10

GLOBAL OPTIMIZATION FOR PROBLEMS WITH CONTINUOUS AND DISCRETE VARIABLES

IN CHAPTERS 6 AND 8 we showed that in continuous variable minimization problems with convex feasible regions and convex objectives, any local minimum is the global minimum. As discussed in Section 8.2, many problems do not satisfy these convexity conditions, and it is often difficult to verify whether they satisfy them or not. Models that include nonlinear equality constraints fall in this latter category. These constraints arise from nonlinear material balances (for which both flows and concentrations are unknowns), nonlinear physical property relations, nonlinear blending equations, nonlinear process models, and so on. Another source of nonconvexity can be in the objective function if it is concave, which can occur when production costs increase with the amount produced, but at a decreasing rate due to economies of scale. A problem involved in minimizing a concave objective function over a convex region defined by linear constraints is that it may have many local minima, one at each extreme point of the region. Nonconvex objective functions and local minima can also occur when you estimate the values of the model parameters using the least-squares or maximum likelihood objective functions.

Any problem containing discretely valued variables is nonconvex, and such problems may also be solved by the methods described in this chapter. The search methods discussed in Section 10.5 are often applied to supply chain and production-sequencing problems.

10.1 METHODS FOR GLOBAL OPTIMIZATION

If an NLP algorithm such as SLP, SQP, or GRG described in Chapter 8 is applied to a smooth nonconvex problem, it usually converges to the "nearest" local minimum, which may not be the global mimimum. We refer to such algorithms in this chapter as "local solvers." The problem of finding a global minimum is much more difficult than that of finding a local one, but several well-established general-purpose approaches to the problem are discussed subsequently.

Figure 10.1 shows a classification of global optimization methods. Exact methods, if allowed to run until they meet their termination criteria, are guaranteed to find an arbitrarily close approximation to a global optimum and to verify that they have done so. These include branch-and-bound (BB) methods, which were discussed in the context of mixed-integer linear and nonlinear programming in Chapter 9, methods based on interval arithmetic (Kearfott, 1996), and some multistart procedures, which invoke a local solver from multiple starting points. Heuristic search methods may and often do find global optimal solutions, but they are not guaranteed to do so, and we are usually unable to prove that they have found a global solution even when they have done so. Nonetheless, they are widely used, often find very good solutions, and can be applied to both mixed-integer and combinatorial problems. A heuristic search method starts with some current solution, explores all solutions in some neighborhood of that point looking for a better one, and repeats if an improved point is found. Metaheuristics algorithms guide and improve on a heuristic algorithm. These include tabu search, scatter search, simulated annealing, and genetic algorithms. They use a heuristic procedure for the

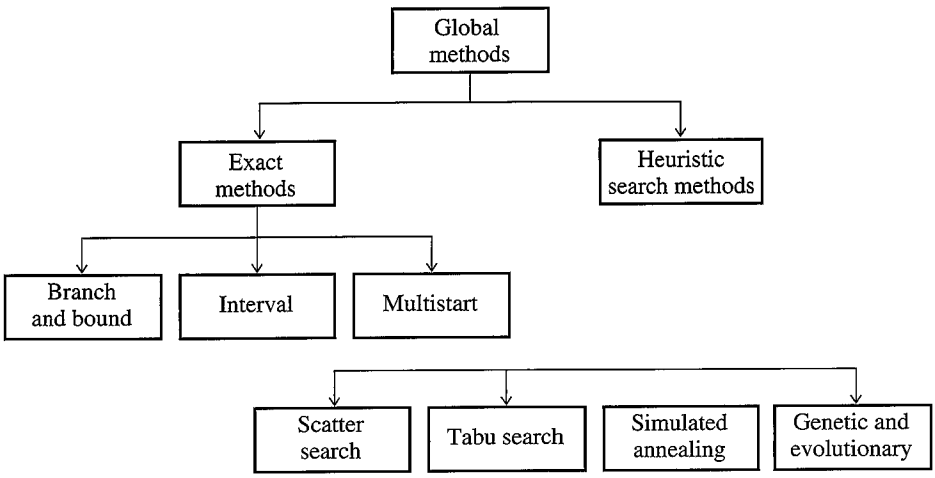

FIGURE 10.1
Classification of global optimization methods.

problem class, which by itself may not be able to find a global optimum, and guide the procedure by changing its logic-based search so that the method does not become trapped in a local optimum. Genetic and evolutionary algorithms use heuristics that mimic the biological processes of crossover and mutation. They are "population-based" methods that combine a set of solutions (the "population") in an effort to find improved solutions and then update the population when a better solution is found. Scatter search is also a population-based procedure.

The methods mentioned earlier are general-purpose procedures, applicable to almost any problem. Many specialized global optimization procedures exist for specific classes of nonconvex problems. See Pinter (1996a) for a brief review and further references. Typical problems are

- Problems with concave objective functions to be minimized over a convex set.
- "Differential convex" (DC) problems of the form

$$\begin{aligned} &\text{Minimize:} && f(\mathbf{x}) \\ &\text{Subject to:} && g_j(\mathbf{x}) \leq \mathbf{0}, \quad j = 1,2, \ldots, J \\ &\text{and} && \mathbf{x} \in C \end{aligned}$$

where C is a convex set, and f and each constraint function g_j can be expressed as the difference of two convex functions, such as $f(x) = p(x) - q(x)$.

- Indefinite quadratic programs, in which the constraints are linear and the objective function is a quadratic function that is neither convex nor concave because its Hessian matrix is indefinite.
- Fractional programming problems, where the objective is a ratio of two functions.

If a problem has one of these forms, the special-purpose solution methods designed for it often produce better results than a general-purpose approach. In this

chapter, we focus on general-purpose methods and on frameworks that are applicable to wide classes of problems and do not discuss special problem classes further.

10.2 SMOOTHING OPTIMIZATION PROBLEMS

All gradient-based NLP solvers, including those described in Chapter 8, are designed for use on problems in which the objective and constraint functions have continuous first partial derivatives everywhere. Examples of functions that do not have continuous first partials everywhere are

1. $|f(x)|$
2. $\max(f(x),g(x))$
3. $h(x) = \{\text{if } x_1 \leq 0 \text{ then } f(x) \text{ else } g(x)\}$
4. A piecewise linear function interpolating a given set of (y_i, x_i) values.

