CHAPTER

17

Nature-Inspired Search Methods

Upon completion of this chapter, you will be able to:

- Explain the basic concepts, terminology, and steps of genetic algorithms (GAs)
- Explain and use the differential evolution algorithm (DEA)
- Explain and use the ant colony optimization (ACO) algorithm
- Explain and use the particle swarm optimization (PSO) algorithm

In this chapter, optimization algorithms inspired by natural phenomena are described. These fall into the general class of *direct search methods* described earlier in chapter: More on Numerical Methods for Unconstrained Optimum Design. However, in contrast to some direct search methods, they do not require the continuity or differentiability of problem functions. The only requirement is to be able to evaluate functions at any point within the allowable ranges for the design variables. Nature-inspired methods use stochastic ideas and random numbers in their calculations to search for the optimum point. Decisions made at most steps of the algorithms are based on random number generation. Therefore, executed at different times, the algorithms can lead to a different sequence of designs and a different solution even with the same initial conditions. They tend to converge to a global minimum point for the function, but there is no guarantee of convergence or global optimality of the final solution.

Nature-inspired approaches have been called stochastic programming, evolutionary algorithms, genetic programming, swarm intelligence, and evolutionary computation. They are also called nature-inspired metaheuristics methods, as they make no assumptions about the optimization problem and can search very large spaces for candidate solutions.

Nature-inspired algorithms can overcome some of the challenges that are due to multiple objectives, mixed design variables, irregular/noisy problem functions, implicit problem functions, expensive and/or unreliable function gradients, and uncertainty in the model and the environment. The methods are very general and can be applied to all kinds of problems—discrete, continuous, and nondifferentiable. They are relatively easy to use and program since

Introduction to Optimum Design. http://dx.doi.org/10.1016/B978-0-12-800806-5.00017-2

they do not require the use of gradients of cost or constraint functions. For this reason, there has been considerable interest in their development and in their application to a wide variety of practical problems. Several books on various methods have been published; a few examples are Goldberg (1989), Gen and Cheng (1997), Corne et al. (1999), Kennedy et al. (2001), Glover and Kochenberger (2002), Coello-Coello et al. (2002), Osyczka (2002), Price et al. (2005), and Qing (2009).

There have also been conferences and workshops on various nature-inspired methods such as the IEEE Congress on Evolutionary Computation, Soft Computing, Genetic and Evolutionary Computation Conference (GECCO), International Conference on Parallel Problem Solving from Nature (PPSN), Ant Colony Optimization and Swarm Intelligence (ANTS), the Evolutionary Programming Conference, and others. Journals devoted to research on nature-inspired methods include: *IEEE Transactions on Evolutionary Computation*, *Applied Intelligence*, *Neural Network World*, *Artificial Intelligence Review*, *Applied Soft Computing*, *Physics of Life Reviews*, *AI Communications*, *Evolutionary Computing*, *Journal of Artificial Intelligence Research*, *Journal of Heuristics*, and *Artificial Life*.

The *drawbacks* of these algorithms are as follows:

1. They require a large amount of function evaluations for even reasonably sized problems. For problems where evaluation of functions itself requires massive calculation, the amount of computing time required to solve the problem can be prohibitive.
2. There is no absolute guarantee that a global solution has been obtained.

The first drawback can be overcome to some extent by the use of massively parallel computers. The second drawback can be overcome to some extent by executing the algorithm several times and allowing it to run longer.

The methods usually start with a collection of design points called the *population*. Using certain stochastic processes, the methods try to come up with a better design point for each *generation* (iteration of the algorithm). To give a flavor of nature-inspired methods, we will describe four methods in this chapter that are relatively popular. (Other methods in this class are noted in Das and Suganthan, 2011.) Each one uses specific terminology from the corresponding biological phenomenon or other natural phenomena that may be unfamiliar to engineers, so we will describe such terminology wherever used.

The methods presented here treat the following optimization problem:

Minimize

$$f(\mathbf{x}) \quad \text{for} \quad \mathrm{x} \in S \tag{17.1}$$

where S is the feasible set of designs and $\mathbf{x}$ is the n-dimensional design variable vector. If the problem is unconstrained, the set S is the entire design space, and if it is constrained, S is determined by the constraints. The methods presented in this chapter are generally used for unconstrained problems. However, constrained optimization problems, can be addressed using the penalty function approach described in chapter: More on Numerical Methods for Unconstrained Optimum Design or the exact penalty function defined in chapter: Numerical Methods for Constrained Optimum Design.

In the following presentation, the terms *design vector*, *design point*, and *design* are used interchangeably. They all refer to the n-dimensional design variable vector $\mathbf{x}$.

17.1 GENETIC ALGORITHMS (GA) FOR OPTIMUM DESIGN

In this section, concepts and terminology associated with GAs are defined and explained for the optimization problem. Fundamentals of GAs are presented and explained. Although the algorithm can be used for continuous problems, our focus will be on discrete variable optimization problems. Various steps of a GA are described that can be implemented in different ways.

Most of the material for this chapter is derived from the work of the author and his coworkers and is introductory in nature (Arora et al., 1994; Huang and Arora, 1997; Huang et al., 1997; Arora, 2002). Numerous other good references on the subject are available (eg, Holland, 1975; Goldberg, 1989; Mitchell, 1996; Gen and Cheng, 1997; Pezeshk and Camp, 2002).

17.1.1 Basic Concepts and Definitions Related to GA

Genetic algorithms loosely parallel *biological evolution* and are based on Darwin's theory of natural selection. The specific mechanics of the algorithm uses the language of microbiology, and its implementation mimics genetic operations. We will explain this in subsequent paragraphs and sections. *The basic idea of the approach is to start with a set of designs*, randomly generated using the allowable values for each design variable. Each design is also assigned a fitness value, usually using the cost function for unconstrained problems or the penalty function for constrained problems. From the current set of designs, a subset is selected randomly with a bias allocated to more fit members of the set. Random processes are used to generate new designs using the selected subset of designs.

The size of the design set is kept fixed. Since more fit members of the set are used to create new designs, the successive sets of designs have a higher probability of having designs with better fitness values. The process is continued until a stopping criterion is met. In the following paragraphs, some details of implementing these basic steps are presented and explained. First we will define and explain various terms associated with the algorithm.

Population: The set of design points at the current iteration is called a population. It represents a group of designs as potential solution points. N_p is the number of designs in a population; this is also called the population size.
Generation: An iteration of the GA is called a generation. A generation has a population of size N_p that is manipulated in a GA.
Chromosome: This term is used to represent a design point. Thus a chromosome represents a design of the system, whether feasible or infeasible. It contains values for all the design variables of the system.
Gene: This term is used for a scalar component of the design vector; that is, it represents the value of a particular design variable.

Design Representation

A method is needed to represent design variable values in their allowable sets and to represent design points so that they can be used and manipulated in the algorithm. This is called a *schema*, and it needs to be encoded (ie, defined). Although binary encoding is the most common approach, real-number coding, and integer encoding are also possible. Binary encoding implies a string of 0s and 1s. Binary strings are also useful because it is easier to explain the operations of the GA with them.

A binary string of 0s and 1s can represent a design variable (a gene). Also, an L-digit string with a 0 or 1 for each digit, where L is the total number of binary digits, can be used to specify a design point (a chromosome). Elements of a binary string are called *bits*; a bit can have a value of 0 or 1. We will use the *term "V–string" for a binary string that represents the value of a variable*; that is, the component of a design vector (a gene). Also, we will use the *term "D–string" for a binary string that represents a design of the system*—that is, a particular combination of n V–strings, where n is the number of design variables. This is also called a *genetic string* (or a chromosome).

An m-digit binary string has 2^m possible 0–1 combinations implying that 2^m discrete values can be represented. The following method can be used to transform a V–string consisting of a combination of m 0s and 1s to its corresponding discrete value of a variable having N_c allowable discrete values: let m be the smallest integer satisfying $2^m > N_c$; calculate the integer j:

$$j = \sum_{i=1}^{m} ICH(i)2^{(i-1)} + 1 \tag{17.2}$$

where $ICH(i)$ is the value of the ith digit (either 0 or 1). Thus the jth allowable discrete value corresponds to this 0–1 combination; that is, the jth discrete value corresponds to this V–string. Note that when $j > N_c$ in Eq. (17.2), the following procedure can be used to adjust j such that $j \le N_c$:

$$j = \text{INT}\left(\frac{N_c}{2^m - N_c}\right)(j - N_c) \tag{17.3}$$

where INT(x) is the integer part of x. As an example, consider a problem with three design variables each having $N_c = 10$ possible discrete values. Thus, we will need a four-digit binary string to represent discrete values for each design variable; that is, $m = 4$ implying that 16 possible discrete values can be represented. Let a design point $\mathbf{x} = (x_1, x_2, x_3)$ be encoded as the following D–string (genetic string):

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ |0110| & |1111| & |1101| \end{bmatrix} \tag{17.4}$$

Using Eq. (17.2), the j values for the three V–strings are calculated as 7, 16, and 12. Since the last two numbers are larger than $N_c = 10$, they are adjusted by using Eq. (17.3) as 6 and 2, respectively. Thus the foregoing D–string (genetic string) represents a design point where the seventh, sixth, and second allowable discrete values are assigned to the design variables x_1, x_2, and x_3, respectively.

Initial Generation/Starting Design Set

With a method to represent a design point defined, the first population consisting of N_p designs needs to be created. This means that N_p D–strings need to be created. In some cases, the designer already knows some good usable designs for the system. These can be used as *seed* *designs* to generate the required number of designs for the population using some random process. Otherwise, the initial population can be generated randomly via the use of a random number generator. Several methods can be used for this purpose. The following procedure shows a way to produce a 32-digit D–string:

1. Generate two random numbers between 0 and 1 as "0.3468 0254 7932 7612 and 0.6757 2163 5862 3845."
2. Create a string by combining the two numbers as "3468 0254 7932 7612 6757 2163 5862 3845."
3. The 32 digits of the above string are converted to 0s and 1s by using a rule in which "0" is used for any value between 0 and 4 and "1" for any value between 5 and 9, as "0011 0010 1100 1100 1111 0010 1110 0101."

