have a requirement that each element of a string should appear only once in the string. We first describe the standard one-point crossover for the binary coding as an example of crossover operators. Next we explain a one-point order crossover as a similar kind of crossover operator which fulfills the requirement. That is, all the elements appear once in a newly generated string by the crossover operator for permutation problems.

A. Crossover for binary strings

The standard one-point crossover is typical operator for binary strings. The operator is applied to selected parent strings as follows: a crossover point is randomly selected between two adjacent elements. Two new strings are generated by swapping all elements in the head part of the strings. Fig. 2.2 shows an example of the one-point crossover operator. In this figure, a crossover point is selected between the sixth position and the seventh position in the strings. All the elements from the first position to the sixth position are swapped. In this way, two new offspring are generated.

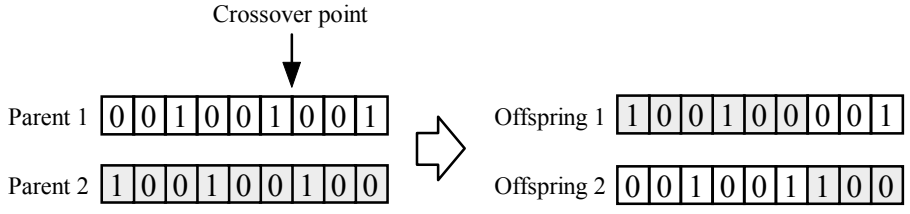

Fig. 2.2 The standard one-point crossover for binary strings.

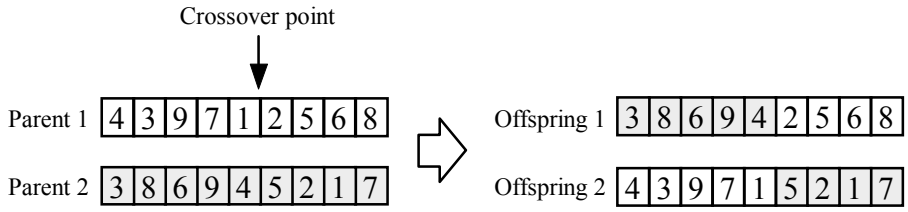

Fig. 2.3 The standard one-point crossover applied to permutation strings.

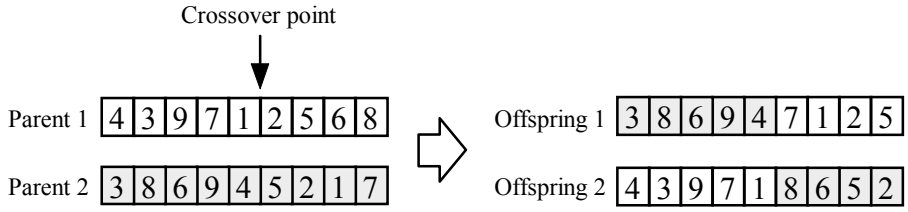

Fig. 2.4 The one-point order crossover for permutation strings.

B. Crossover for permutation strings

If the one-point crossover for binary strings is used for permutation strings, the operator generates offspring which do not fulfill the requirement that each string has to be a permutation of all elements such as jobs or cities. Fig. 2.3 shows offspring generated by the standard one-point crossover. Neither of offspring in the figure represents a legal permutation. In Offspring 1, the elements "6" and "8" appear twice, but the elements "1" and "7" do not appear. Then Offspring 1 does not fulfill the requirement. Offspring 2 is also an illegal permutation because all the elements do not appear in the string. Therefore we need crossover operators which generate legal offspring for permutation problems. The one-point order crossover is one of those crossover operators. This crossover is illustrated in Fig. 2.4. A crossover point is randomly selected for dividing the parent strings. The set of elements in the head part of the strings are swapped each other. The remaining elements of one string, which are not included in the head part, are reordered in the order of their appearance in the other string. By this operator, both of the generated offspring are legal strings which fulfill the requirement of permutation problems. In Fig. 2.4, the head part of Parent 2 is inherited to Offspring 1, and the elements in the tail part of Parent 2 is inherited to the tail part of Offspring 1 in the order of their appearance in Parent 1. In this way, two new offspring are generated.

In this subsection, we briefly described the difference between crossover operators for binary

strings and those for permutation strings using one-point crossover operators. Various crossover operators have been proposed in [23,79,82,89,101,118]. We will explain some of those crossover operators for binary strings and for permutation strings in the succeeding chapters, Chapter 3 and Chapter 5.

2.2.5 Mutation

Mutation is an operator to change elements in a string which is generated by a crossover operator. Such a mutation operator can be viewed as a transition from a current solution to its neighborhood solution in local search algorithms. When we apply mutation operators to strings in GAs, we should be careful as in applying crossover operators. That is, we should be noted that mutation operators for the binary coding is different from those for the permutation coding. We will briefly explain mutation operators for both the coding methods.

A. Mutation for binary strings

A mutation operator for binary strings is quite simple. An element (*i.e.*, "0" or "1") is changed to the other element by the mutation operator. An example of this mutation is shown in Fig. 2.5 where "0" in the second position is changed to "1", and "1" in the fourth position and "1" in the seventh position are changed to "0".

B. Mutation for permutation strings

A mutation operator for permutation strings should be designed in order to generate new strings which fulfill the requirement of permutation problems. Two elements (*i.e.*, numbers from "1" to "*n*") in the string are randomly selected and are swapped with each other. An example of this mutation is shown in Fig. 2.6 where "8" in the second position and "5" in the seventh position are swapped with each other.