If you encounter these functions, you can reformulate them as equivalent smooth functions by introducing additional constraints and variables. For example, consider the problem of fitting a model to n data points by minimizing the sum of weighted absolute errors between the measured and model outputs. This can be formulated as follows:

$$\text{Minimize: } e(\mathbf{x}) = \sum_{i=1}^{n} w_i |y_i - h(v_i, \mathbf{x})|$$

where $\mathbf{x}$ = a vector of model parameter values
w_i = a positive weight for the error at the ith data point
y_i = the measured output of the system being modeled when the vector of system inputs is v_i
$h(v_i, \mathbf{x})$ = the calculated model output when the system inputs are v_i

This weighted sum of absolute values in $e(\mathbf{x})$ was also discussed in Section 8.4 as a way of measuring constraint violations in an exact penalty function. We proceed as we did in that section, eliminating the nonsmooth absolute value function by introducing positive and negative deviation variables dp_i and dn_i and converting this nonsmooth unconstrained problem into an equivalent smooth constrained problem, which is

$$\text{Minimize: } \sum_{i=1}^{n} w_i(dp_i + dn_i) \tag{10.1}$$

$$\text{Subject to: } y_i - h(v_i, \mathbf{x}) = dp_i - dn_i \qquad i = 1, \ldots, n \tag{10.2}$$

and

$$dp_i \geq 0, \qquad dn_i \geq 0 \qquad i = 1, \ldots, n \tag{10.3}$$

In this problem, if the error is positive, then dp_i is positive and dn_i is zero in any optimal solution. For negative errors, dp_i is zero and dn_i is positive. The absolute

error is thus the sum of these deviation variables. A similar reformulation allows the problem of minimizing the maximum error to be posed as a smooth constrained problem.

If it is difficult or impossible to eliminate the nonsmooth functions by these or some other transformations, you can apply a gradient-based optimizer and hope that a nonsmooth point is never encountered. If one is encountered, the algorithm may fail to make further progress because the computed derivatives at the point are not meaningful. A large body of literature on methods for nonsmooth optimization exists (see Hiriart-Urruty and LeMarechal, 1993, for example), but software for nonsmooth optimization is not yet widely available. Alternatively, you can apply an optimization method that does not require first partial derivatives. Such algorithms include the Nelder–Meade simplex method (not the same as the simplex method for linear programming), the Hooke–Jeeves procedure, or a conjugate directions method due to Powell that does not use derivatives (Avriel, 1976). These techniques are not as sensitive to derivative discontinuities as gradient-based algorithms, but continually improve the objective function until they reach an approximation to a local minimum. They are not guaranteed to converge to a local solution for nonsmooth problems, and are basically unconstrained methods. You can incorporate constraints by using penalty functions, but if a large penalty weight is used, the objective function becomes ill-conditioned and hard to optimize with high accuracy. The search methods described in Section 10.5 are not as sensitive to discontinuities and are much less likely than a local solver to be trapped near a local optimum.

10.3 BRANCH-AND-BOUND METHODS

We have already discussed branch-and-bound methods in Sections 9.2 and 9.3 in the context of mixed-integer linear and nonlinear programming. The "divide-and-conquer" principles underlying BB can also be applied to global optimization without discrete variables. The approximation procedure for a function of one variable is shown in Figure 10.2. The nonconvex function f has three local minima over the interval [0, 2]. The convex underestimator function $f_0(x)$ is defined over the entire interval. The underestimating functions $f_1(x)$ and $f_2(x)$ are defined over the two subintervals, and are "tighter" underestimates than f_0. We will discuss procedures for constructing such functions shortly. Because each underestimator is convex, minimizing it using any convergent local solver leads to its global minimum. Let x_i^* minimize $f_i(x)$ over its associated interval, as shown in Figure 10.2. Then $f_i(x_i^*)$ is a lower bound on the global minimum over that interval, and $f(x_i^*)$ is an upper bound over the entire interval. These bounds are used to fathom nodes in the BB tree, in the same way as LP relaxations were used in Chapter 9.

To illustrate, consider a minimization problem involving two variables with upper and lower bounds:

$$\text{Minimize:} \quad f(\mathbf{x})$$

$$\text{Subject to:} \quad 0 \leq x_i \leq 2, \quad i = 1,2$$

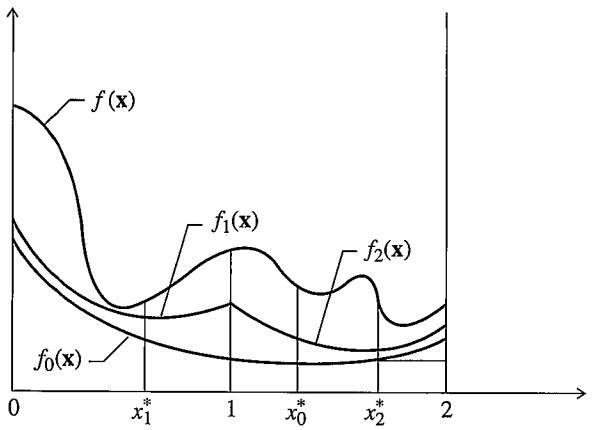

FIGURE 10.2
Convex underestimator of a nonconvex function.

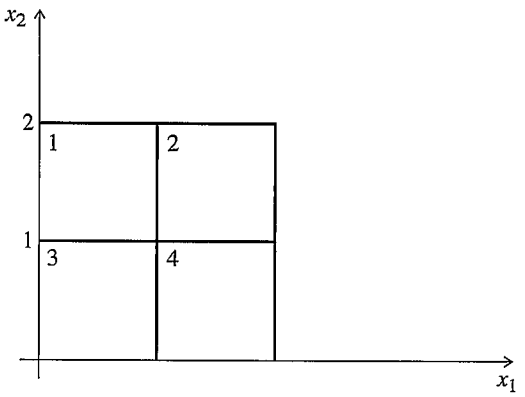

FIGURE 10.3
Branch-and-bound partitions.

where $\mathbf{x} = (x_1, x_2)$ and f is a nonconvex function having several local minima within the rectangle defined by the bounds. Let the initial partition be composed of four smaller rectangles, as shown in Figure 10.3.