Fitness Function

The fitness function defines the relative importance of a design. A higher fitness value implies a better design. The fitness function may be defined in several different ways; it can be defined using the cost function value as follows:

$$F_i = \left(1+\varepsilon\right) f_{\max} - f_i, \tag{17.5}$$

where f_i is the cost function (penalty function value for a constrained problems) for the ith design, $f_{\max}$ is the largest recorded cost (penalty) function value, and ε is a small value (eg, 2×10^{-7}) to prevent numerical difficulties when F_i becomes 0.

17.1.2 Fundamentals of Genetic Algorithms

The basic idea of a GA is to generate a new set of designs (population) from the current set such that the average fitness of the population is improved. The process is continued until a stopping criterion is satisfied or the number of iterations exceeds a specified limit. Three genetic operators are used to accomplish this task: reproduction, crossover, and mutation.

Reproduction is an operator where an old design (D–string) is copied into the new population according to the design's fitness. There are many different strategies to implement this reproduction operator. This is also called the *selection process*.
Crossover corresponds to allowing two selected members of the new population to exchange characteristics of their designs among themselves. Crossover entails selection of starting and ending positions on a pair of randomly selected strings (called *mating strings*), and simply exchanging the string of 0s and 1s between these positions.
Mutation is the third step that safeguards the process from a complete premature loss of valuable genetic material during reproduction and crossover. In terms of a binary string, this step corresponds to selection of a few members of the population, determining a location on the strings at random, and switching 0 to 1 or vice versa.

The foregoing three steps are repeated for successive generations of the population until no further improvement in fitness is attainable. The member in this generation with the highest level of fitness is taken as the optimum design. Some details of the GA implemented by Huang and Arora (1997a) are described in the sequel.

Reproduction Procedure

Reproduction is a process of selecting a set of designs (D–strings) from the current population and carrying them into the next generation. The *selection process* is biased toward more fit members of the current design set (population). Using the fitness value F_i for each design in the set, its probability of selection is calculated as:

$$P_i = \frac{F_i}{Q}; \quad Q = \sum_{j=1}^{N_p} F_j \tag{17.6}$$

It is seen that the members with higher fitness value have larger probability of selection. To explain the process of selection, let us consider a roulette wheel with a handle shown in Fig. 17.1. The wheel has N_p segments to cover the entire population, with the size of the ith segment proportional to the probability P_i. Now a random number w is generated between 0 and 1. The wheel is then rotated clockwise, with the rotation proportional to the random number w. After spinning the wheel, the member pointed to by the arrow at the starting location is selected for inclusion in the next generation. In the example shown in Fig. 17.1, member 2 of the current population is carried into the next generation. Since the segments on the wheel are sized according to the probabilities P_i, the selection process is biased toward the more fit members of the current population.

Note that a member copied to the mating pool remains in the current population for further selection. Thus, the new population may contain identical members and may not contain some of the members found in the current population. This way, the average fitness of the new population is increased.

Crossover

Once a new set of designs is determined, *crossover* is conducted as a means to introduce variation into a population. Crossover is the process of combining or mixing two different

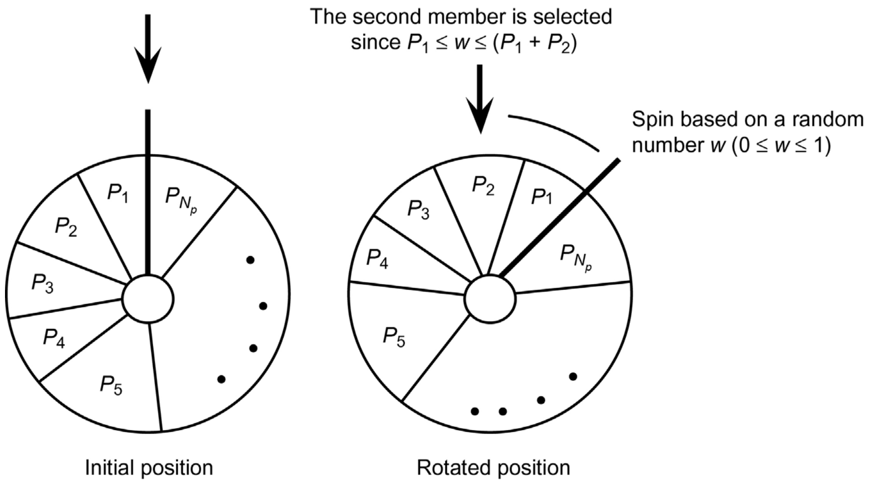
The second member is selected since $P_1 \le w \le (P_1 + P_2)$
Spin based on a random number w $(0 \le w \le 1)$
P_1
P_{N_p}
P_2
P_3
P_4
P_5
P_2
P_1
P_3
P_4
P_{N_p}
P_5
Initial position
Rotated position

FIGURE 17.1 **Roulette wheel process for selection of designs for new generation (reproduction).** *Source: Huang, Hsieh and Arora, 1997.*

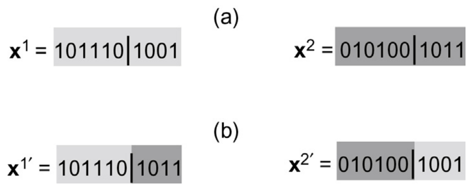

FIGURE 17.2 **Crossover operation with one-cut point.** (a) Designs selected for crossover (parent chromosomes). (b) New designs (children) after crossover.

designs (chromosomes) of the population. Although there are many methods for performing crossover, the most common ones are the *one-cut-point* and *two-cut-point methods*. A cut point is a position on the D–string (genetic string). In the one-cut method a position on the string is randomly selected that marks the point at which two parent designs (chromosomes) split. The resulting four halves are then exchanged to produce new designs (children).

The process is illustrated in Fig. 17.2, where the cut point is determined as four digits from the right end. The lightly shaded four digits 1001 from one parent design are exchanged with heavily shaded four digit 1011 from another parent design. This produces two new designs $\mathbf{x}^{1'}$ and $\mathbf{x}^{2'}$ that replace the old designs (parents). Similarly, the two-cut-point method is illustrated in Fig. 17.3. Selecting how many or what percentage of chromosomes crossover, and at what points the crossover operation occurs, is part of the heuristic nature of GAs. There are many different approaches, and most are based on random selections.

Mutation

Mutation is the next operation on the members of the new design set (population). The idea of mutation is to safeguard the process from a complete premature loss of valuable genetic material during the reproduction and crossover steps. In terms of a genetic string, this step corresponds to selecting a few members of the population, determining a location on each string randomly, and switching 0 to 1 or vice versa. The number of members selected for mutation is based on heuristics, and the selection of location on the string for mutation is based on a random process. Let us select a design as "10 1110 1001" and select location 7 from the right end of its D–string. The mutation operation involves replacing the current value of 1 at the seventh location with 0 as "10 1010 1001."

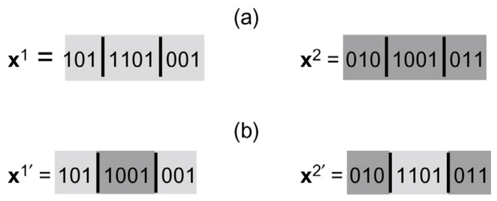

FIGURE 17.3 **Crossover operation with two-cut point.** (a) Designs selected for crossover (parent chromosomes). (b) New designs (children) after crossover.

Number of Crossovers and Mutations

For each generation (iteration), three operators—reproduction or selection, crossover, and mutation—are performed. While the number of the reproduction operations is always equal to the size of the population, the number of crossovers and mutation can be adjusted to fine-tune the performance of the algorithm.

To show the type of operations needed to implement the mutation and crossover at each generation, we present a possible procedure as follows.

1. Set $I_{\max}$ as an integer that controls the amount of crossover. Calculate I_m, which controls the amount of mutation as $I_m = \text{INT}(P_m N_p)$, where P_m represents a fraction of the population that is selected for mutation, and N_p is the size of the population. Too many crossovers can result in a poorer performance of the algorithm since it may produce designs that are far away from the mating designs. Therefore, $I_{\max}$ should be set to a small number. The mutation, however, changes designs in the neighborhood of the current design; therefore, a larger amount of mutation may be allowed. Note also that the population size N_p needs to be set to a reasonable number for each problem. It may be heuristically related to the number of design variables and the number of all possible designs determined by the number of allowable discrete values for each variable.
2. Let f_K^+ denote the best cost (or penalty) function value for the population at the Kth iteration. If the improvement in f_K^+ is less than some small positive number ε' for the last two consecutive iterations, then $I_{\max}$ is doubled temporarily. This "doubling" strategy continues at the subsequent iterations and returns to the original value as soon as f_K^+ is reduced. The concept behind this is that we do not want too much crossover or mutation to ruin the good designs in D–strings as long as they keep producing better offspring. On the other hand, we need more crossover and mutation to trigger changes when progress stops.
3. If improvement in f_K^+ is less than ε' for the last I_g consecutive iterations, P_m is doubled.
4. The crossover and mutation may be performed as follows:

```
FOR $i$ = 1, $I_{\max}$
  Generate a random number $z$ uniformly distributed in [0, 1]
  If $z > 0.5$, perform crossover.
  If $z \leq 0.5$, skip crossover.
  FOR $j$ = 1, $I_m$
  Generate a random number $z$ uniformly distributed in [0, 1]
  If $z > 0.5$, perform mutation.
  If $z \leq 0.5$, skip to next $j$.
  ENDFOR
ENDFOR
```

Leader of the Population

At each generation, the member having the lowest cost function value among all of the designs is defined as the "leader" of the population. If several members have the same lowest cost, only one of them is chosen as the leader. The leader is replaced if another member with lower cost appears. In this way, it is safeguarded from extinction (as a result of reproduction, crossover, or mutation). In addition, the leader is guaranteed a higher probability of selection for reproduction. One benefit of using a leader is that the best-cost (penalty) function value of the population can never increase from one iteration to another, and some of the best design variable values (V–strings or genes) will be able to always survive.

Stopping Criteria

If the improvement for the best-cost (penalty) function value is less than ε' for the last I consecutive iterations, or if the number of iterations exceeds a specified value, then the algorithm terminates.

Genetic Algorithm

Based on the ideas presented here, a sample GA is stated.