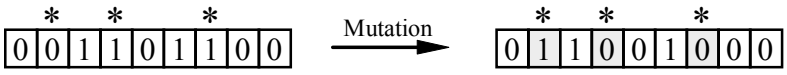

Fig. 2.5 A mutation for binary strings.

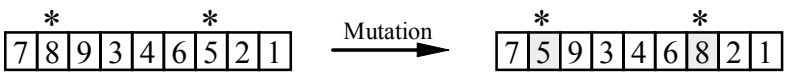

Fig. 2.6 A mutation for permutation strings.

In this subsection, we briefly described the difference between mutation operators for binary strings and those for permutation strings. We will explain some other mutation operators for permutation strings in the succeeding chapter, Chapter 3, which corresponds to optimization problems with a single objective where the permutation coding is used.

2.2.6 Elitist strategy

The elitist strategy is a strategy that the best string in the population should be kept in the next population as it is. That is, the best string is not affected by genetic operators such as crossover and mutation operators. The best string has a lot of chance to be selected as parent strings. It is also kept in the next population as it is in the current population. This strategy is often used for in GAs.

2.2.7 Genetic algorithm

We considered some operations such as coding, evaluation, selection, crossover, mutation, and elitist strategy to construct GAs for optimization problems. We can construct GAs by employing the above operations. The outline of GAs can be written as follows:

Step 0 (Initialization): Randomly generate an initial population of N_{pop} strings where N_{pop} is the population size.

Step 1 (Evaluation): Decode strings to solutions in the phenotype world. Next calculate the value of the objective function for each solution. Then transform the value of the objective function for each solution to the value of the fitness function for each string in the genotype world.

Step 2 (Selection): Select a prespecified number of pairs of strings from the current population according to the selection probability in (2.1).

Step 3 (Crossover): Apply the prespecified crossover operator to each of the selected

pairs in Step 2 to generate N_{pop} strings with the prespecified crossover probability P_c.

Step 4 (Mutation): Apply the prespecified mutation operator to each of the generated strings with the prespecified mutation probability P_m.

Step 5 (Elitist strategy): Randomly remove a string from the current population and add the best string in the previous population to the current one.

Step 6 (Termination test): If a prespecified stopping condition is satisfied, stop this algorithm. Otherwise, return to Step 1.

The outline of GAs is illustrated in Fig. 2.7. For details of genetic operators and parameter specifications in GAs for optimization problems, see the following chapters which concern with flowshop scheduling problems and design of classification systems.

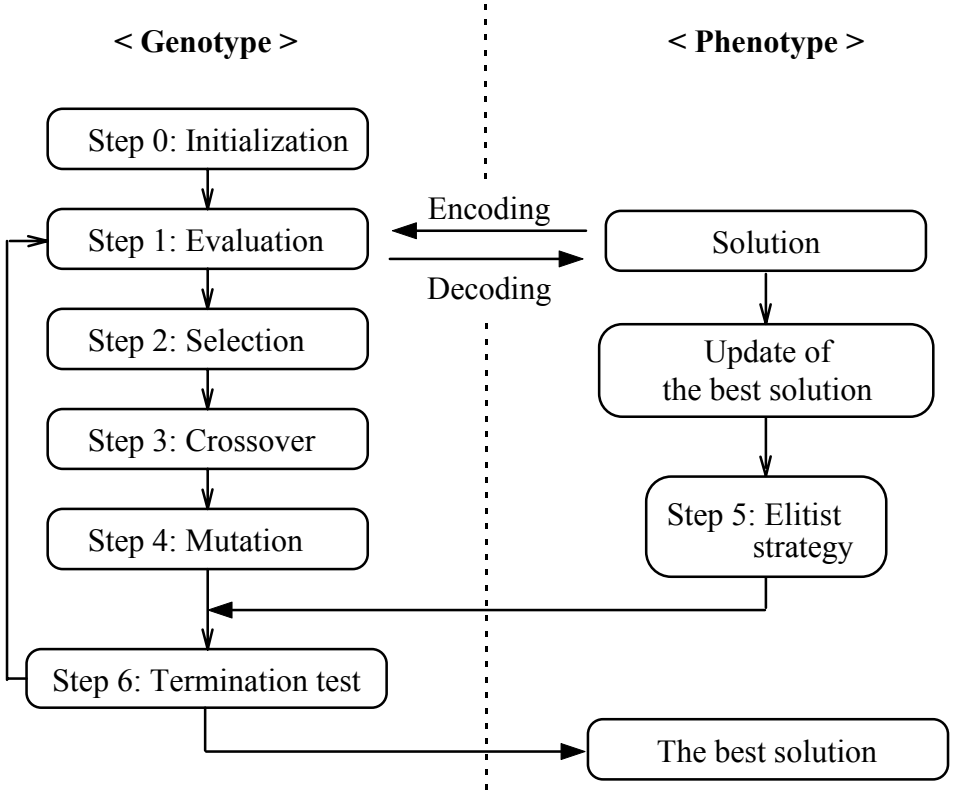

Fig. 2.7 Outline of GAs for optimization problems.

2.3 GENETIC ALGORITHM FOR MULTI-OBJECTIVE OPTIMIZATION PROBLEMS

In this section, we consider an extension of GAs for single-objective optimization problems to the case of multi-objective optimization problems. Since Schaffer's work [98], extensions of GAs to multi-objective optimization were proposed in several manners (*e.g.*, see Fonseca & Fleming [14,15], Horn *et al.*[30], Kita *et al.*[60], Kursawe [63], Murata & Ishibuchi [77], and Tamaki *et al.* [111,112]). The basic scheme of GAs for multi-objective optimization problems are the same as that for single-objective optimization problems. That is, the coding method for multi-objective optimization problems is the same as in Subsection 2.2.1. And the genetic operators such as the crossover and the mutation are the same as in Subsections 2.2.4 and 2.2.5. Therefore, the genetic operators such as evaluation, selection, and elitist strategy should be modified for multi-objective optimization problems. Before we describe modified genetic operators for multi-objective optimization problems in [77], first we explain the background of multi-objective optimization problems, and we describe a multi-objective genetic algorithms.