Figure 10.4 shows a BB tree, with the root node corresponding to the original rectangle, and each node on the second level associated with one of these four partitions. Let $f_i(x)$ be the underestimating function for the partition associated with node i. The lower bounds shown next to each node are illustrative and are derived by minimizing $f_i(x)$ over its partition using any local solver, and the upper bounds

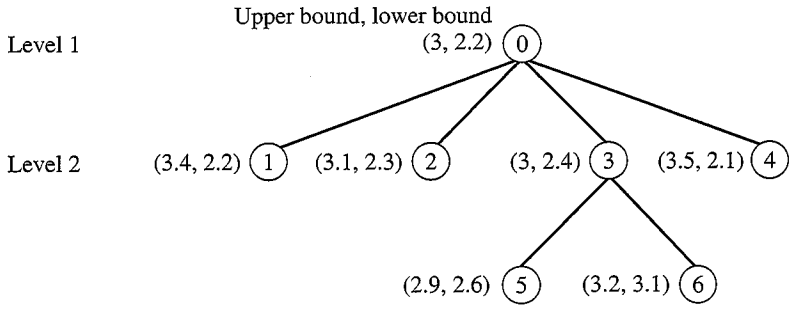

FIGURE 10.4
Branch-and-bound tree.

are the f values at the points that minimize f_i. Because the f_i functions for each partition are better estimates for f over their rectangles than f_0 is, the lower bounds over each partition are generally larger than the lower bound at the root node, as shown in Figure 10.4. The upper bounds need not improve at each level, but the f and $\mathbf{x}$ values associated with the smallest upper bound found thus far are retained as the "incumbent," the best point found thus far. The best f value at level two is 3.0.

The iterative step in this BB procedure consists of choosing a node to branch on and performing the branching step. There are several rules for choosing this node. A popular rule is to select the node with the smallest upper bound, on the assumption (possibly incorrect) that it leads to better f values sooner. This is node 3 in Figure 10.4. This node's rectangle is partitioned into two (or possibly more) subsets, leading to nodes 5 and 6, and the convex subproblems at each node are solved, yielding the upper and lower bounds shown. Because the lower bound at node 6 (3.1) exceeds the incumbent value of 3.0, an f value lower than 3.0 cannot be found by further branching from node 6. Hence this node has been fathomed. The procedure stops when the difference between the incumbent f value and the lower bound at each unfathomed node is smaller than a user-defined tolerance.

If each underestimating function is "tighter" than that at the node immediately above in the tree, the BB procedure eventually terminates. Floudas (2000a) suggests procedures for constructing these underestimating functions, which apply to specific commonly occurring nonconvex functions, such as the bilinear function xy, quotients x/y, concave functions of a single variable, and so on. For a general function $f(\mathbf{x})$ of n variables, defined over a rectangle $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$, a convex underestimator is

$$L(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{n} \alpha_i (l_i - x_i)(u_i - x_i) \tag{10.4}$$

where the α_i's are positive scalars. Because the summation in Equation (10.4) is nonpositive, $L(x) \leq f(x)$, so L is an underestimator of f. The summation is a positive linear combination of convex quadratic functions, so this term is convex. For L to

be convex, the scalars α_i must be sufficiently large, as can easily be seen by considering the Hessian matrix of L:

$$\nabla^2 L(\mathbf{x}) = \nabla^2 f(\mathbf{x}) + \mathbf{D} \tag{10.5}$$

where $\mathbf{D}$ is a positive diagonal matrix with diagonal elements $2\alpha_i$. If these elements are sufficiently large, the Hessian of L is positive-definite for all x in the domain of f, which implies that L is convex over that domain. Tests to determine how large the α_i's need to be are in Floudas (2000a, b), and other references cited there. Floudas has also shown that the maximum difference between the approximating function L and the actual function f is

$$d_{\max} = \frac{1}{4} \sum_{i=1}^{n} \alpha_i (u_i - l_i)^2 \tag{10.6}$$

As rectangles are partitioned, the difference $(u_i - l_i)$ decreases, so the successive underestimating functions become tighter approximations to f.

10.4 MULTISTART METHODS

Because software to find local solutions of NLP problems has become so efficient and widely available, *multistart methods*, which attempt to find a global optimum by starting the search from many starting points, have also become more effective. As discussed briefly in Section 8.10, using different starting points is a common and easy way to explore the possibility of local optima. This section considers multistart methods for unconstrained problems without discrete variables that use randomly chosen starting points, as described in Rinnooy Kan and Timmer (1987, 1989) and more recently in Locatelli and Schoen (1999). We consider only unconstrained problems, but constraints can be incorporated by including them in a penalty function (see Section 8.4).

Consider the unconstrained global minimization of a smooth function of n variables $f(\mathbf{x})$. We assume that upper and lower bounds can be defined for each variable, so that all local minima lie strictly inside the rectangle R formed by the bounds. Let L denote the local optimization procedure to be used. L is assumed to operate as follows: Given any starting point, $\mathbf{x}_0$ in R, L converges to a local minimum of f that depends on $\mathbf{x}_0$ and is "closest" to $\mathbf{x}_0$ in a loose sense. If L is started from each of N randomly generated starting points, which are uniformly distributed in R, the probability that the best local optimum found in these N trials is global approaches one as N approaches infinity (Rinnooy Kan and Timmer, 1989). In fact, L is not even necessary for this asymptotic result to hold, because the best function value over all the starting points converges to the globally optimal value as $N \to \infty$ with probability 1, but this search usually converges much more slowly.

Because each local optimum may be found many times, this multistart procedure is inefficient. If $\mathbf{x}_i^*$ denotes the ith local optimum, we define the *region of*

attraction of $\mathbf{x}_i^*$, R_i, to be the set of all starting points in R from which L converge to $\mathbf{x}_i^*$. The goal of an efficient multistart method is to start L *exactly once* in each region of attraction.