Step 1: Define a schema to represent different design points. Randomly generate N_p genetic strings (members of the population) according to the schema, where N_p is the population size. Or use the seed designs to generate the initial population. For *constrained problems*, only the feasible strings are accepted when the penalty function approach is not used. Set the iteration counter $K = 0$. Define a fitness function for the problem, as in Eq. (17.5).
Step 2: Calculate the fitness values for all the designs in the population. Set $K = K + 1$, and the counter for the number of crossovers $I_c = 1$.
Step 3: *Reproduction*. Select designs from the current population according to the roulette wheel selection process described earlier for the mating pool (next generation) from which members for crossover and mutation are selected.
Step 4: *Crossover*. Select two designs from the mating pool. Randomly choose two sites on the genetic strings and swap strings of 0s and 1s between the two chosen sites. Set $I_c = I_c + 1$.
Step 5: *Mutation*. Choose a fraction (P_m) of the members from the mating pool and switch a 0 to 1 or vice versa at a randomly selected site on each chosen string. If, for the past I_g consecutive generations, the member with the lowest cost remains the same, the mutation fraction P_m is doubled. I_g is an integer defined by the user.
Step 6: If the member with the lowest cost remains the same for the past two consecutive generations, then increase $I_{\max}$. If $I_c < I_{\max}$, go to step 4. Otherwise, continue.
Step 7: *Stopping criterion*. If after the mutation fraction P_m is doubled, the best value of the fitness is not updated for the past I_g consecutive generations, then stop. Otherwise, go to step 2.

Immigration

It may be useful to introduce completely new designs into the population in an effort to increase diversity. This is called immigration, which may be done at a few iterations during the solution process when progress toward the solution point is slow.

Multiple Runs for a Problem

It is seen that the GAs make decisions at several places based on random number generation. Therefore, when the same problem is run at different times, it may give different final designs. It is suggested that the problem be run a few times to ensure that the best possible solution has been obtained.

17.1.3 Genetic Algorithm for Sequencing-Type Problems

There are many applications in engineering where the sequence of operations needs to be determined. To introduce the type of problems being treated, let us consider the design of a metal plate that is to have 10 bolts at the locations shown in Fig. 17.4. The bolts are to be inserted into predrilled holes by a computer-controlled robotic arm. The objective is to minimize the movement of the robot arm while it passes over and inserts a bolt into each hole. This class of problems is generally known as *traveling salesman problem*, which is defined as: given a list of N cities and a means to calculate the traveling distance between the two cities, we must plan a salesman's route that passes through each city once (with the option of returning to the starting point) while minimizing the total distance.

For such problems, a feasible design is a string of numbers (a sequence of the cities to be visited) that do not repeat themselves (eg, "1 3 4 2" is feasible and "3 1 3 4" is not). Typical operators used in GAs, such as crossover and mutation, are not applicable to these types of problems since they usually create infeasible designs with repeated numbers. Therefore, other operators need to be used to solve such problems. We will describe some such operators in the following paragraphs.

Permutation type 1: Let n_1 be a fraction for selection of the mating pool members for carrying out Type 1 permutation. Choose Nn_1 members from the mating pool at random, and reverse the sequence between two randomly selected sites on each chosen string. For example, a chosen member with a string of "3$\underline{4}$52$\underline{1}$6" and two randomly selected sites of "4" and "1," is changed to "3$\underline{1}$25$\underline{4}$6."
Permutation type 2: Let n_2 be a fraction for selection of the mating pool members for carrying out the Type 2 permutation. Choose Nn_2 members from the mating pool at random, and exchange the numbers of two randomly selected sites on each chosen string. For example, a chosen member with a string of "3$\underline{4}$52$\underline{1}$6" and two randomly selected sites of "4" and "1," is changed to "3$\underline{1}$52$\underline{4}$6."
Permutation type 3: Let n_3 be a fraction for selection of the mating pool members for carrying out the Type 3 permutation. Choose Nn_3 members from the mating pool at random, and exchange the numbers of one randomly selected site and the site next to it

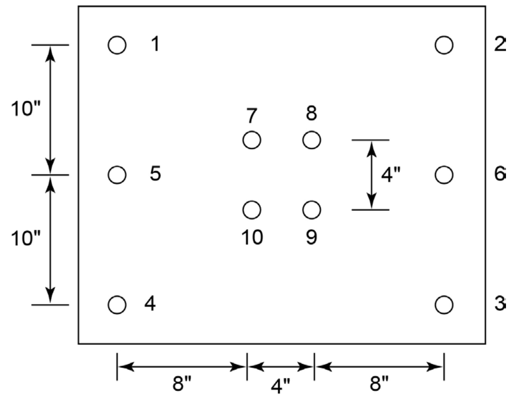

FIGURE 17.4 Bolt insertion sequence determination at 10 locations. *Source: Huang, Hsieh and Arora, 1997.*

on each chosen string. For example, a chosen member with a string of “$3\underline{45}216$” and a randomly selected site of “4”, is changed to “$3\underline{54}216$”.

Relocation

Let n_r be a fraction for selection of the mating pool members for carrying out relocation. Choose Nn_r members from the mating pool at random, remove the number of a randomly selected site, and insert it in front of another randomly selected site on each chosen string. For example, a chosen member with a string of “$3\underline{4}52\underline{1}6$” and two randomly selected sites of “4” and “1”, is changed to “$352\underline{41}6$.”

A computer program based on the previously mentioned operators is developed and used to solve the bolt insertion sequence problem in Example 17.1.

EXAMPLE 17.1 BOLT INSERTION SEQUENCE DETERMINATION AT 10 LOCATIONS

Solve the problem shown in Fig. 17.4 using the GA to minimize the total distance traveled by the robotic arm.

Solution

The problem is solved by using the GA explained in the foregoing (Huang, Hsieh and Arora, 1997). The population size N_p is set to 150, and I_g (the number of consecutive iterations for which the best cost function did not improve by at least ε') is set to 10. No seed designs are used for the problem. The optimum bolting sequence is not unique to the problem. With hole 1 as the starting point, the optimum sequence is determined as (1, 5, 4, 10, 7, 8, 9, 3, 6, 2) and the cost function value is 74.63 in. The number of function evaluations is 1445, which is much smaller than the total number of possibilities ($10! = 3{,}628{,}800$).

Another case solved in Huang, Hsieh and Arora (1997) concerns determining the bolting sequence for 16 locations. The optimum sequence is not unique for this problem either. The solution is obtained in 3358 function evaluations compared with the total number of possibilities, $16! \cong 2.092 \times 10^{13}$.

EXAMPLE 17.2 A-PILLAR SUBASSEMBLY WELDING SEQUENCE

This problem concerns the *A-pillar subassembly welding sequence* determination for a passenger vehicle (Huang, Hsieh and Arora 1997). There are 14 welding locations. The objective is to determine the best welding sequence that minimizes the deformation at some critical points of the structure. Cases where one and two welding guns are used are also considered. This is equivalent to having two salesmen traveling between N cities for the traveling salesman problem. The optimum sequences are obtained with 3341 and 3048 function evaluations for the two cases, which are much smaller than those for the full enumeration.

17.1.4 Applications of GA

Numerous applications of GAs for different classes of problems have been presented in the literature. There are specialty conferences focusing on developments in genetic and other evolutionary algorithms and their applications. The literature in this area is substantial. Therefore, a survey of all the applications is not attempted here. For mechanical and structural design, some of the applications are covered in Arora (2002), Pezeshk and Camp (2002), Arora and Huang (1996), and Chen and Rajan (2000). Applications of the GAs for optimum design of electric transmission line structures are given in Kocer and Arora (1996, 1997, 1999, 2002).

17.2 DIFFERENTIAL EVOLUTION ALGORITHM

The differential evolution algorithm (DEA) works with a population of designs. At each iteration, called a generation, a new design is generated using some current designs and certain random operations. If the new design is better than a preselected parent design, then it replaces that design in the population; otherwise, the old design is retained and the process is repeated. In this section, the steps of a basic DEA are described. The material is derived from the article by Das and Suganthan (2011).

Compared to GAs, DEAs are easier to implement on the computer. Unlike GAs, they do not require binary number coding and encoding, as seen later (although GAs have been implemented with real number coding as well). Therefore, they are quite popular for numerous practical applications. There are four steps in executing the basic DEA:

Step 1: Generation of the initial population of designs.
Step 2: Mutation with difference of vectors to generate a so-called *donor design vector*.
Step 3: Crossover/recombination to generate a so-called *trial design vector*.
Step 4: Selection, that is, acceptance or rejection of the trial design vector using the *fitness function*, which is usually the cost function.

Details of these steps are described in the following subsections. The notation and terminology listed in Table 17.1 are used.

17.2.1 Generation of Initial Population for DEA

A first step in DEA is to generate an initial population of N_p design points; N_p is usually selected as a large number, say, between $5n$ and $10n$. Each design point/vector is also called a *chromosome*. Initial designs can be generated by any procedure that tries to cover the entire design space in a uniformly distributed random manner. If some designs for the system are known, they can be included in the initial population. One way to generate the initial set of designs is to use the lower and upper limits on the design variables and uniformly distributed random numbers. For example, the ith member (design) of the population may be generated as follows:

$$x_j^{(i,0)} = x_{jL} + r_{ij}\left(x_{jU} - x_{jL}\right);\ \ j = 1 \text{ to } n \tag{17.7}$$

where r_{ij} is a uniformly distributed random number between 0 and 1 that is generated for each component of the design point. Each member of the population is a potential solution/optimum point.

TABLE 17.1 Notation and Terminology for the DEA

Notation	Terminology
Cr	Crossover rate; an algorithm parameter
F	Scale factor, usually in the interval [0.4, 1.0]; an algorithm parameter
k	kth generation of the iterative process
$k_{\max}$	Limit on the number of generations
n	Number of design variables
N_p	Number of design points in the population; population size
r_{ij}	Random number uniformly distributed between 0 and 1 for the ith design and its jth component
x_j	jth component of the design variable vector $\mathbf{x}$
$U^{(p,k)}$	Trial design vector at the kth generation/iteration associated with the parent design p
$V^{(p,k)}$	Donor design vector at the kth generation/iteration associated with the parent design p
$\mathbf{x}^{(i,k)}$	ith design point of the population at the kth generation/iteration
$x^{(p,k)}$	Parent design (also called the target design) of the population at the kth generation/iteration
$\mathbf{x}_L$	Vector containing the lower limits on the design variables
$\mathbf{x}_U$	Vector containing the upper limits on the design variables

17.2.2 Generation of a Donor Design for DEA

In this subsection, we describe the idea of a donor design and its generation. A *donor design* is generated using mutation of a selected design with the difference of two other distinct designs in the population. Biologically, mutation means a change in the *gene* (a component of the design vector) characteristics of a *chromosome* (the complete design vector). The donor design point is created by changing a design point of the current population. This change is accomplished by combining the design vector with the difference of two other vectors of the population, all selected randomly. The design vector thus generated is called the donor design/vector. In the context of donor design, then, mutation implies changing all components of a design vector.