2.3.1 Background of multi-objective genetic algorithms

Multi-objective genetic algorithms usually try to find all the non-dominated solutions of an optimization problem with multiple objectives. Let us consider the following multi-objective optimization problem with n objectives:

$$\text{Maximize} \quad f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \ ..., f_n(\boldsymbol{x}), \tag{2.2}$$

where $\boldsymbol{x}$ is a vector to be determined, and $f_1(\cdot), f_2(\cdot), \ ..., f_n(\cdot)$ are n objective functions to be maximized. If a feasible solution is not dominated by any other feasible solutions of the multi-objective optimization problem, that solution is said to be a non-dominated solution. When the following inequalities hold between two solutions $\boldsymbol{x}$ and $\boldsymbol{y}$, it is said that the solution $\boldsymbol{x}$ is dominated by the solution $\boldsymbol{y}$:

$$\forall i: \ f_i(\boldsymbol{x}) \le f_i(\boldsymbol{y}) \quad \text{and} \quad \exists j: \ f_j(\boldsymbol{x}) < f_j(\boldsymbol{y}). \tag{2.3}$$

Examples of non-dominated solutions are shown in Fig. 2.8 where dominated solutions and non-dominated solutions are depicted by open circles and closed circles in a two-dimensional objective space, respectively. The two-dimensional objective space in Fig. 2.8 corresponds to the following two-objective optimization problem:

$$\text{Maximize} \quad f_1(\boldsymbol{x}) \quad \text{and} \quad f_2(\boldsymbol{x}). \tag{2.4}$$

As is shown in Fig. 2.8, multi-objective optimization problems usually have several non-dominated solutions.

The aim of our multi-objective algorithms is not to determine a single final solution but to find all the non-dominated solutions of the multi-objective optimization problem in (2.2). Since it is difficult to choose a single solution for a multi-objective optimization problem without iterative interaction with the decision maker, one general approach is to show the set of non-dominated solutions to the decision maker. Then one of the non-dominated solutions can be chosen depending on the preference of the decision maker. Since Schaffer's work [98], extensions of GAs to multi-objective optimization problems have been proposed in several manners. Fonseca & Fleming [15] have published an excellent survey on GAs for multi-objective optimization. Almost all approaches which have already been proposed can be categorized into one of two classes: a "population-based non Pareto approach" or a "Pareto-based approach" by their selection schemes [15].

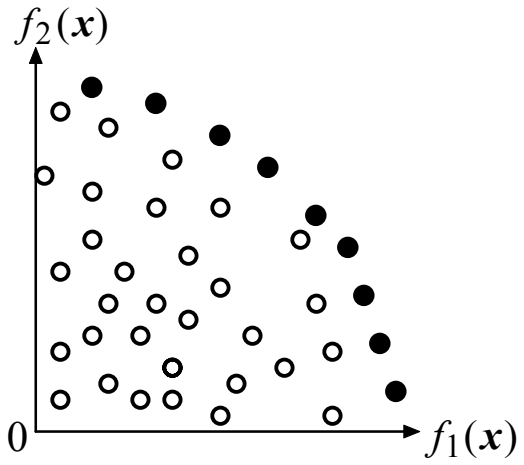

Fig. 2.8 Non-dominated solutions (closed circles) and dominated solutions (open circles).

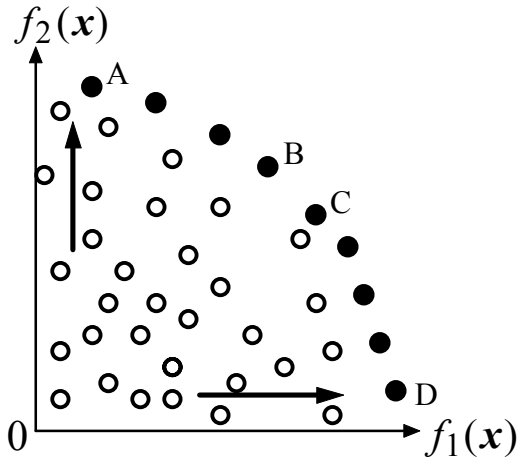

Fig. 2.9 The search directions in Schaffer's approach and Kursawe's approach.

The vector evaluated genetic algorithm (VEGA) proposed by Schaffer [98], which is a pioneer work in this field, is one of "population-based approaches" because its selection to form n subpopulations is implemented according to one of the n objectives separately. Kursawe [63] has also proposed a "population-based approach" where he suggested an idea to choose one of the n objectives according to the user-definable probability assigned to each objective. Thus GAs have n search directions in Schaffer [98] and Kursawe [63]. We show the search directions of these approaches in Fig. 2.9 for the case of the two-objective optimization problem in (2.4). As we can expect from Fig. 2.9, these approaches can easily find the solutions A and D, but it is not easy to find the solutions B and C.

In order to find all the non-dominated solutions by GAs, the variety of individuals (*i.e.*, solutions) should be kept in each generation. Recently, Horn *et al.*[30] proposed the niched Pareto genetic algorithm (NPGA) which is one of "Pareto-based approaches". In their selection scheme, two candidates for selection were picked randomly from the current population. A comparison set which consists of a predefined number of individuals were also selected from the current population. Then each of the candidates is compared against each individual in the comparison set using the inequalities in (2.3) (when all objectives are to be maximized). If one candidate is dominated by the comparison set but the other is not dominated, the latter is selected for the crossover operator. If neither or both are dominated by the comparison set, a fitness sharing technique is adopted (for details, see [30]).