Rinnooy Kan and Timmer (1987, 1989) developed an efficient multistart procedure called multilevel single linkage (MLSL), based on a simple rule. A uniformly distributed sample of N points in R is generated, and the objective function f is evaluated at each point. The points are sorted according to their f values, and the pN best points are retained, for which p is an algorithm parameter between 0 and 1. L is started from each point of this reduced sample, except if another sample point with a lower f value exists within a certain critical distance. L is also not started from sample points that are too near the boundary of R or too close to a previously discovered local minimum. Then, N additional uniformly distributed points are generated, and the procedure is applied to the union of these points and those retained from previous iterations. The critical distance decreases each time a new set of sample points is added. The authors show that, even if the sampling continues indefinitely, the total number of local searches ever initiated by MLSL is finite with a probability of 1, and each local minimum of f is located with a probability of 1. They also developed Bayesian stopping rules, which incorporate assumptions about the costs and potential benefits of further function evaluations, to determine when to stop the procedure.

10.5 HEURISTIC SEARCH METHODS

Chapter 9 describes several types of problems that require the use of integer-valued variables and discusses two solution approaches for such problems: branch-and-bound (BB) and outer approximation (OA). These methods can guarantee a global solution under certain conditions, but the computational effort required increases rapidly with the number of integer variables. In addition, BB is guaranteed to find a global optimum only if the global optimum of each relaxed subproblem is found. As discussed in Sections 10.3 and 10.4, this may be very hard to do if the subproblems are not convex. OA also requires convexity assumptions to guarantee a global solution (see Section 9.4). Hence, there is a need for alternative methods that are not guaranteed to find an optimal solution, but often find good solutions more rapidly than BB or OA. We describe such methods, called heuristic search procedures, in this section. They include genetic algorithms (or, more generally, evolutionary algorithms), simulated annealing, tabu search, and scatter search.

Heuristic search procedures can be applied to certain types of combinatorial problems when BB and OA are difficult to apply or converge too slowly. In these problems, it is difficult or impossible to model the problem in terms of a vector of decision variables, which must satisfy bounds on a set of constraint functions, as required by OA. One example is the "traveling salesman" problem, in which the feasible region is the set of all "tours" in a graph, that is, closed cycles or paths that visit every node only once. The problem is to find a tour of minimal distance or cost,

which is to be used by a vehicle that is routed to several stops. The traveling salesman problem is the simplest type of vehicle-routing problem, with a single vehicle leaving from a single starting point. Multivehicle, multistarting point problems can have constraints such as time windows within which stops must be visited, vehicle capacities, restrictions on which vehicles can visit which stops, and so on (Crainic and Laporte, 1998). In all cases, the problem is to find a set of routes and an assignment of vehicles to routes, that visit all stops and meet all the constraints.

Another important class of combinatorial problems is "job-shop" scheduling, in which you seek an optimal sequence or order in which to process a set of jobs on one or more machines. Such problems are often encountered in chemical engineering when sequencing a set of products through a batch process, in which set-up times and costs must be incurred for each unit operation before a product can be produced, and these times depend on the product previously produced. If there is a single machine, and the products are numbered from 1 to n, then the feasible region is the set of all permutations of the positive integers 1, 2, ..., n, corresponding to the order in which the jobs are processed. This is equivalent to a traveling salesman problem in which each node in a graph corresponds to a job, and the travel times between nodes are the set-up times between jobs.

These combinatorial problems, and many others as well, have a finite number of feasible solutions, a number that increases rapidly with problem size. In a job-shop scheduling problem, the size is measured by the number of jobs. In a traveling salesman problem, it is measured by the number of arcs or nodes in the graph. For a particular problem type and size, each distinct set of problem data defines an *instance* of the problem. In a traveling salesman problem, the data are the travel times between cities. In a job sequencing problem the data are the processing and set-up times, the due dates, and the penalty costs.

One measure of the efficiency of an algorithm designed to solve a class of combinatorial problems is an upper bound on the time required to solve any problem instance of a given size, and this time increases with size. Time is often measured by the number of arithmetic operations or constraint and objective function evaluations to find a solution. If, for a given algorithm and problem class, it can be shown that the time required for the algorithm to solve any instance of the problem is bounded by a polynomial in the problem size parameter(s), then that algorithm is said to solve the problem class in *polynomial time*. Some combinatorial problems, for example, sorting a list, are solvable in polynomial time. For many combinational problems, however, no known algorithm can solve all instances in polynomial time. Such problems are called *NP-hard*. Although methods to find optimal solutions have been devised that avoid complete enumeration of all solutions (often based on branch-and-bound concepts), none of them can guarantee a solution in polynomial time. Hence, heuristic and metaheuristic search methods, which cannot guarantee an optimal solution but often find a good (or optimal) one quickly, are now widely used.

10.5.1 Heuristic Search

Consider the problem: Minimize $f(\mathbf{x})$ subject to $\mathbf{x} \in X$, where $\mathbf{x}$ represents the variables or other entities over which we are optimizing f. The objective function f may

TABLE 10.1
Data for sequencing problem

Job	Processing time (days)	Due date (day)	Tardiness penalty ($/day)
1	2	5	1
2	6	7	2
3	4	9	4

TABLE 10.2A
Objective function computation for the sequence (3, 1, 2)

Job	Completion time (days)	Tardiness (days)	Delay cost ($)
1	$4 + 2 = 6$	$6 - 5 = 1$	$1 * 1 = 1$
2	$6 + 6 = 12$	$12 - 7 = 5$	$5 * 2 = 10$
3	4	Max $(4 - 9, 0) = 0$	$0 * 4 = 0$

TABLE 10.2B
Swap neighborhood of (3, 1, 2)

i	j	New permutation	Move value
1	2	(1, 3, 2)	$10 - 11 = -1$
1	3	(2, 1, 3)	$15 - 11 = 4$
2	3	(3, 2, 1)	$13 - 11 = 2$

be linear or nonlinear, and X is defined by the constraints of the problem. $\mathbf{x}$ may be a cycle in a graph, a permutation (representing a sequence in which to process jobs on a machine), or, in the simplest case, a vector of n decision variables. The constraints may be bounds on functions of $\mathbf{x}$, or they may include verbal logic-based statements or conditions like "x is a tree in this graph that connects all nodes" or "if–then" statements.

As an example, consider a problem of sequencing three jobs on a single machine to minimize the sum of weighted "tardiness" for all jobs, where tardiness is defined as the difference between the completion time of a job and its due date if this difference is positive, and zero otherwise. Job processing times, due dates, and delay penalties for an instance of this problem are shown in Table 10.1.