To generate the donor design vector, we randomly select three distinct design points from the current population in the generation k: $\mathbf{x}^{(r_1,k)}$, $\mathbf{x}^{(r_2,k)}$, and $\mathbf{x}^{(r_3,k)}$, where the superscripts r_1, r_2, and r_3 refer to three different designs. In addition, we select a fourth point $\mathbf{x}^{(p,k)}$, called the *parent/target* design point; its use in the crossover operation is explained later (the superscript p refers to the parent design). We then form a difference vector using two design points, say r_2 and r_3, as $\left(\mathbf{x}^{(r_2,k)} - \mathbf{x}^{(r_3,k)}\right)$. This difference vector is scaled and added to the third vector to form the donor design vector $\mathbf{V}^{(p,k)}$:

$$\mathbf{V}^{(p,k)} = \mathbf{x}^{(r_1,k)} + F \times \left(\mathbf{x}^{(r_2,k)} - \mathbf{x}^{(r_3,k)}\right) \tag{17.8}$$

where F is a scale factor, typically selected between 0.4 and 1. Note that any procedure can be used to randomly select the foregoing four members of the current population; one example is the roulette wheel procedure described earlier in Section 17.1.2.

17.2.3 Crossover Operation to Generate the Trial Design in DEA

A crossover operation is performed after generating the donor design through mutation. In it, the donor design vector $\mathbf{V}^{(p,k)}$ exchanges some of its components with the parent design vector to form the trial design vector $x_j^{(p,k)}$. The crossover operation is described in the following equation:

$$U_j^{(p,k)} = \begin{cases} V_j^{(p.k)}, & \text{if } r_{pj} \le Cr \quad \text{or} \quad j = j_r \\ x_j^{(p,k)}, & \text{otherwise} \end{cases} \quad ; \quad j = 1 \text{ to } n \tag{17.9}$$

where r_{pj} is a uniformly distributed random number between 0 and 1 and j_r is a randomly generated index between 1 and n that ensures that $\mathbf{U}^{(p,k)}$ receives at least one component from $\mathbf{V}^{(p,k)}$.

The crossover operation in Eq. (17.9) says that when the random number r_{pj} for each component of the design vector does not exceed the Cr value, or if $j = j_r$, set the trial design component $U_j^{(p,k)}$ to the donor design component $V_j^{(p,k)}$; otherwise, replace it with the parent design component $x_j^{(p,k)}$. With this approach, the number of components inherited from the donor design vector has a (nearly) binomial distribution. Therefore, this operation sometimes is called *binomial crossover*.

17.2.4 Acceptance/Rejection of the Trial Design in DEA

The next step of the algorithm is to check if the trial design $\mathbf{U}^{(p,k)}$ is better than the parent design $\mathbf{x}^{(p,k)}$; if it is, it replaces the parent design in the population to keep the population size constant (as a variation, both vectors may be retained sometimes increasing the size of the population by one every time). Usually called the *selection* step, this is described in the following equation:

$$\mathbf{x}^{(p,k+1)} = \begin{cases} \mathbf{U}^{(p,k)}, & \text{if } f\left(\mathbf{U}^{(p,k)}\right) \le f\left(\mathbf{x}^{(p,k)}\right) \\ \mathbf{x}^{(p,k)}, & \text{otherwise} \end{cases} \tag{17.10}$$

Accordingly, if the cost function value for the trial design point does not exceed that for the parent design, it replaces the parent design point in the next generation; otherwise, the parent design is retained. Thus the population either gets better or remains the same in fitness status, but it never deteriorates. Note that in Eq. (17.10) the parent design is replaced by the trial design even if both yield the same value for the cost function. This allows the design vectors to move over the flat fitness landscape.

17.2.5 Differential Evolution Algorithm

The basic DEA is quite straightforward to implement. It requires specification of only three parameters: N_p, F, and Cr. A flow diagram describing the basic steps of the algorithm is shown in Fig. 17.5.

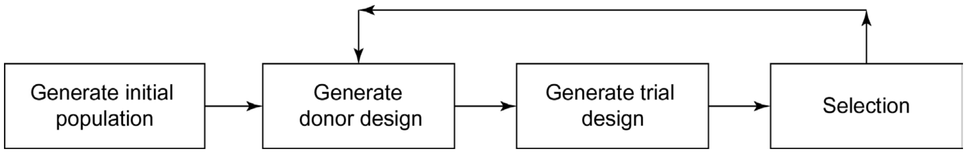

FIGURE 17.5 **Main steps of the DEA.**

The termination criteria for the algorithm are defined as follows:

1. A specified limit $k_{\max}$ on the number of generations is reached.
2. The best fitness/cost function value of the population does not change appreciably for several generations.
3. A prespecified value for the cost function is reached.

Because of its simplicity, the DEA has been quite popular in many application fields since its inception in the mid-1990s. It was inspired by the Nelder and Mead (1965) direct search method, which also uses the difference of vectors, as described in chapter: More on Numerical Methods for Unconstrained Optimum Design. Numerous variations of the algorithm have been studied and evaluated. It has been used to solve continuous variable, mixed–discrete–continuous variable, and multi-objective optimization problems, and it has also been evaluated against many other nature-inspired algorithms. A detailed review is beyond the scope of the present text. An excellent recent survey paper by Das and Suganthan (2011) and numerous references cited there should be consulted.

EXAMPLE 17.3 APPLICATION OF DEA

Minimize

$$f(x) = (x_1 - 1)^2 + (x_2 - 2)^2 \tag{a}$$

subject to:

$$-10 \le x_1 \le 10, \; -10 \le x_2 \le 10 \tag{b}$$

Solution

For the example problem, the DEA parameters are set to be within their recommended ranges as follows:

$n = 2$ since there are only two design variables in this problem.
$N_p = 10$ since the problem contains only 2 design variables, $5 \times 2 = 10$, N_p is set to 10.
$k_{\max} = 10{,}000$ Iterations.
$Cr = 0.8$
$F = 0.6$

Step 1: Generation of an Initial Population. The initial population is generated using Eq. (17.7). Table 17.2 shows the initial population:

TABLE 17.2 Initial Population for Example 17.3

x_i *number*	x_1	x_2
1	3.717	−1.600
2	9.400	−4.380
3	9.048	−8.659
4	−2.935	−2.920
5	−5.423	3.962
6	−4.442	2.470
7	−0.848	7.648
8	−8.394	−5.238
9	2.678	−2.884
10	7.059	−1.567

Step 2: Generation of Donor Design. As described earlier the generation of the donor design requires randomly selecting three distinct design points: $\mathbf{x}^{(r_1,k)}$, $\mathbf{x}^{(r_2,k)}$ and $\mathbf{x}^{(r_3,k)}$, in addition to a fourth design point which is the *parent/target* design $\mathbf{x}^{(p,k)}$. For the first iteration the four randomly selected design points are as follows:

$$\begin{aligned}\mathbf{x}^{(r_1,1)} &= (-5.423, \quad 3.962)\\ \mathbf{x}^{(r_2,1)} &= (9.40, \quad -4.380)\\ \mathbf{x}^{(r_3,1)} &= (-0.848, \quad 7.648)\\ \mathbf{x}^{(p,1)} &= (3.717, \quad -1.600)\end{aligned} \tag{c}$$

The donor following design is generated according to Eq. (17.8) as:

$$\mathbf{V}^{(p,1)} = (0.725, \quad -3.254) \tag{d}$$

Step 3: Crossover Operation to Generate the Trial Design. The cross over operation is accomplished as described in Eq. (17.9). In the first iteration, the randomly distributed numbers r_{p1} and r_{p2} were 0.13 and 0.56, which are both less than the *Cr,* which means both components of the trial design should come from the donor design.

$$\mathbf{U}^{(p,1)} = \mathbf{V}^{(p,1)} = (0.725, \quad -3.254) \tag{e}$$

Step 4: Acceptance/Rejection of the Trial Design. The trial design is accepted and replaces the parent design in the next iteration if it has a better (smaller) cost function value than that for the parent design. In the first iteration the cost function value for the trial design is $f\left(\mathbf{U}^{(p,1)}\right) = 27.686$ and for the parent design $f\left(\mathbf{x}^{(p,1)}\right) = 20.342$, which means that the parent design is retained in the next iteration.

The previous steps are repeated until the maximum number of iterations $k_{\max}$ is reached. After 10,000 iterations the trial design point (0.97, 1.96) was reached with a cost function value of 0.00222 which is close to the true solution of (1, 2) with a cost function value of 0.0.

17.3 ANT COLONY OPTIMIZATION

Ant colony optimization (ACO), another nature-inspired approach, emulates the food-searching behavior of ants. It was developed by Dorigo (1992) to search for an optimal path for a problem represented by a graph based on the behavior of ants seeking the shortest path between their colony and a food source. ACO falls into the class of metaheuristics and swarm intelligence methods. It can be viewed as a stochastic technique for solving computational problems that can be reduced to finding optimal paths through graphs.

Ants are social insects that live in colonies. From the colony, they go out to search for food and, surprisingly, find the shortest path from the colony to the food source. In this section, the process that ants use is described and translated into a computational algorithm for design optimization. The algorithm was developed originally for discrete variable combinatorial optimization problems, although it has been applied to continuous variable and other problems as well. Some of the material in this section is derived from Blum (2005) and associated references.

ACO uses the following terminology:

Pheromone: The word is derived from the Greek words *pherin* (to transport) and *hormone* (to stimulate). It refers to a secreted or excreted chemical factor that triggers a social response in members of the same species. Pheromones are capable of acting outside the body of the secreting individual in order to impact the behavior of the receiving individual. This is also called a chemical messenger.

Pheromone trail: Ants deposit pheromones wherever they go. This is called the pheromone trail. Other ants can smell the pheromones and are likely to follow an existing trail.

Pheromone density: When ants travel on the same path again and again, they continuously deposit pheromones on it. In this way the amount of pheromones increases and is called the pheromone density. The ants are likely to follow paths having higher pheromone densities.