2.3.2 Evaluation

When we apply GAs to the *n*-objective optimization problem, we have to evaluate the values of *n* objective functions for each solution. Using these values of *n* objective functions, the fitness value of each string should be defined. GAs search a string with a better fitness value in the genotype world as in single-objective optimization problems. A way to transform the values of objective functions to the fitness value of each string in the genotype world is to combine the *n* objective functions into a scalar function as follows:

$$f(\boldsymbol{x}) = w_1 f_1(\boldsymbol{x}) + w_2 f_2(\boldsymbol{x}) + \ \ldots \ + w_n f_n(\boldsymbol{x}), \tag{2.5}$$

where $f(\boldsymbol{x})$ is the fitness function of $\boldsymbol{x}$, and $w_1, \ldots, w_n$ are non-negative weights for the *n* objectives. These weights satisfy the following relations:

$$w_i \geq 0 \quad \text{for} \quad i = 1,2,\ldots,n, \tag{2.6}$$

$$w_1 + w_2 + \ \ldots \ + w_n = 1. \tag{2.7}$$

If we use constant weight values for the two-objective optimization problem in (2.4), the search direction by GAs is fixed as shown in Fig. 2.10. The search direction in Fig. 2.10 corresponds to the weight vector $\boldsymbol{w} = (w_1, w_2) = (0.5,\ 0.5)$ in the two-dimensional objective space. When the search direction is fixed, it is not easy to obtain a variety of non-dominated solutions. In the case of Fig. 2.10, GAs with the constant weight vector $\boldsymbol{w} = (w_1, w_2) = (0.5,\ 0.5)$ may easily find the solutions B and C, but it is very difficult to find the solutions A and D.

From the above discussions, we can see that neither the constant weight value approach nor the choice of one objective is appropriate for finding all the non-dominated solutions of the multi-objective optimization problem in (2.2). This is because various search directions are required to find a variety of non-dominated solutions. In order to realize various search directions, we suggest an idea of randomly specified weight values. The weight values are determined as follows:

$$w_i = random_i \,/\, (random_1 + \ \ldots \ + random_n), \quad i = 1,2,\ldots,n, \tag{2.8}$$

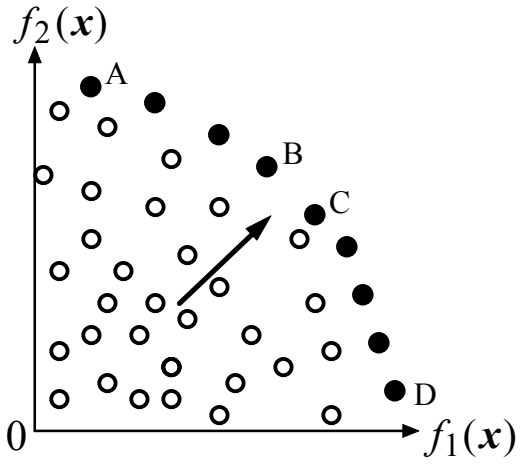

Fig. 2.10 The search direction determined by the constant weight vector $(w_1, w_2) = (0.5, 0.5)$.

where $random_1$, $random_2$, ..., $random_n$ are non-negative random real numbers (or non-negative random integers). The fitness function with various weights is utilized in the selection operator. For details, we will explain in the next subsection.

2.3.3 Selection

When a pair of parent strings are to be selected from a current population Ψ for generating an offspring by a crossover operator, first n weight values $(w_1, w_2, ..., w_n)$ are randomly specified by (2.8). Then the fitness value of each solution $\boldsymbol{x}$ in the current population Ψ is calculated as the weighted sum of the n objectives by (2.5). The selection probability $P_s(\boldsymbol{x}_i)$ of each string $\boldsymbol{x}$ based on the linear scaling is defined by the roulette wheel selection as follows:

$$P_s(\boldsymbol{x}_i) = \frac{f(\boldsymbol{x}_i) - f_{\min}(\Psi)}{\sum_{j=1}^{N_{pop}} \{f(\boldsymbol{x}_j) - f_{\min}(\Psi)\}}, \quad \text{for } i = 1, 2, ..., N_{pop}, \tag{2.9}$$

where $f_{\min}(\Psi)$ is the minimum fitness value (*i.e.*, the worst fitness value) in the current population Ψ. According to this selection probability, a pair of parent strings are selected from the current population Ψ.

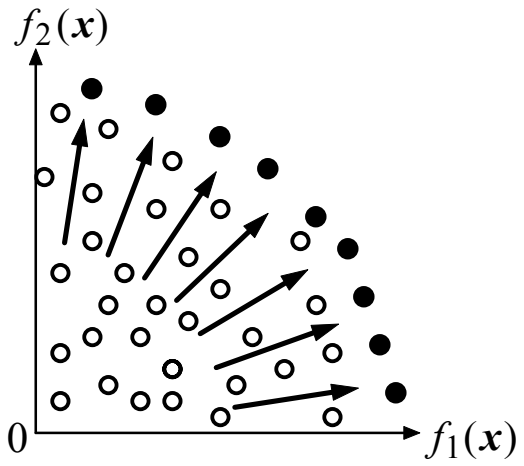

Fig. 2.11 Various search directions of the multi-objective genetic algorithm.