To show how the objective function is computed, consider the sequence $\mathbf{x} = (3, 1, 2)$. The job completion times, tardiness values, and delay costs for this sequence are shown in Table 10.2A.

The objective value for this sequence is the sum of the costs in the "delay cost" column:

$$f((3, 1, 2)) = 11$$

TABLE 10.3
Descent method using the search neighborhood $N(\mathbf{x})$

1. Start with $\mathbf{x} \in X$
2. Find $\mathbf{x}' \in N(x)$ such that $f(\mathbf{x}') < f(\mathbf{x})$.
3. If no such $\mathbf{x}'$ exists, stop and return $\mathbf{x}$.
4. Otherwise replace $\mathbf{x}$ by $\mathbf{x}'$ and return to step 2.

In neighborhood-based heuristic searches, each $\mathbf{x} \in X$ has an associated neighborhood $N(\mathbf{x})$ that contains all the feasible solutions that the search will explore when the current point is $\mathbf{x}$. Each alternative solution $\mathbf{x}' \in N(\mathbf{x})$ is reached from $\mathbf{x}$ by an operation called a move. Consider again the three-job problem based on Table 10.2A. Let the current sequence x be (3, 1, 2), and suppose that we consider only neighboring permutations $\mathbf{x}'$ that can be reached from $\mathbf{x}$ by swapping a pair of jobs in $\mathbf{x}$. This "swap neighborhood" is shown in Table 10.2B, in which i and j are the indices of the jobs to be swapped. If there are n jobs, then a swap neighborhood contains $n(n-1)/2$ permutations.

The "move value" column in Table 10.2B contains the change in objective value realized by making the move, $f(\mathbf{x}') - f(\mathbf{x})$. Because $\mathbf{x} = (3, 1, 2)$, we determined earlier that $f(\mathbf{x}) = 11$. If the objectives for the new permutations shown in Table 10.2B are evaluated, move values can be obtained. By moving to permutation (1, 3, 2), we improve the objective by one unit. Then the same procedure can be applied at this new point.

This straightforward descent method can be generalized for discrete-variable problems as shown in Table 10.3 (Glover and Laguna, 1997, Chapter 2). This algorithm is similar to the algorithms for linear programs and continuous-variable nonlinear programs discussed in Chapters 6–8, where step 2 was conducted by choosing a search direction and performing a line search along that direction. The variation of this algorithm that seeks the $\mathbf{x}' \in N(\mathbf{x})$ with lowest f value is called steepest descent (see Chapter 6). Although this simple descent method solves some combinatorial problems from any starting point, for many important problems (routing and sequencing included) it usually stops at a nonoptimal point, which is often far from optimal. Such a point is called a local solution relative to the neighborhood $N(\mathbf{x})$. As noted by Glover and Laguna (1997), descent methods by themselves have had very limited success in solving hard combinatorial optimization problems, but they provide an underlying heuristic for a *metaheuristic* procedure to guide the search. The resulting metaheuristic algorithms have been widely and successfully used.

In fact, most metaheuristics do not require a preexisting heuristic. They simply require a way to define a neighborhood of any current solution, which contains alternative solutions as possible moves. For example, tabu search, which is discussed in the following section, includes strategies for operating directly with such neighborhoods. Some neighborhood structures allow solutions to be built up one

element at a time in a constructive way. For example, a spanning tree in a network may be constructed one arc at a time, each time choosing a new arc that creates no loops, connects a new node, and has the greatest value or least cost.

10.5.2 Tabu Search

Tabu search (TS) is widely used by operations research analysts, but has received little attention from chemical engineers, even though it can be used to solve many important and difficult real-world problems. These include problems of the following types: planning and scheduling, telecommunications and multiprocessor computing systems, transportation networks and vehicle routing, operation and design of manufacturing systems, and financial analysis. An excellent survey of these applications and pertinent references is found in Glover and Laguna (1997).

As discussed in the previous section, descent heuristics fail to solve many problems because they get trapped in local minima (relative to the type of neighborhoods they use). That is, they stop at the first solution encountered where no neighboring solution is better. TS, and in fact any metaheuristic search method, overcomes this limitation by allowing nonimproving moves. The term *tabu* refers to TS's definition of certain moves as forbidden. These are usually specified as moves to solutions with particular attributes, as illustrated in the following example. The tabu moves are specified so as to keep previously performed moves from being reversed or to prevent already visited solutions from being revisited. These and other mechanisms force the search process to move beyond the nearest local minimum and to explore regions where improved solutions may lie.

We explain the ideas behind TS using a problem from Barnes and Vanston (1981) of sequencing five different product batches through a single-batch process. Each batch has a processing time and a delay penalty cost, as shown in Table 10.4. The penalty is charged for any delay in starting production beyond time zero; setup costs must also be taken into account.

It is reasonable to schedule jobs with short processing times and high penalty costs first. This is motivation for a heuristic that computes the ratio of processing time to penalty cost (see column four of Table 10.4) and sequences the batches in order of increasing value of this ratio, which is the order given in the table. However,

TABLE 10.4
Batch processing times and delay penalties

Batch	Processing time (h)	Delay penalty (100$/h)	Ratio
1	3	7	3/700 = 4.28E-3
2	4	8	4/800 = 5.0E-3
3	1	1	1/100 = 1.0E-2
4	4	3	4/300 = 1.33E-2
5	5	2	5/200 = 2.5E-2

TABLE 10.5
Batch set-up costs

		$j \rightarrow$					
i		**1**	**2**	**3**	**4**	**5**	**6**
↓	**0**	11	6	12	20	14	
	1		13	7	12	11	10
	2	9		11	13	6	12
	3	9	10		20	7	15
	4	10	7	8		6	12
	5	14	13	12	13		9

TABLE 10.6
Calculated completion times

Batch	Completion time
1	3
2	4 + 4 = 8
3	3 + 1 = 4
4	13 + 4 = 17
5	8 + 5 = 13

this ignores the fact that, if batch i was last produced, and batch j is next, there is a set-up cost of $s_{i,j}$ dollars before batch j can begin, representing the time and expense associated with cleaning up after batch i and preparing the process to produce batch j. These set-up costs are shown in Table 10.5. The table includes fictitious batches 0 and 6 (always sequenced first and last, respectively, and with zero processing times and delay penalties), whose set-up costs represent the cost of starting up the first batch and cleaning up after the last one.