Pheromone evaporation: Pheromones have the property of evaporation over time. Therefore, if a path is not being traveled by the ants, the pheromones evaporate, and the path disappears over time.

17.3.1 Ant Behavior

A first step in developing the ACO algorithm is to understand the behavior of ants, which is described in this subsection. Initially ants move from their nest randomly to search for food. Upon finding it, they return to their colony following the path they took to it while laying down pheromone trails. If other ants find such a path, they are likely to follow it instead of moving randomly. The path is thus reinforced, since ants deposit more pheromone on it. However, the pheromone evaporates over time; the longer the path, the more time there is available for it to evaporate. For a shorter path, pheromone reinforcement is quicker as more and more ants travel this route. Therefore, the pheromone density is higher on shorter paths than on the longer ones. Pheromone provides a positive feedback mechanism for ants, so eventually all the ants follow the shortest path.

The basic idea of an ant colony algorithm is to emulate this behavior with "virtual ants," which means that we need to model the pheromone deposit, measure its density, and model its evaporation. The following notation and terminology are used in this section:

Q	= positive constant; an algorithm parameter
ρ	= pheromone evaporation rate, $\rho \in (0, 1]$; an algorithm parameter
N_a	= number of ants
τ_i	= pheromone value for the ith path

A Simple Model/Algorithm

To transcribe the ants' food-searching behavior into a computational algorithm, we consider a simplified model consisting of two paths from the ant colony to the food source and six ants, as shown in Fig. 17.6a. This is a highly idealized model, introduced to explain the transcription of ant behavior into a computational algorithm. The model can be represented in a graph $G = (N, L)$, where N consists of two nodes (n_c, representing the ant colony, and n_f, representing the food source; in general a graph has many nodes as seen later), and L consists of two links, L_1 and L_2, between n_c and n_f.

Let L_1 have a length of d_1, and L_2 a length of d_2, with $d_1 > d_2$, implying that L_2 is a shorter path between n_c and n_f. Fig. 17.6 is a graph that shows various stages of ant movement, which are explained as follows:

1. Six ants start from their colony in search for food. Randomly, three ants (shown as solid circles) take the shorter route and three (shown as open circles) take the longer route.
2. The three ants that took the shorter route have reached their destination, while the ants on the longer route are still traveling. Initially, the pheromone concentration is the same for the two routes, as shown by the dashed lines.

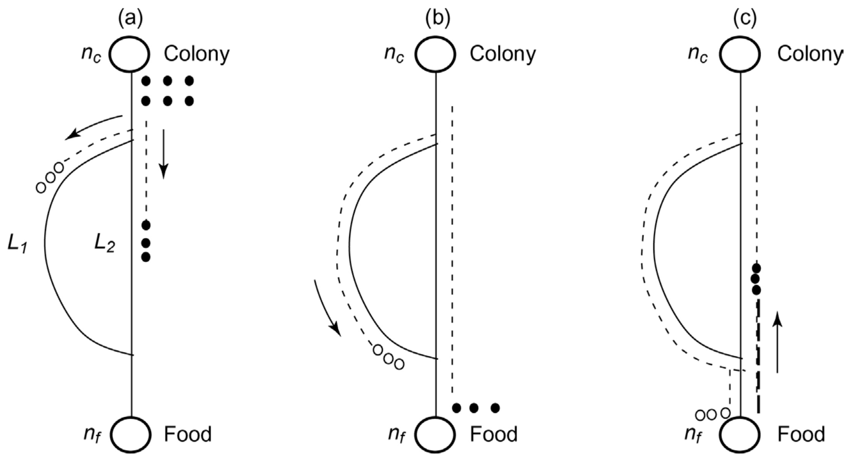

FIGURE 17.6 **A simple set up showing shortest path finding capability of ants.** (a) Movement of ants from colony to food source. (b) Ants taking the shorter route have reached the food source. (c) Ants taking the longer route have reached the food source while ants that took the shorter route are already returning to their colony.

3. The ants that took the shorter route are on their return journey to the colony while the ants taking the longer route are just arriving at their destination. Pheromone concentration on the shorter route is higher, as shown by the heavier dashed line.

The ants deposit pheromone while traveling on a route. The pheromone trails are modeled by introducing a virtual pheromone value τ_i for each of the two routes, $i = 1, 2$ (initially both values may be set as one). Such a value indicates the strength of the pheromone trail on the corresponding route.

Each ant behaves as follows: Starting from the node n_c (ie, the colony), the ant chooses between route L_1 and route L_2 to reach n_f with the probability:

$$p_i = \frac{\tau_i}{\tau_1 + \tau_2}, \quad i = 1, 2 \tag{17.11}$$

If $\tau_2 > \tau_1$, the probability of choosing L_2 is higher, and vice versa. The selection of a path by an ant is based on some selection scheme that uses probabilities from Eq. (17.11) and a random number, such as the roulette wheel selection procedure described earlier in Section 17.1.2. While returning from the node n_f to the node n_c, the ant uses the same route it chose to reach n_f. It deposits additional virtual pheromone on the route to increase its density (this is also called pheromone reinforcement) as follows:

$$\tau_i \leftarrow \tau_i + \frac{Q}{d_i} \tag{17.12}$$

where the positive constant Q is a parameter of the model. Equation (17.12) models the higher amount of virtual pheromone deposit for a shorter path and a smaller amount for a longer path.

In the iterative process, all ants start from the node n_c at the beginning of each iteration. Each ant moves from that node n_c to node n_f depositing pheromone on the chosen route. However, with time the pheromone is subject to evaporation. This evaporation process in the virtual model is simulated as follows:

$$\tau_i \leftarrow (1 - \rho)\tau_i \tag{17.13}$$

where $\rho \in (0, 1]$ is a parameter of the model that regulates evaporation. After reaching the food source, the ants return to their colony, reinforcing the chosen path by depositing more pheromone on it.

17.3.2 ACO Algorithm for the Traveling Salesman Problem

The procedure described in the previous subsection to simulate the food-searching behavior of ants cannot be used directly for combinatorial optimization problems. The reason is that we assume the solution to the problem to be known and the pheromone values to be associated with the solution, as in Eq. (17.12). In general this is not the case because we are trying to find the optimum solution and the associated path with the minimum distance. Therefore, for combinatorial optimization problems, the pheromone values are associated with the solution components. Solution components are the units from which the entire solution to the problem can be constructed. This will become clearer later, when we describe the ACO algorithm for combinatorial optimization problems.

In this subsection, we describe an ant colony algorithm for discrete variable, or *traveling salesman* (TS), problems. The TS problem is a classical *combinatorial optimization* problem. In it a traveling salesman is required to visit a specified number of cities (called a *tour*). The goal is to visit a city only once while minimizing the total distance traveled. Many practical problems can be modeled as the TS problem; another example is the welding sequence problem described earlier in Example 17.2.

The following assumptions are made in deriving the algorithm:

1. While a real ant can take a return path to the colony that is different from the original path depending on the pheromone values, a virtual ant takes the return path that is the same as the original path.
2. The virtual ant always finds a feasible solution and deposits pheromone only on its way back to the nest.
3. While real ants evaluate a solution based on the length of the path from their nest to the food source, virtual ants evaluate their solution based on a cost function value.

To describe the ACO algorithm for the TS problem, we consider a simple problem of touring four cities by the traveling salesman. The situation is depicted in Fig. 17.7, where the cities are represented as the nodes c_1 through c_4 of the graph, with distances between the cities known. From each city, there are links to other cities; that is, the salesman can travel to any other city, but travel to the already visited cities (ie, backtracking) is not allowed. Thus, a feasible solution to the problem consists of a sequence of cities visited on a tour—for example, $c_1c_3c_2c_4c_1$. The distance traveled on a tour is the cost function $f(\mathbf{x})$, which depends on the links used.

The definition of the task for the virtual ant changes from "finding a path from the nest to the food source" to "finding a feasible solution to the TS problem."

The TS tour must start from a city that can be randomly selected. We will call it c_1; the remaining cities are numbered randomly. To complete a four-city tour, four links need to be selected. The following notation and terminology are used in this subsection:

Q_a = positive constant; an algorithm parameter
ρ = pheromone evaporation rate, $\rho \in (0, 1]$; an algorithm parameter
n = number of design variables; four for the example

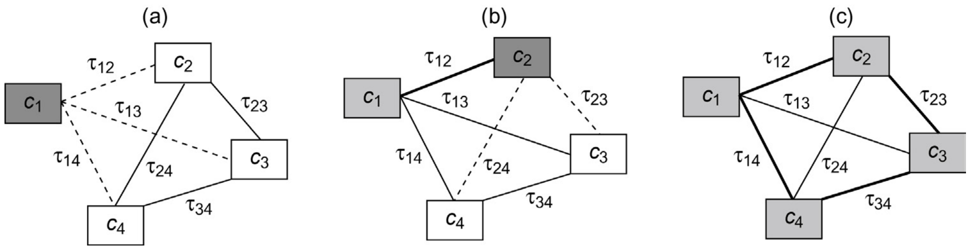

FIGURE 17.7 **Traveling salesman problem for four cities.** (a) Start of tour at c_1; feasible links from the current city are shown by dashed lines; current city is displayed with darker shading. (b) The link already traveled is shown with a thicker line; city already visited is displayed with lighter shading. (c) A feasible solution is shown.

N_a = number of virtual ants used in the algorithm
τ_{ij} = pheromone value for the link ij
x_j = jth component of the design variable vector $\mathbf{x}$; represents the link selected from the jth city
x_{ij} = link between the ith city and the jth city; also represents the distance between them
D_i = the list of integers corresponding to the cities that can be visited from the ith city

For the example in Fig. 17.7, x_{12}, x_{13} and x_{14} are the links from city c_1 to cities c_2, c_3 and c_4, respectively; $D_1 = \{2, 3, 4\}$ for city c_1, with the associated feasible links given as $\{x_{12}, x_{13}, x_{14}\}$. The design variable vector is given as $\mathbf{x} = [x_1\ x_2\ x_3\ x_4]^T$. A feasible solution to the problem is given as $\mathbf{x} = [x_{12}\ x_{24}\ x_{43}\ x_{31}]^T$.