An offspring (*i.e.*, a new string) is generated by a crossover operator from the selected pair of parent strings. Then a mutation operator is applied to the new string. When another pair of parent strings are selected, we randomly specify the n weight values $(w_1, w_2, \ldots, w_n)$ again. That is, we use a different weight vector for each selection. Thus the selection in our multi-objective genetic algorithm has various search directions as shown in Fig. 2.11.

2.3.4 Elitist strategy

During execution of GAs for multi-objective optimization, two sets of solutions are stored: a current population and a tentative set of non-dominated solutions. After evaluating all the strings in the current population, the tentative set of non-dominated solutions is updated by the current population. That is, if a string in the current population is not dominated by any other strings in the current population and the tentative set of non-dominated solutions, this string is added to the tentative set. Then all solutions dominated by the added one are eliminated from the tentative set. In this manner, the tentative set of non-dominated solutions is updated at every generation in GAs for multi-objective optimization problems. From the tentative set of non-dominated solutions, a few solutions are randomly selected and added to the current population. The randomly selected non-dominated solutions may be viewed as a kind of elite solutions because they are added to the current population with no genetic operations. Update of the current population and the tentative set of non-dominated solutions is illustrated in Fig. 2.12.

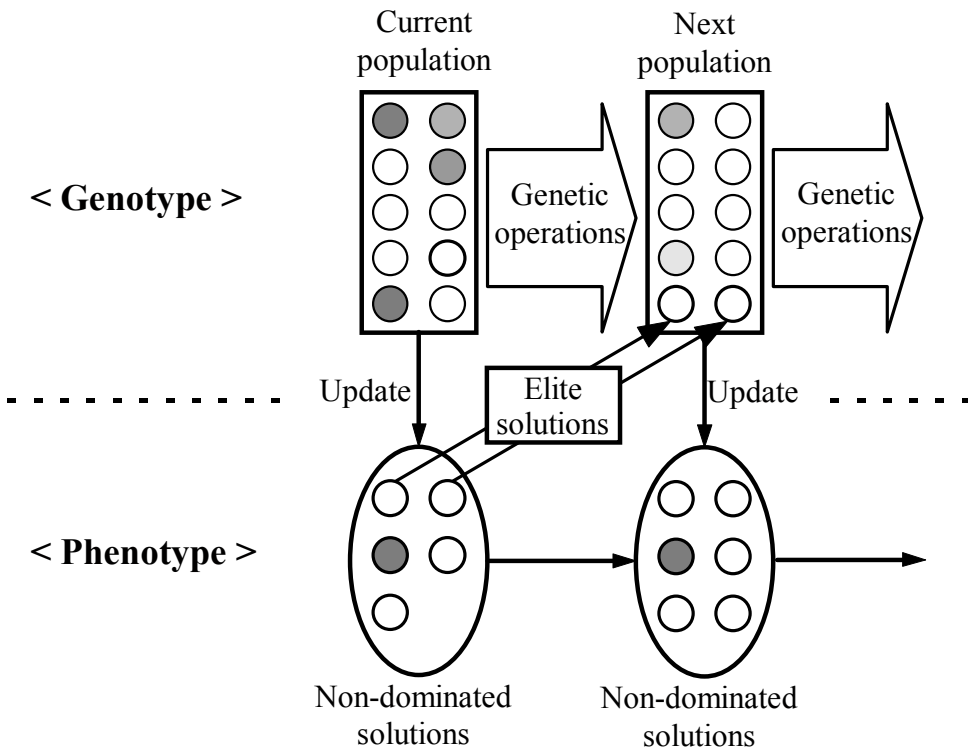

Fig. 2.12 Update of the two sets of strings stored in the MOGA.

2.3.5 Multi-objective genetic algorithm

We considered some modified operations such as evaluation, selection, and elitist strategy in the previous sections in order to construct a genetic algorithm for multi-objective optimization problems. We can construct a multi-objective genetic algorithm (MOGA) by employing those operations for multi-objective optimization. The outline of the MOGA can be written as follows:

Step 0 (Initialization): Randomly generate an initial population of N_{pop} strings where N_{pop} is the population size.

Step 1 (Evaluation): Decode strings to solutions in the phenotype world. Next calculate the values of the n objectives for each solution. Then update the tentative set of non-dominated solutions.

Step 2 (Selection): Repeat the following procedure to select parent strings to generate N_{pop} strings.

(i) Randomly specify the weight values $w_1, w_2, ..., w_n$ in the fitness function (2.5) by (2.8).

(ii) According to the selection probability in (2.9), select a pair of parent strings.

Step 3 (Crossover): Apply a crossover operator to each of the selected pairs in Step 2.

Step 4 (Mutation): Apply a mutation operator to each of the generated strings with mutation probability P_m.

Step 5 (Elitist strategy): Randomly remove N_{elite} solutions from the generated N_{pop} solutions, and add N_{elite} solutions that are randomly selected from the tentative set of non-dominated solutions.

Step 6 (Termination test): If a prespecified stopping condition is satisfied, stop this algorithm. Otherwise, return to Step 1.