Let

$$P = (p(1), p(2), \ldots, p(5))$$

be a permutation of the integers 1 through 5, representing a sequence for producing the batches, where $p(i)$ is the index of the job in position i. If $P = (1, 3, 2, 5, 4)$, then the completion times of the jobs are as shown in Table 10.6.

The corresponding objective value $obj(P)$ is computed as follows:

$$\text{Objective}(P) = \text{Delay cost}(P) + \text{Setup cost}(P)$$

$$\text{Delay cost}(P) = 700(3) + 800(8) + 100(4) + 300(17) + 200(13) = 16{,}600$$

$$\text{Setup cost}(P) = 1100 + 700 + 1000 + 600 + 1300 + 1200 = 5900$$

$$\text{Objective}(P) = 22{,}500$$

A TS algorithm for this problem described in Laguna, et al. (1991) modifies the swap heuristic as follows:

- At each iteration, certain moves are forbidden or *tabu.*
- One or more move *attributes* are chosen, and the tabu moves are those whose attribute(s) satisfy the specified tabu conditions.
- A *short-term memory function* determines how long a tabu restriction remains active. This can be expressed as the number of iterations a tabu condition is enforced once it is imposed.
- The tabu status of a move can be overridden if the objective value after the move is better than a specified threshold, called an *aspiration level.*
- A *long-term memory function* determines when to restart the entire procedure and what the new starting point should be. These new starting points are chosen to be in regions of the search space (i.e., the space of all permutations) that have not been previously explored. This *diversifies* the search. Long-term memory can diversify the search in ways other than by direct restarting (Glover and Laguna, 1997) and can also *intensify* the search by inducing it to explore attractive areas more thoroughly.

The purpose of a tabu restriction is to prevent a move from being reversed during the length of the short-term memory, which is a number of future moves specified by the variable *tabu_size*. If, at a given iteration, jobs $p(i)$ and $p(j)$ are swapped, then any move that places job $p(i)$ *earlier* in the sequence than position i is tabu, until *tabu_size* iterations have occurred or the aspiration level is exceeded. To keep track of which moves are tabu and to free those moves from their tabu status, Laguna et al. define the following data structures

- tabu_list $(k) = p(i)$ if job $p(i)$ is prevented from moving to the left of its tabu position at iteration k. This is a circular list of length tabu_size.
- tabu_position $p(i)$ = tabu position for job $p(i)$.
- tabu_state $(p(i))$ = number of times job $p(i)$ appears on the tabu list.
- aspiration_level $(p(i))$ = aspiration level for job $p(i)$.

The aspiration level allows a tabu move of a job $p(j)$ to an earlier position if

$$\text{Current objective value} + \text{move\_value} < \text{aspiration level for job } p(j)$$

The aspiration level for a job is initialized to a large value and updated as follows. Let P be the current sequence and assume that the move of jobs $p(i)$ and $p(j)$ has the best move_value.

- If aspiration level $(p(j))$ > objective(P), then aspiration level $(p(i))$ = *objective*(P)
- If aspiration level $(p(j))$ > objective(P) + move_value, then aspiration level $(p(j))$ = objective(P) + move_value.

This prevents the immediate reversal of a nonimproving move (one with a positive move_value) in the next iteration. The reversal of this move now has a negative move_value, but it is classified as tabu, and the previous update does not allow it to satisfy the aspiration criterion.

Begin
- Initialize long term memory function
- Best_obj = large value
- **Do while** (Best_obj has changed in the last max_moves_long starting points) **Begin1**
  Generate starting solution P, and set Best_solution = P
  Evaluate $obj(P)$
  Initialize move_value_matrix
  Initialize short term memory function
  Do while (moves without improvement < max_moves) **Begin2**
- Update long term memory function
- Best_move value = large value
- **For** (all candidate moves) **Begin3**
- **If** (candidate move is admissable) **Begin4**
- **If** (move_value<best_move value) **Begin5**
- Best_move value = move_value
- Best_move = current_move **End5**

End4
End3
- Execute best_move
- Update objective value: $obj(P) = obj(P) + best\_move\_value$
- Update move_value_matrix
- Update short term memory function
- **If** $obj(P)$<Best_obj, then **Begin6**
- Best_obj = $obj(P)$
- Best_solution = P **End6**

End2
End1

FIGURE 10.5
Tabu search procedure for batch sequencing.

Figure 10.5 shows the TS procedure in pseudo-code. This entire procedure is executed until max_moves_long successive restarts fail to improve the best objective value. Given a starting solution, the inner **do** loop is executed until there are max_moves successive moves without improvement in the best solution found in the current "pass," that is, using the current starting solution. A move is considered a *candidate* if the jobs being swapped are within a specified "distance" (number of positions) of one another. This limitation allows search time to be limited, but a complete search can be done by making this distance equal to the number of batches. A candidate move is admissible if either it is not tabu or it is tabu but its tabu status is overridden by the aspiration criterion.

The long-term memory function uses the matrix called "move_value_matrix" in Figure 10.5, whose (i, j) element is the number of times that job i has been scheduled in position j. This matrix is updated after every move by adding 1 to the (i, j) element if $p(i) = j$ in the current sequence. Then, the fraction of time each job has spent in each position can be calculated by dividing these matrix elements by the

total number of moves so far. Penalty costs proportional to these time fractions are defined and are used in the heuristic that generates starting solutions to force it to choose diverse starting points. This one-pass heuristic starts by scheduling batch 0 in position 0. Then, the unsequenced job j that minimizes the "distance" from the previously selected job, say job i, is scheduled next. The "distance" is the set-up cost between jobs i and j plus a multiple of the ratio of the delay penalty for job j divided by the largest delay penalty for all unsequenced jobs. If this heuristic is being used to restart the algorithm, a multiple of the fraction of time that job j has occupied the current position is added to the "distance," biasing it to choose different positions for the jobs from those they occupied frequently thus far. Such diversification strategies are important elements of intelligent search procedures.

The performance of this TS algorithm on the five-batch problem described earlier is shown in Table 10.7, using the following TS parameter values:

- Maximum moves without improvement = max_moves = 2.
- Maximum number of positions between swapped jobs = 1.
- Length of short-term memory = tabu_size = 3.
- Maximum restarts without improvement = max_moves_long = 4.