Now let us begin the tour. From each city, selection of the next city to visit by the virtual ant is based on certain probabilities. For the ACO algorithm, the probabilities are calculated using the pheromone values τ_{ij} for each of the links from the current city; initially all τ_{ij} can be selected as 1 for all links. Also, the number of virtual ants N_a is selected as reasonable depending on the number of design variables (say $5n$ to $10n$). Individual ants can start randomly from any city. Their task is to construct a feasible solution (ie, a feasible tour) for the TS problem, one component at a time; that is, from each city visited, a link to the next feasible city is determined in a sequence.

Each ant constructs a feasible solution (tour) for the problem, starting from a randomly selected city and moving from one city to another one that has not been visited. At each step, the traveled link is added to the solution under construction by a specific ant. In this way the ACO algorithm constructs a solution one component at a time: For example, x_1 and then x_2, and so on. Different ants pursue feasible solutions concurrently, although different ants may find the same one. When no unvisited city is left for a specific ant, that ant moves to the starting city to complete the tour. This solution process implies that an ant has memory M to store already visited cities. Using this memory, we can construct an index set D_i of feasible cities to visit from the current city i.

The ACO algorithm constructs a feasible solution, one component (ie, one design variable) at a time.

Fig. 17.7a shows the starting city for a virtual ant as c_1; the starting city is identified by darker shading. The feasible links from the city are shown with dashed lines: $D_1 = \{2, 3, 4\}$, and the associated link list is $\{x_{12}, x_{13}, x_{14}\}$. The probability of taking a feasible route from the ith city is calculated as

$$p_{ij} = \frac{\tau_{ij}}{\sum_{k \in D_i} (\tau_{ik})}; \quad \text{for all} \quad j \in D_i \tag{17.14}$$

where D_i is the list of feasible cities that can be visited from city i. For Fig. 17.7a, the probabilities for the cities that can be visited from city c_1 are calculated as

$$p_{1j} = \frac{\tau_{1j}}{\tau_{12} + \tau_{13} + \tau_{14}}; \quad j = 2, 3, 4 \tag{17.15}$$

Once these probabilities are calculated, a selection process is used for the route and the city to visit next. The roulette wheel selection process described earlier in Section 17.1.2, or any other procedure, can be used for this. That process requires calculation of a random number between 0 and 1. Based on it, let the next city to visit be c_2. Thus the link x_{12} is used here and the design variable x_1 is set as x_{12}. This is shown by a darker line in Fig. 17.7(b). From c_2, city c_3 or c_4 can be visited. This is shown by the dashed lines in Fig. 17.7b. The cities that have already been visited are shown by lighter shading. Therefore, $D_2 = \{3, 4\}$ and the associated link list is $\{x_{23}, x_{24}\}$. The probabilities of visiting cities c_3 and c_4 from city c_2 are given as:

$$p_{2j} = \frac{\tau_{2j}}{\tau_{23} + \tau_{24}}; \quad j = 3, 4 \tag{17.16}$$

Using the foregoing procedure, the virtual ant completes the tour as follows:

$$c_1 \to c_2 \to c_3 \to c_4 \to c_1 \tag{17.17}$$

This gives the design variable values as:

$$\mathbf{x} = \left[x_{12}\ x_{23}\ x_{34}\ x_{41}\right]^T \tag{17.18}$$

Using this design, the cost function $f(\mathbf{x})$, which is the total distance traveled on this tour, can be calculated.

Once all virtual ants have constructed their solution, pheromone evaporation (ie, a reduction in the pheromone density for each link) is performed as follows:

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} \quad \text{for all } i \text{ and } j \tag{17.19}$$

Now the virtual ants start their return journey, depositing pheromone on the path that was used to reach the destination. This is equivalent to increasing the pheromone level for the links belonging to each ant's solution. For the kth ant, pheromone deposit is performed as follows:

$$\tau_{ij} \leftarrow \tau_{ij} + \frac{Q}{f\left(\mathbf{x}^{(k)}\right)} \quad \text{for all } i, j \text{ belonging to } k\text{th ant's solution} \tag{17.20}$$

where Q is a positive constant and $f(\mathbf{x}^{(k)})$ is the cost function value for the kth ant's solution $\mathbf{x}^{(k)}$. The process of pheromone deposit in Eq. (17.20) is repeated for the solution of each of the N_a ants. Note that a tour (solution) that has a smaller cost function value deposits a larger pheromone value. Also, a link that is traveled in multiple solutions receives a pheromone deposit multiple times.

The foregoing process represents one iteration of the ACO algorithm. It is repeated several times until a stopping criterion is satisfied—that is, all ants follow the same route or the limit on the number of iterations or on CPU time is reached.

17.3.3 ACO Algorithm for Design Optimization

Problem Definition

In this subsection, we discuss the ACO algorithm for the following unconstrained discrete variable design optimization problem:

Minimize

$$f(\mathbf{x}) \tag{17.21}$$

$$x_i \in D_i; \quad D_i = \left(d_{i1},\ d_{i2},\ \ldots,\ d_{iq_i}\right), \quad i = 1 \text{ to } n \tag{17.22}$$

where D_i is the set of discrete values and q_i is the number of discrete values allowed for the ith design variable. This type of design problem is encountered quite frequently in practical applications, as was discussed in chapter: Discrete Variable Optimum Design Concepts and Methods. For example, the thickness of members must be selected from an available set, structural members must be selected from the members available in the catalog, concrete reinforcing bars must be selected from the available bars on the market, and so forth.

The problem described in Eqs. (17.21) and (17.22) is quite similar to the TS problem described and discussed in the previous subsection. One major difference is that the set of available values for a design variable is predefined, whereas for the TS problem it must be determined once a city is reached (ie, once a component of the design variable vector has been determined). The procedure described in the previous subsection can be adapted to solve this discrete variable optimization problem.

EXAMPLE 17.4 DESCRIPTION OF ACO WITH AN EXAMPLE

To describe the solution algorithm, we consider a simpler problem having three design variables, with each variable having four allowable discrete values. Therefore, $n = 3$, and $q_i = 4$, $i = 1$ to 4 in Eqs. (17.21) and (17.22). The problem can be displayed in a multilayered graph as shown in Fig. 17.8. The graph shows the starting node 00 as the nest and the destination node as the food source. The starting point is called Layer 0. Layer 1 represents the allowable values for the design variable x_1 in the set D_1; each allowable value is represented as a node, such as node d_{12}. There are links from the nest to each of these nodes. Level 2 represents the allowable values for the design variable x_2 as nodes. For example, from d_{13} there are links to d_{21}, d_{22}, d_{23}, and d_{24}. Similarly, from d_{11} there are links to d_{21}, d_{22}, d_{23}, and d_{24}, and so on (note that all these links are not shown in Fig. 17.8).

The ACO algorithm proceeds as follows: An ant starts from the nest and chooses a link to travel to a node at Layer 1 based on probabilities such as the link to node d_{13}; that is, design variable x_1 is assigned the value d_{13} From this node, the probabilities are calculated again for all links to the next layer on the graph, and the ant moves to, say, node d_{22}. This procedure is repeated for the next layer, and the ant moves to node d_{34}. Since there are no further layers, this ant has reached its destination. Its feasible solution is obtained as $\mathbf{x} = (d_{13}, d_{22}, d_{34})$, with the cost function value as $f(\mathbf{x})$. The route for this ant is shown by the darker lines in Fig. 17.8.

Once all the ants have found feasible solutions, pheromone evaporation is performed for all of the links using Eq. (17.19) or one similar to it. Then each ant traces its path back to the nest, depositing pheromone using Eq. (17.20) or one similar to it on each link that it previously traveled. This is equivalent to updating (increasing) the pheromone values for the links traveled by the ants. The entire process is then repeated until a stopping criterion is satisfied.

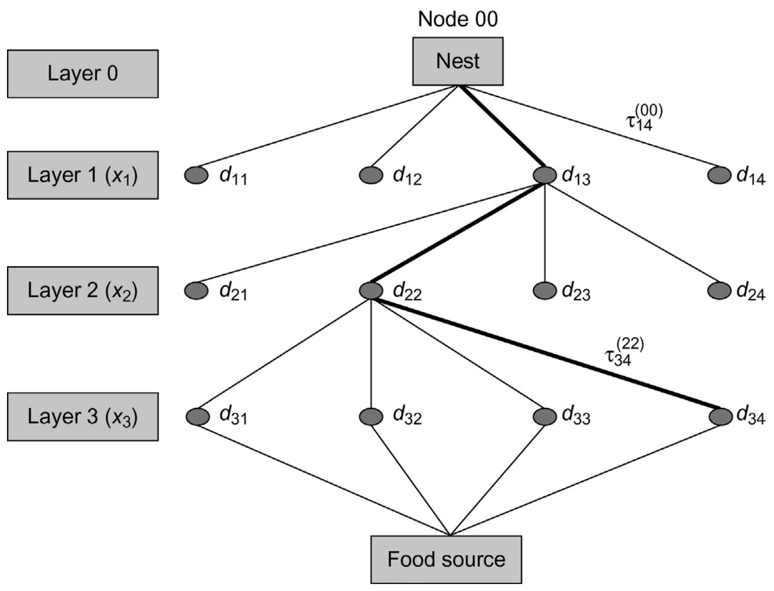

FIGURE 17.8 **A multilayered graphical representation of a discrete variable problem with 3 design variables each one having 4 allowable values; the links chosen by the ant are shown using the darker lines.** (Note that all possible links are not shown.)

Finding Feasible Solutions

The foregoing procedure can be generalized to the case of *n* design variables (an *n*-layered graph), each having q_i discrete values. The following notation is used:

$\tau_{ij}^{(rs)}$ = pheromone value for the link from node *rs* to node *ij*; note that since the procedure moves from one layer to the next, $i = r + 1$ (eg, $\tau_{34}^{(22)}$ between nodes d_{22} and d_{34} in Fig. 17.8). Thus, the superscript *r* represents the layer number (design variable number), the superscript *s* represents the allowable value number for the design variable number *r*, subscript *i* represents the next layer (next design variable), and subscript *j* represents the allowable design variable number for the *i*th design variable.
$p_{ij}^{(rs)}$ = probability of selection of the link from node *rs* to node *ij*.

To find a feasible solution for a virtual ant *k*, the following steps are suggested.