The outline of the MOGA is illustrated in Fig. 2.13. For details of genetic operators and

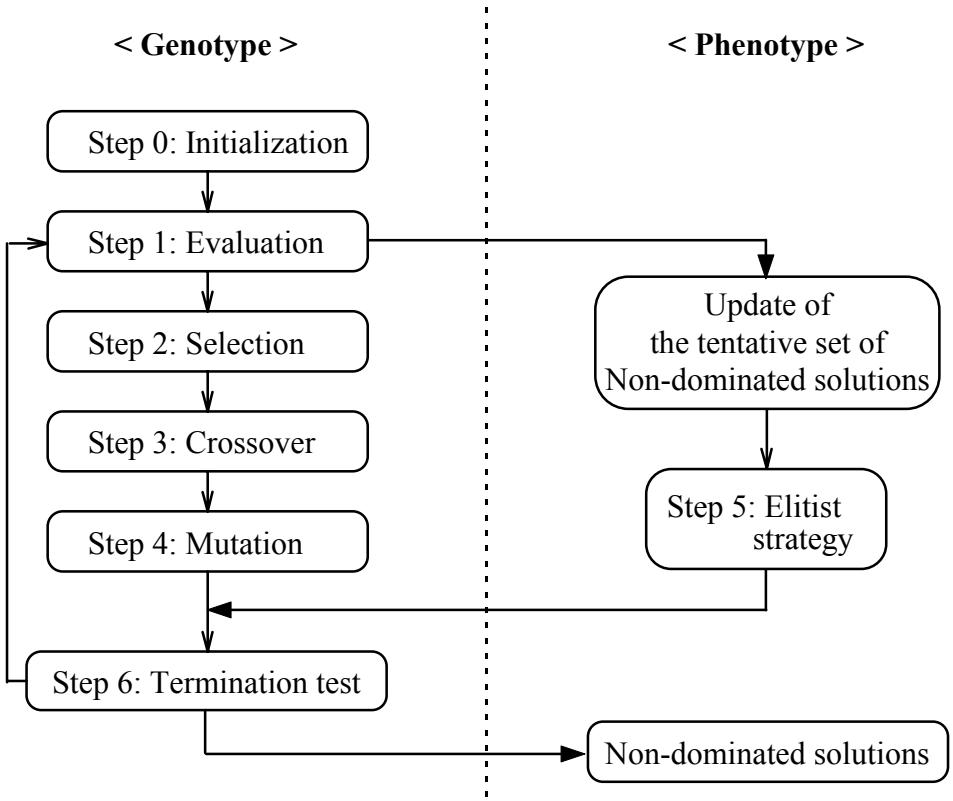

Fig. 2.13 Outline of the MOGA.

parameter specifications in GAs for multi-objective optimization problems, see the following chapters which are concerned with flowshop scheduling problems and rule selection problems. In the next subsection, we apply our multi-objective genetic algorithm to a simple test problem with two objectives and a function optimization problem with a non-convex feasible region in the objective space.

2.3.6 Computer simulations and discussions

In this section, we first compare our multi-objective genetic algorithm (MOGA) described in the previous subsections with the vector evaluated genetic algorithm (VEGA) by Schaffer [98] and the niched Pareto genetic algorithm (NPGA) by Horn *et al.*[30] on a simple test problem used in [30]. Then we apply the MOGA to a function optimization problem used in [112] which has a non-convex feasible region in the objective space. By these computer simulations, high search ability of the MOGA is demonstrated.

A. Application to a simple test problem

First we show simulation results by three genetic algorithms (VEGA, NPGA, and MOGA) for a simple test problem [30] called "unitation versus pairs". This problem has two objectives to be maximized: unitation and complementary adjacent pairs. Unitation $Unit(\boldsymbol{x})$ is simply the number of 1's in the fixed length bit string $\boldsymbol{x}$ (*e.g.*, $Unit(11010100) = 4$). Pairs $Prs(\boldsymbol{x})$ is the number of pairs of complementary adjacent bits (*i.e.*, 01 or 10). For example, $Prs(11010100) = 5$.

When our MOGA was applied to this problem, we used the following fitness function:

$$f(\boldsymbol{x}) = w_{\text{Unit}} \cdot Unit(\boldsymbol{x}) + w_{\text{Prs}} \cdot Prs(\boldsymbol{x}), \tag{2.10}$$

where w_{Unit} and w_{Prs} are randomly specified non-negative weights for $Unit(\cdot)$ and $Prs(\cdot)$, respectively.

True non-dominated solutions (*i.e.*, Pareto optimal solutions) of this problem with 12-bit binary strings are shown in Fig. 2.14 [30] where "P" indicates a Pareto optimal solution and "-" denotes a feasible solution dominated by the Pareto optimal solutions.

In all the three algorithms, the standard one-point crossover described in Subsection 2.2.4

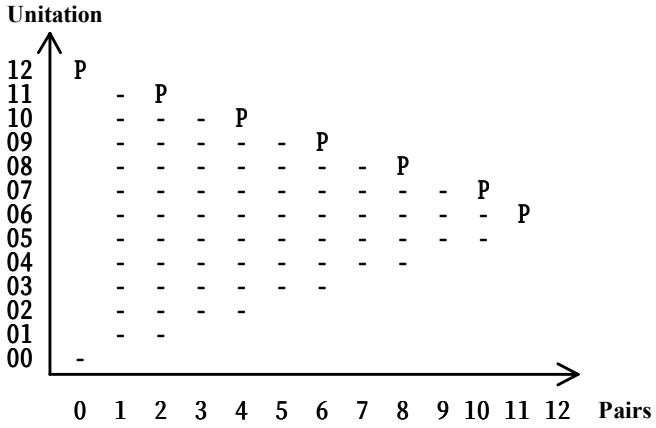

Fig. 2.14 Feasible (-) and Pareto (P) solutions of the simple test problem [30].

was employed with the crossover probability 0.9. The number of strings (*i.e.*, individuals) in each population (*i.e.*, population size) was specified as $N_{\text{pop}} = 100$. Fig. 2.15 ～ Fig. 2.17 show the final solutions by the three algorithms after 100 generations (*i.e.*, solutions in the 100-th generation). In Fig. 2.15 ～ Fig. 2.17, each numeral plotted in the two-dimensional objective space shows the number of obtained solutions (*i.e.*, individuals) at the corresponding coordinate. Simulation results in Fig. 2.15 and Fig. 2.17 were obtained by our simulation, and Fig. 2.16 was quoted from Horn *et al.* [30]. Pareto optimal solutions in these figures are indicated by encircling the corresponding numerals (*e.g.*, ②).