At iteration 1, the best move interchanges jobs 1 and 3, with a move value of –1000. This leads to the new sequence in row 2. In row 1, because job 3 was moved to the right, it is added to tabu_list in its first position, tabu_state (3) is set to 1 because job 3 appears once on tabu_list, and tabu_position (3) is set to 1, the original position of job 3. Moves that swap job 3 back to position 1 are henceforth tabu, unless they satisfy the aspiration criterion. At iteration 2, the best available move is to swap jobs 2 and 3, so job 3 is again added to tabu_list, its tabu_state is increased to 2, and its tabu_position entry is changed to 2. Iterations 2 and 3 fail to improve Best_obj, so the inner loop is restarted at iteration 4. Note that the current schedule in row 4 is quite different from those in earlier rows, due to the long-term memory function. The schedule in row 5 is optimal, but there is no way to prove its optimality, so the search must continue. It is restarted at iterations 7, 9, and 11, and the procedure stops at iteration 12 due to the limit of four successive restarts without improvement. A linear mixed-integer programming formulation of a similar production sequencing problem is described in Chapter 16.

Unfortunately, no general-purpose TS software is commercially available. Thousands of TS implementations have been made over the last 15 years (Glover and Laguna, 1997), but all address specific classes of problems, such as the job sequencing problem discussed earlier. Many of these implementations have been extremely successful, because the flexibility of TS allows an experienced analyst to incorporate his or her knowledge of the problem into the algorithm in many ways. Specific knowledge can include selecting the neighborhood that defines the possible next solutions, the short- and long-term memory structures, and the attributes that determine which solutions are tabu, among other things.

In closing this section, we emphasize that the adaptive memory structures used in TS encompass a variety of elements not treated in this simple example. Further details can be found in Glover and Laguna (1997).

TABLE 10.7

Performance of tabu search on a five-batch sequencing problem

Iteration	Current schedule	Current objective	Best move	Move value	Tabu state	Tabu list	Tabu position	Best objective
0*	(3,1,2,4,5)	14900			(0,0,0,0,0,3)	(6,6,6)	(0,0,0,0,0)	14900
1	(3,1,2,4,5)	14900	(3,1)	–1000	(0,0,1,0,0,2)	(3,6,6)	(0,0,1,0,0)	
2	(1,3,2,4,5)	13900	(3,2)	1000	(0,0,2,0,0,1)	(3,3,6)	(0,0,2,0,0)	13900
3	(1,2,3,4,5)	14900	(1,2)	–900	(1,0,2,0,0,0)	(3,3,1)	(1,0,2,0,0)	
4*	(2,4,1,3,5)	15500	(4,1)	–2000	(0,0,0,1,0,2)	(4,6,6)	(0,0,0,2,0)	
5	(2,1,4,3,5)	13500	(4,3)	500	(0,0,0,2,0,1)	(4,4,6)	(0,0,0,3,0)	13500
6	(2,1,3,4,5)	14000	(1,3)	0	(1,0,0,2,0,0)	(4,4,1)	(2,0,0,3,0)	
7*	(3,2,1,5,4)	16500	(5,4)	–1600	(0,0,0,0,1,2)	(5,6,6)	(0,0,0,0,4)	
8	(3,2,1,4,5)	14900	(3,2)	–900	(0,0,1,0,1,1)	(5,3,6)	(0,0,1,0,4)	
9*	(3,1,2,5,4)	15900	(3,1)	–1000	(0,0,1,0,0,2)	(3,6,6)	(0,0,1,0,0)	
10	(1,3,2,5,4)	14900	(5,4)	–1000	(0,0,1,0,1,1)	(3,5,6)	(0,0,1,0,4)	
11*	(3,1,4,2,5)	16200	(4,2)	–1300	(0,0,0,1,0,2)	(4,6,6)	(0,0,0,3,0)	
12	(3,1,2,4,5)	14900	(3,1)	–1000	(0,0,1,1,0,1)	(4,3,6)	(0,0,1,3,0)	

*Current solution is a new starting point.

10.5.3 Simulated Annealing

Simulated annealing (SA) is a class of metaheuristics based on an analogy to the annealing of metals. Consider a solid with crystalline structure being heated to a molten state and then cooled until it solidifies. If the temperature is reduced rapidly, irregularities appear in the crystal structure of the cooling solid, and the energy level of the solid is much higher than in a perfectly structured crystal. If the material is cooled slowly, with the temperature held steady at a series of levels long enough for the material to reach thermal equilibrium with its environment, the final energy level will be minimal. Let the state of the system at any temperature be described by a vector of coordinates $\mathbf{q}$. At a given temperature, while the system is attaining equilibrium, the state changes in a random way, but transitions to states with lower energy levels are more likely at lower temperatures than at higher ones.

To apply these ideas to a general optimization problem, let the system state vector $\mathbf{q}$ correspond to the objects to be optimized (job sequences, vehicle routes, or vectors of decision variables), denoted by $\mathbf{x}$. The system energy level corresponds to the objective function $f(\mathbf{x})$. As in Section 10.5.1, let $N(\mathbf{x})$ denote a neighborhood of $\mathbf{x}$. The following procedure (Floquet et al., 1994) specifies a basic SA algorithm:

- Choose an initial solution $\mathbf{x}$, an initial temperature T, a lower limit on temperature *TLOW*, and an inner iteration limit L.
- While ($T > TLOW$), **do**
  - For $k = 1, 2, \ldots, L$, **do**
    - Make a random choice of an element $\mathbf{x}' \in N(\mathbf{x})$.
    - Move_value $= f(\mathbf{x}') - f(\mathbf{x})$
    - If move_value ≤ 0 (downhill move), set $\mathbf{x} = \mathbf{x}'$
    - If move_value > 0 (uphill move), set $\mathbf{x} = \mathbf{x}'$ with probability exp(−move_value/T).
  - End inner loop
- Reduce temperature according to an *annealing schedule*. An example is new $T = cT$, where $0 < c < 1$.
- End temperature loop