STEP 1. SELECTION OF AN INITIAL LINK

Ant *k* starts from the nest (ie, node 00 of layer 0). Calculate probabilities for the links from node 00 to all nodes for layer 1 (design variable x_1) as follows:

$$p_{1j}^{(00)} = \frac{\tau_{1j}^{(00)}}{\sum_{r=1}^{q_1} \tau_{1r}^{(00)}}; \quad j = 1 \text{ to } q_1 \tag{17.23}$$

Using these probabilities and a selection process, choose a link to a node at layer 1 and go to that node. Let this node be k_1; the design variable x_1 is thus assigned the value d_{kl}.

STEP 2. SELECTION OF A LINK FROM LAYER R

Let ant k be at node rs. Calculate probabilities of the links from node rs to all nodes at the next layer:

$$p_{ij}^{(rs)} = \frac{\tau_{ij}^{(rs)}}{\sum_{l=1}^{q_1} \tau_{il}^{(rs)}}; \quad j = 1 \text{ to } q_i \tag{17.24}$$

Note that $i = r + 1$. Using these probabilities and a selection procedure, select a link to the next layer and the corresponding node for ant k to travel. Repeat this step until the nth layer is reached, at which point ant k has reached its destination and a feasible solution has been obtained.

STEP 3. OBTAINING FEASIBLE SOLUTIONS FOR ALL ANTS

Repeat steps 1 and 2 for each virtual ant to obtain all N_a feasible solutions. Let the solutions and the corresponding cost function values be represented as:

$$\mathbf{x}^{(k)}, \ f\left(\mathbf{x}^{(k)}\right); \quad k = 1 \text{ to } N_a \tag{17.25}$$

Pheromone Evaporation

Once all of the ants have reached their destination (all of them have found solutions), pheromone evaporation (ie, reduction in the pheromone level) is performed for all links as follows:

$$\tau_{ij}^{(rs)} \leftarrow (1-\rho)\tau_{ij}^{(rs)} \ \text{ for all } r, s, i \text{ and } j \tag{17.26}$$

Pheromone Deposit

After pheromone evaporation, the ants start their journey back to their nest, which means that they will deposit pheromone on the return trail. This involves increasing the pheromone density of the links that they have traveled. For the kth ant, the pheromone deposit is performed as follows:

$$\tau_{ij}^{(rs)} \leftarrow \tau_{ij}^{(rs)} + \frac{Q}{f\left(\mathbf{x}^{(k)}\right)} \quad \text{for all} \quad r, s, i, j \quad \text{belonging to } k\text{th ant's solution} \tag{17.27}$$

The operation in Eq. (17.27) is performed for all solutions obtained by the ants. It is seen that the solutions that have a smaller cost function value receive more pheromone deposit. Also, the links that are traveled multiple times receive more reinforcement of pheromone. A larger value of the pheromone for a link gives a larger probability value from Eq. (17.23), which favors it selection for travel by the virtual ants in subsequent iterations of the ACO algorithm.

We see that the ACO algorithm is quite simple to implement, requiring specification of only three parameters, N_a, ρ, and Q. N_a can be given a reasonable value of say, $5n$ to $10n$; $\rho \in [0, 1]$, a value of say, 0.4 to 0.8; Q may be selected as a typical value for the cost function $f(\mathbf{x})$.

17.4 PARTICLE SWARM OPTIMIZATION

Particle swarm optimization (PSO), another nature-inspired method, mimics the social behavior of bird flocking or fish schooling. It falls into the class of *metaheuristics* and *swarm intelligence* methods. It is also a population-based stochastic optimization technique, introduced by Kennedy and Eberhart (1995). PSO shares many similarities with evolutionary computation techniques such as GA and DE. Just like those approaches, PSO starts with a randomly generated set of solutions called the initial population. An optimum solution is then searched by updating generations.

An attractive feature of PSO is that it has fewer algorithmic parameters to specify compared to GAs. It does not use any of the GAs' evolutionary operators such as crossover and mutation. Also, unlike GAs, the algorithm does not require binary number encoding or decoding and thus is easier to implement into a computer program. PSO has been successfully applied to many classes of problems, such as mechanical and structural optimization and multi-objective optimization, artificial neural network training, and fuzzy system control.

In this section, we present the basic ideas of PSO and a simple PSO algorithm. Many variations on the method are available in the literature, and research on the subject continues to develop better algorithms and expand the range of their application (Kennedy et al., 2001).

17.4.1 Swarm Behavior and Terminology

The PSO computational algorithm tries to emulate the social behavior of a swarm of animals, such as a flock of birds or a school of fish (moving in search for food). In a swarm, an individual behaves according to its limited intelligence as well as to the intelligence of the group. Each individual observes the behavior of its neighbors and adjusts its own behavior accordingly. If an individual member discovers a good path to food, other members follow this path no matter where they are situated in the swarm.

PSO uses the following terminology:

Particle: This term is used to identify an individual in the swarm (eg, a bird in the flock or a fish in the school). *Agent* is also used in some circles. Each particle has a location in the swarm. In the optimization algorithm, each particle location represents a design point that is a potential solution to the problem.

Particle position: This term refers to the coordinates of the particle. In the optimization algorithm, it refers to a design point (a vector of design variables).

Particle velocity: The term refers to the rate at which the particles are moving in space. In the optimization algorithm, it refers to the design change.

Swarm leader: This is the particle having the best position. For the optimization algorithm, the term refers to a design point having the smallest value for the cost function.

17.4.2 Particle Swarm Optimization Algorithm

The PSO translates the social behavior of the swarm described previously into a computational algorithm. The notation shown in Table 17.3 is used in the subsequent step-by-step algorithm.

Each particle in the swarm keeps track of its own current position and its best position (solution) achieved during the running of the algorithm. This implies that each point stores not only its current value but also its best value achieved thus far. The best position for the ith particle (design point) is denoted $\mathbf{x}_P^{(i,k)}$. Another "best" value that is tracked by the particle swarm optimizer is the best position for the entire swarm, denoted $\mathbf{x}_G^{(k)}$. The PSO algorithm consists of changing, at each time step (iteration), the velocity of each particle toward its own best position as well as the swarm's best position (also sometimes referred to as accelerating the particle toward the best known position).

The step-by-step PCO algorithm is stated as follows.

Step 0: *Initialization*. Select N_p, c_1, c_2, and $k_{\max}$ as the maximum number of iterations. Set the initial velocity of the particle $\mathbf{v}^{(i,0)}$ to 0. Set the iteration counter at $k = 1$.
Step 1: *Initial generation*. Using a random procedure, generate N_p particles $\mathbf{x}^{(i,0)}$. The procedure described in Eq. (17.7) can be used to generate these points within their allowable ranges. Evaluate the cost function for each of these points $f(\mathbf{x}^{(i,0)})$. Determine the best solution among all particles as $\mathbf{x}_G^{(k)}$—that is, a point having the smallest cost function value.

TABLE 17.3 Notation and Terminology for the Particle Swarm Optimization Algorithm

Notation	**Terminology**
c_1	Algorithm parameter (ie, cognitive parameter); taken between 0 and 4, usually set to 2
c_2	Algorithm parameter (ie, social parameter); taken between 0 and 4, usually set to 2
r_1, r_2	Random numbers between 0 and 1
k	Iteration counter
$k_{\max}$	Limit on the number of iterations
n	Number of design variables
N_p	Number of particles (design points) in the swarm; *swarm size* (usually 5n to 10n)
x_j	jth component of the design variable vector $\mathbf{x}$
$\mathbf{v}^{(i,k)}$	Velocity of the ith particle (design point) of the swarm at the kth generation/iteration
$\mathbf{x}^{(i,k)}$	Location of the ith particle (design point) of the swarm at the kth generation/iteration
$\mathbf{x}_P^{(i,k)}$	Best position of the ith particle based on its travel history at the kth generation/iteration
$\mathbf{x}_G^{(k)}$	Best solution for the swarm at the kth generation; considered the leader of the swarm
$\mathbf{x}_L$	Vector containing lower limits on the design variables
$\mathbf{x}_U$	Vector containing upper limits on the design variables

Step 2: *Calculate velocities*. Calculate the velocity of each particle at the $k + 1$ iteration as:

$$\mathbf{v}^{(i,k+1)} = \mathbf{v}^{(i,k)} + c_1 r_1 \left(\mathbf{x}_P^{(i,k)} - \mathbf{x}^{(i,k)}\right) + c_2 r_2 \left(\mathbf{x}_G^{(k)} - \mathbf{x}^{(i,k)}\right); \quad i = 1 \text{ to } N_p \tag{17.28}$$

Update the positions of the particles as:

$$\mathbf{x}^{(i,k+1)} = \mathbf{x}^{(i,k)} + \mathbf{v}^{(i,k+1)}; \quad i = 1 \text{ to } N_p \tag{17.29}$$

Check and enforce bounds on the particle positions:

$$\mathbf{x}_L \le \mathbf{x}^{(i,k+1)} \le \mathbf{x}_U \tag{17.30}$$

Step 3: *Update the best solution*. Calculate the cost function at all new points $f(\mathbf{x}^{(i,k+1)})$. For each particle, perform the following check:

$$\begin{aligned} &\text{If } f\left(\mathbf{x}^{(i,k+1)}\right) \le f\left(\mathbf{x}_P^{(i,k)}\right), \quad \text{then} \quad \mathbf{x}_P^{(i,k+1)} = \mathbf{x}^{(i,k+1)}; \\ &\text{otherwise} \quad \mathbf{x}_P^{(i,k+1)} = \mathbf{x}_P^{(i,k)} \quad \text{for each} \quad i = 1 \text{ to } N_p \end{aligned} \tag{17.31}$$

$$\text{If } f\left(\mathbf{x}_P^{(i,k+1)}\right) \le f\left(\mathbf{x}_G\right), \quad \text{then} \quad \mathbf{x}_G = \mathbf{x}_P^{(i,k+1)}, \quad i = 1 \text{ to } N_p \tag{17.32}$$

Step 4: *Stopping criterion*. Check for convergence of the iterative process. If a stopping criterion is satisfied (ie, $k = k_{\max}$ or if all of the particles have converged to the best swarm solution), stop. Otherwise, set $k = k + 1$ and go to step 2.

EXERCISES FOR CHAPTER 17*

Section 17.1 Genetic Algorithm

Solve the following problems using a GA.

17.1 Example 15.1 with the available discrete values for the variables as $x_1 \in \{0, 1, 2, 3\}$, and $x_2 \in \{0, 1, 2, 3, 4, 5, 6\}$. Compare the solution with that obtained with the branch and bound method.