From these figures, we can observe the characteristic of each approach. Many individuals in the VEGA were driven to the two extreme solutions, *i.e.*, there were 38 and 21 individuals at $(\text{Pairs}, \text{Unitation}) = (0,\ 12)$ and (11, 6), respectively (see, Fig. 2.15). Two Pareto optimal solutions $(\text{Pairs}, \text{Unitation}) = (4,\ 10)$ and (6, 9) were not included in the final generation obtained by the VEGA (see, also Fig. 2.15). The NPGA succeeded in maintaining equal size subpopulations, but one Pareto optimal solution $(\text{Pairs}, \text{Unitation}) = (11,\ 6)$ was not included in the final generation (see, Fig. 2.16). Our MOGA could find all the Pareto optimal solutions (see, Fig. 2.17).

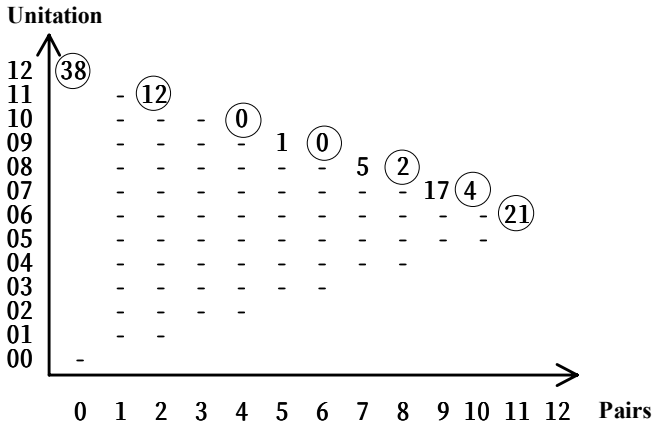

Fig. 2.15 The number of Pareto optimal solutions obtained by the VEGA.

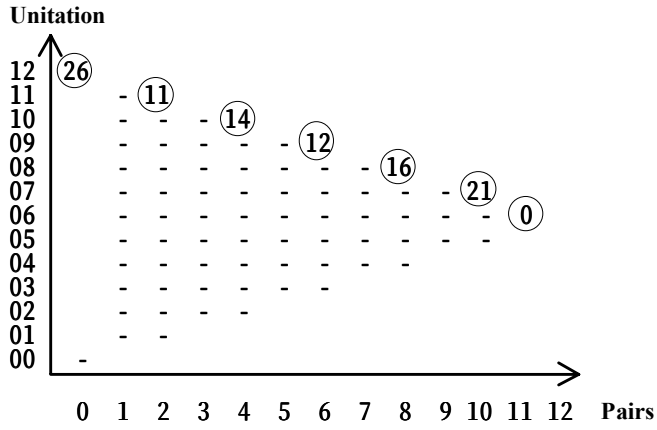

Fig. 2.16 The number of Pareto optimal solutions obtained by the NPGA [30].

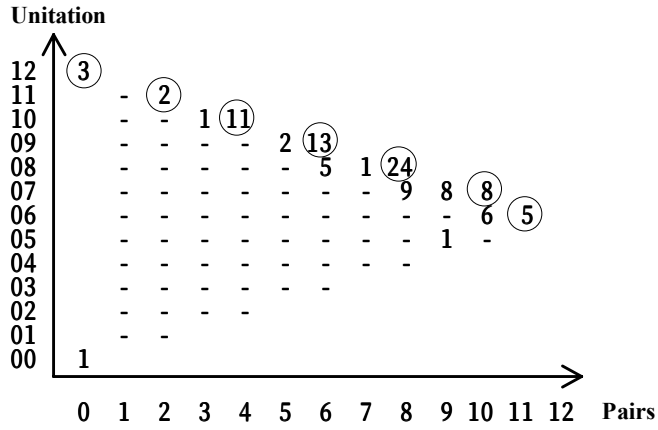

Fig. 2.17 The number of Pareto optimal solutions obtained by the MOGA.

B. Application to a test problem with a non-convex feasible region

Weighted sum approaches usually do not work well for multi-objective optimization problems with non-convex feasible regions in objective spaces. In this subsection, we demonstrate that our MOGA can handle such multi-objective optimization problems. As a test problem, let us consider the following two-objective optimization problem.

$$\text{Minimize} \quad f_1(\boldsymbol{x}) = 2\sqrt{x_1} \quad \text{and} \quad f_2(\boldsymbol{x}) = x_1(1 - x_2) + 5, \tag{2.11}$$

$$\text{subject to} \quad 1 \le x_1 \le 4 \quad \text{and} \quad 1 \le x_2 \le 2, \tag{2.12}$$

where $\boldsymbol{x} = (x_1, x_2)$. This test problem was used for examining some multi-objective GAs in Tamaki *et al.*[112]. Substituting (2.11) into (2.12), we obtained the relation between $f_1(\boldsymbol{x})$ and $f_2(\boldsymbol{x})$ as follows:

$$f_2(\boldsymbol{x}) = \frac{1 - x_2}{4} \cdot \{f_1(\boldsymbol{x})\}^2 + 5. \tag{2.13}$$

When $x_2 = 2$, (2.13) gives the non-dominated solutions of this problem. We show the feasible region of this test problem and the non-dominated solutions in the two-dimensional objective space in Fig. 2.18. From Fig. 2.18, we can see that this test problem has the non-convex feasible region.