17.2 Exercise 3.34 using the outside diameter d_0 and the inside diameter d_i as design variables. The outside diameter and thickness must be selected from the following available sets:

$$d_0 \in \{0.020, 0.022, 0.024, \dots, 0.48, 0.50\}\,\text{m}; \quad t \in \{5, 7, 9, \dots, 23, 25\}\,\text{mm}$$

Check your solution using the graphical method of chapter: Graphical Solution Method and Basic Optimization Concepts. Compare continuous and discrete solutions. Study the effect of reducing the number of elements in the available discrete sets.

17.3 Formulate the minimum mass tubular column problem described in Section 2.7 using the following data: $P = 100$ kN, length, $l = 5$ m, Young's modulus, $E = 210$ GPa, allowable stress, $\sigma_a = 250$ MPa, mass density, $\rho = 7850\ \text{kg/m}^3$, $R \le 0.4$ m, $t \le 0.05$ m, and $R, t \ge 0$. The design variables must be selected from the following sets:

$$R \in \{0.01, 0.012, 0.014, \ldots, 0.38, 0.40\}\,\text{m}; \quad t \in \{4, 6, 8, \ldots, 48, 50\}\,\text{mm}$$

Check your solution using the graphical method of chapter: Graphical Solution Method and Basic Optimization Concepts. Compare continuous and discrete solutions. Study the effect of reducing the number of elements in the available discrete sets.

17.4 Consider the plate girder design problem described and formulated in Section 6.8. The design variables for the problem must be selected from the following sets:

$$h, b, \in \{0.30, 0.31, 0.32, \ldots, 2.49, 2.50\}\,\text{m}; \quad t_w, t_f \in \{10, 12, 14, \ldots, 98, 100\}\,\text{mm}$$

Compare the continuous and discrete solutions. Study the effect of reducing the number of elements in the available discrete sets.

17.5 Consider the plate girder design problem described and formulated in Section 6.8. The design variables for the problem must be selected from the following sets:

$$h, b, \in \{0.30, 0.32, 0.34, \ldots, 2.48, 2.50\}\,\text{m}; \quad t_w, t_f \in \{10, 14, 16, \ldots, 96, 100\}\,\text{mm}$$

Compare the continuous and discrete solutions. Study the effect of reducing the number of elements in the available discrete sets.

17.6 Solve problems of Exercises 17.4 and 17.5. Compare the two solutions, commenting on the effect of the size of the discreteness of variables on the optimum solution. Also, compare the continuous and discrete solutions.

17.7 Formulate the spring design problem described in Section 2.9 and solved in Section 6.7. Assume that the wire diameters are available in increments of 0.01 in., the coils can be fabricated in increments of $1/16$ in., and the number of coils must be an integer. Compare the continuous and discrete solutions. Study the effect of reducing the number of elements in the available discrete sets.

17.8 Formulate the spring design problem described in Section 2.9 and solved in Section 6.7. Assume that the wire diameters are available in increments of 0.015 in., the coils can be fabricated in increments of $1/8$ in., and the number of coils must be an integer. Compare the continuous and discrete solutions. Study the effect of reducing the number of elements in the available discrete sets.

17.9 Solve problems of Exercises 17.7 and 17.8. Compare the two solutions, commenting on the effect of the size of the discreteness of variables on the optimum solution. Also, compare the continuous and discrete solutions.

17.10 Formulate the problem of optimum design of prestressed concrete transmission poles described in Kocer and Arora (1996a). Compare your solution to that given in the reference.

17.11 Formulate the problem of optimum design of steel transmission poles described in Kocer and Arora (1996b). Solve the problem as a continuous variable optimization problem.

17.12 Formulate the problem of optimum design of steel transmission poles described in Kocer and Arora (1996b). Assume that the diameters can vary in increments of 0.5 in. and the thicknesses can vary in increments of 0.05 in. Compare your solution to that given in the reference.

17.13 Formulate the problem of optimum design of steel transmission poles using standard sections described in Kocer and Arora (1997). Compare your solution to the solution given in the reference.

17.14 Formulate and solve three-bar truss of Exercise 3.50 as a discrete variable problem where the cross-sectional areas must be selected from the following discrete set:

$$A_i \in \{50, 100, 150, \ldots, 4950, 5000\}\ \text{mm}^2$$

Check your solution using the graphical method of chapter: Graphical Solution Method and Basic Optimization Concepts. Compare continuous and discrete solutions. Study the effect of reducing the number of elements in the available discrete sets.

17.15 Solve Example 17.1 of bolt insertion sequence at 10 locations. Compare your solution to the one given in the example.

17.16 Solve the 16-bolt insertion sequence determination problem described in Huang, Hsieh and Arora (1997). Compare your solution to the one given in the reference.

17.17 The material for the spring in Exercise 17.7 must be selected from one of three possible materials given in Table E17.17 (refer to Section 15.8 for more discussion of the problem) (Huang and Arora, 1997). Obtain a solution to the problem.

17.18 The material for the spring in Exercise 17.8 must be selected from one of three possible materials given in Table E17.17 (refer to Section 15.8 for more discussion of the problem) (Huang and Arora, 1997). Obtain a solution to the problem.

Sections 17.2–17.4

17.19 Implement the DE algorithm into a computer program. Solve the Example 17.1 of bolt insertion sequence determination using your program. Compare performance of the DE and GA algorithms.

17.20 Implement the ACO algorithm into a computer program. Solve the Example 17.1 of bolt insertion sequence determination using your program. Compare performance of the ACO and GA algorithms.

TABLE E17.17 Material Data for the Spring Design Problem

Material Type	G (lb/in.^{2})	ρ (lb s^2/in.^{4})	τ_a (lb/in.^{2})	U_p
1	11.5×10^6	7.38342×10^{-4}	80,000	1.0
2	12.6×10^6	8.51211×10^{-4}	86,000	1.1
3	13.7×10^6	9.71362×10^{-4}	87,000	1.5

G = shear modulus; ρ = mass density; τ_a = shear stress; U_p = relative unit price.

17.21 Implement the PSO algorithm into a computer program. Solve the Example 17.1 of bolt insertion sequence determination using your program. Compare performance of the PSO and GA algorithms.

References

Arora, J.S., 2002. Methods for discrete variable structural optimization. In: Burns, S. (Ed.), Recent Advances in Optimal Structural Design. Structural Engineering Institute, Reston, VA, pp. 1–40.

Arora, J.S., Huang, M.W., 1996. Discrete structural optimization with commercially available sections: a review. J. Struct. Earthquake Eng. JSCE 13 (2), 93–110.

Arora, J.S., Huang, M.W., Hsieh, C.C., 1994. Methods for optimization of nonlinear problems with discrete variables: a review. Struct. Optim. 8 (2/3), 69–85.

Blum, C., 2005. Ant colony optimization: introduction and recent trends. Phys. Life Rev. 2, 353–373.

Chen, S.Y., Rajan, S.D., 2000. A robust genetic algorithm for structural optimization. Struct. Eng. Mech. 10, 313–336.

Coello-Coello, C.A., Van Veldhuizen, D.A., Lamont, G.B., 2002. Evolutionary Algorithms for Solving Multi-Objective Problems. Kluwer Academic, New York.

Corne, D., Dorigo, M., Glover, F. (Eds.), 1999. New Ideas in Optimization. McGraw-Hill, New York.

Das, S., Suganthan, N., 2011. Differential evolution: a survey of the state-of-the-art. IEEE Transac.Evolut. Comput. 15 (1), 4–31.

Dorigo, M., 1992. Optimization, learning and natural algorithms. Ph.D. Thesis, Politecnico di Milano, Italy.

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

Glover, F., Kochenberger, G. (Eds.), 2002. Handbook on Metaheuristics. Kluwer Academic, Norwell, MA.

Goldberg, D.E., 1989. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading, MA.

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

Huang, M.W., Arora, J.S., 1997a. Optimal design with discrete variables: some numerical experiments. Int. J. Nuer. Methods Eng. 40, 165–188.

Huang, M.W., Arora, J.S., 1997. Optimal design of steel structures using standard sections. Struct. Multidiscip. Optim. 14, 24–35.

Huang, M.W., Hsieh, C.C., Arora, J.S., 1997. A genetic algorithm for sequencing type problems in engineering design. Int. J. Numer. Methods Eng. 40, 3105–3115.

Kennedy, J., Eberhart, R.C., 1995. Particle swarm optimization. Proceedings of IEEE International Conference on Neural Network, vol. IV, IEEE Service Center, Piscataway, NJ, 1942–1948.

Kennedy, J., Eberhart, R.C., Shi, Y., 2001. Swarm Intelligence. Morgan Kaufmann, San Francisco.

Kocer, F.Y., Arora, J.S., 1996a. Design of prestressed concrete poles: an optimization approach. J. Struct. Eng. ASCE 122 (7), 804–814.

Kocer, F.Y., Arora, J.S., 1996b. Optimal design of steel transmission poles. J. Struct. Eng. ASCE 122 (11), 1347–1356.

Kocer, F.Y., Arora, J.S., 1997. Standardization of transmission pole design using discrete optimization methods. J. Struct. Eng. ASCE 123 (3), 345–349.

Kocer, F.Y., Arora, J.S., 1999. Optimal design of H-frame transmission poles subjected to earthquake loading. J. Struct. Eng. ASCE 125 (11), 1299–1308.

Kocer, F.Y., Arora, J.S., 2002. Optimal design of latticed towers subjected to earthquake loading. J. Struct. Eng. ASCE 128 (2), 197–204.

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

Nelder, J.A., Mead, R.A., 1965. A Simplex method for function minimization. Comput. J. 7, 308–313.

Osyczka, A., 2002. Evolutionary Algorithms for Single and Multicriteria Design Optimization. Physica Verlag, Berlin.

Pezeshk, S., Camp, C.V., 2002. State-of-the-art on use of genetic algorithms in design of steel structures. In: Burns, S. (Ed.), Recent Advances in Optimal Structural Design. Structural Engineering Institute, ASCE, Reston, VA.

Price, K., Storn, R., Lampinen, J., 2005. Differential Evolution—A Practical Approach to Global Optimization. Springer, Berlin.

Qing, A., 2009. Differential Evolution—Fundamentals and Applications in Electrical Engineering. Wiley-Interscience, New York.