Because the two objectives of this test problem should be minimized, we specified the fitness function of each string $\boldsymbol{x}$ as follows:

$$f(\boldsymbol{x}) = -w_1 f_1(\boldsymbol{x}) - w_2 f_2(\boldsymbol{x}). \tag{2.14}$$

This fitness function was used in the selection of our MOGA. In the same manner as in Tamaki *et al.*[112], we denoted each decision variable x_i by a 10-bit string. For example,

$$x_1 = 1010101000 \quad \text{and} \quad x_2 = 1111010100. \tag{2.15}$$

Thus each solution $\boldsymbol{x} = (x_1, x_2)$ of the test problem was denoted by a 20-bit string in the genotype world. For example,

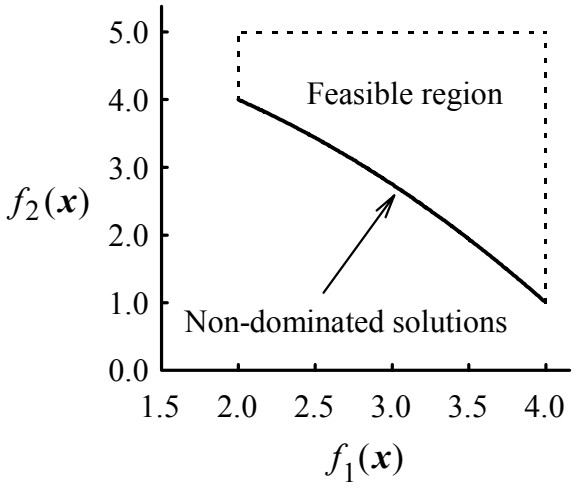

Fig. 2.18 Feasible region and non-dominated solutions of a test problem.

$$\boldsymbol{x} = 10101010001111010100. \tag{2.16}$$

For 20-bit strings, we used a two-point crossover and the standard binary mutation (*i.e.*, $0 \to 1$ and $1 \to 0$) in the same manner as in Tamaki *et al*.[112]. We applied our MOGA to the test problem with the following parameter specifications:

Population size: $N_{\text{pop}} = 100$, Crossover probability: 0.9,

Mutation probability for each bit: 0.01, The number of elite solutions: $N_{\text{elite}} = 5$,

Stopping condition: 20 generations.

These parameter specifications are similar to those in Tamaki *et al*.[112].

We show the final set of non-dominated solutions in Fig. 2.19. From Fig. 2.19, we can see that non-dominated solutions on the non-convex surface of the feasible region were obtained by the MOGA.

We also applied single objective genetic algorithms where weight w_1 and w_2 were fixed as follows:

$$(w_1, w_2) = (100, 1), (50, 1), (20, 1), (15, 1), (10, 1), (5, 1), (2, 1), (1, 1), (1, 2), (1, 5),$$
$$(1, 10), (1, 20), (1, 50), (1, 100).$$

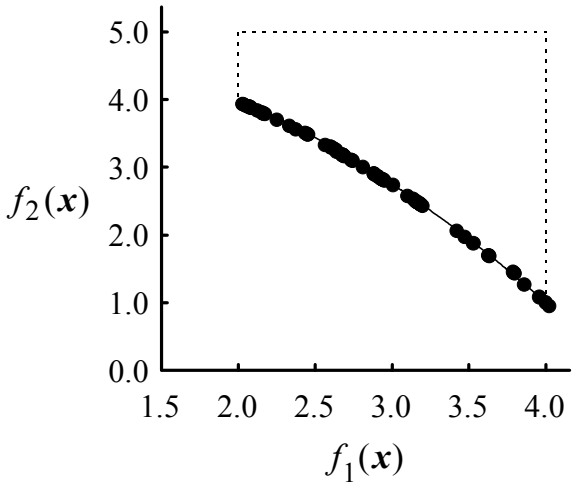

Fig. 2.19 Final set of non-dominated solutions obtained by the MOGA.

The single objective genetic algorithms with these 15 different weight values found only two non-dominated solutions: $(f_1, f_2) = (2, 4)$ and $(f_1, f_2) = (4, 1)$. From these simulation results, we can see that the single objective genetic algorithms with constant weights can not find the non-dominated solutions on the non-convex surface of the feasible region.

2.4 SUMMARY

In this chapter, we explained GAs for optimization problems. Several researchers [22,49,78,82] pointed out that the performance of GAs on some optimization problems was a bit inferior to that of neighborhood search algorithms (*e.g.*, local search, simulated annealing [90], and tabu search [108, 119]). Hybridization of GAs with other search methods is one way to improve the performance of GAs. In the following chapters, we consider hybrid algorithms in order to improve performance of GAs. For flowshop scheduling problems, GAs are hybridized with a local search algorithm and a simulated annealing algorithm. For designing fuzzy classification systems, we incorporate a learning procedure into GAs which improves performance of fuzzy classification systems to be constructed.

First we considered a genetic algorithm for single-objective optimization. We described that GAs should be designed according to the type of optimization problems which they treat. We also considered an extension of GAs for single-objective optimization problems to multi-objective optimization problems. In order to construct GAs for multi-objective optimization, we introduced operations such as evaluation, selection, and elitist strategy for multi-objective optimization problems. We compared the MOGA with the VEGA by Schaffer [98] and the NPGA by Horn *et al*.[30] on a simple test problem used in [30]. By computer simulations, it was demonstrated that the MOGA is comparable to the other algorithms. We applied the MOGA to a function optimization problem with a non-convex feasible region in the objective space used in [112]. By applying the MOGA to such problem, the ability of the MOGA to find set of non-dominated solutions on the non-convex surface of the feasible region was demonstrated.