Simulated annealing depends on randomization to diversify the search, both in selecting a move to evaluate (all moves to neighboring points are equally likely) and in deciding whether or not to accept a move. This basic SA algorithm uses the *Metropolis algorithm* (Johnston et al., 1989) to determine move acceptance, in which downhill moves are always accepted and uphill moves are accepted with a probability *exp*(−*move_value/T*). Note that, as T approaches 0, the probability of accepting an uphill move approaches 0. Hence, when the temperature is high, many uphill moves may be accepted, thereby possibly preventing the method from being trapped at a local minimum with respect to the neighborhood $N(\mathbf{x})$. The *Glauber algorithm* accepts all moves with the following probability:

$$\text{Glauber probability} = \frac{\exp(-\text{move\_value}/T)}{1 + \exp(-\text{move\_value}/T)}$$

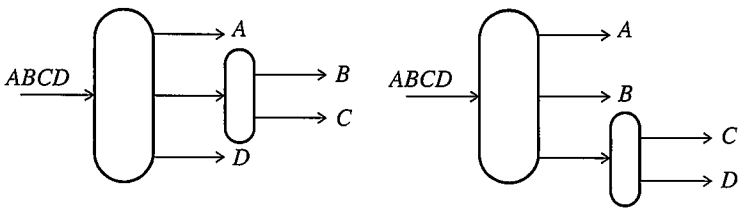

FIGURE 10.6
Separation sequences.

so here an improving move may be rejected. This leads to a search that is well diversified, so it will come closer to a global optimum, but may take longer than a Metropolis-based search, which is more likely to find a good solution quickly.

Applying simulated annealing to separation sequence synthesis

Floquet et al. (1994) applied SA to problems of separating a mixture of n components into pure products at minimal annual investment plus operating costs. The assumptions used were

- Each component of the feed stream exits in exactly one output stream of a separator. This is called *sharp separation.*
- Only one input/two output (simple) or one input/three output (complex) sharp separators are used.

Under these assumptions, the problem is to select the separators to be connected and the way they will be connected. Two possible separation sequences are shown in Figure 10.6. Floquet et al. (1994) show how to encode possible separation sequences as vectors containing the entries $\{-1, 0, 1\}$, which satisfy appropriate restrictions, and how to transform such vectors into neighboring sequences. For example, some transformations correspond to the insertion or deletion of a complex separator. Given this definition of a solution $\mathbf{x}$ and its neighborhood $N(\mathbf{x})$, and given fixed and operating costs for each type of separator that defines the objective function $f(\mathbf{x})$, the authors applied simulated annealing to find the cheapest separation sequence. In solving problems with 5, 10, and 16 components with known optimal solutions, their SA algorithm found optimal solutions for all cases, and less than 2% of the feasible sequences were evaluated when the best solution was found. Recall, however, that an optimal solution is not guaranteed in general, and there is no way to tell when an optimal solution has been found unless the optimal objective value is known in advance.

10.5.4 Genetic and Evolutionary Algorithms

With the exception of parallel implementations (which are becoming increasingly important), tabu search and simulated annealing operate by transforming a single solution at a given step. By contrast, genetic algorithms (GAs) work with a set of

solutions $P = \{x_1, x_2, \ldots, x_p\}$, called a *population*, with each population member x_i, called an *individual* or *member*. An initial population is created, and the population at the start of an iteration is modified by replacing one or more individuals with new solutions, which are created either by combining two individuals (*crossover*) or by changing an individual (*mutation*). The procedure is inspired by the evolution of populations of living organisms, whose chromosomes undergo crossover and mutation during reproduction. The genetic algorithm template that follows corresponds to the description in Reeves (1997).

- Choose an initial population, and evaluate the fitness of each individual.
- While termination condition not satisfied **do**
  - **If** crossover condition satisfied **then**
    Select parent individuals.
    Choose crossover parameters.
    Perform crossover.
  - **If** mutation condition satisfied **then**
    Choose mutation points.
    Perform mutation.
  - Evaluate fitness of offspring.
  - Update population.

We now discuss the main steps of this algorithm. For more details, see Reeves (1997) and several other articles in that issue of the *INFORMS Journal on Computing*.

Solution encoding

In the original genetic algorithms proposed by Holland (1975), the individuals were binary vectors that represented encodings of solutions. For example, if a solution $\mathbf{x}$ is a vector of n decision variables, a binary encoding is obtained by representing each component of $\mathbf{x}$ as a binary number and concatenating these bit strings. In this encoding, the bits 0 and 1 are called the *alphabet*. Other alphabets are possible, and many GAs are designed to deal with $\mathbf{x}$ vectors of n variables directly without any encoding.

Initial population and population size

The initial population should be diverse. Elements are often generated randomly using a uniform distribution over the solution space. As for population size, many authors have reported satisfactory results with population sizes as small as 30, although values of 50–100 are more common.

Crossover and mutation conditions

Crossover and mutation conditions are usually randomized rules, which determine if these operators are to be applied in the current iteration. Crossover is commonly applied in most if not all iterations, whereas mutation is applied less frequently.

Crossover and mutation

The crossover operation replaces some of the elements in each parent solution with those in the other. For example, in *one-point crossover*, with parents $P1$ and $P2$ represented by real-valued vectors, and with the crossover point after the third component, the parents and offspring are as shown here for a five-variable problem:

$$P1 = (1.2, 3, 5, 3.1, 4) \qquad O1 = (1.2, 3, 5, 6.3, 5)$$

$$P2 = (2, 1, 0, 6.3, 5) \qquad O2 = (2, 1, 0, 3.1, 4)$$

Multipoint crossover is also used, with r crossover points chosen randomly. Crossover can be further generalized by making r a random variable, and copying an element from the first parent with probability q, and from the second parent with probability $(1 - q)$. The case $q = 0.5$ is called *uniform crossover*. As an example of a mutation operator, for populations of real-valued vectors, Fogel (1995) suggests simply adding a Gaussian random variable to each component of a population member. When the individuals are bit strings, the "mutation points" are often randomly selected bits, which are then complemented to create the new solution.

In an *evolutionary algorithm*, the "classical" crossover operation is replaced by a more general "recombination" operation, which can be any procedure that combines two or more "parents" to produce one or more "offspring." As an example, the scatter search procedure described in Glover and Laguna (1997) uses linear combinations of several individuals to produce offspring. Fogel (1995) creates one offspring from each individual (a vector of n real numbers) by adding an independent, normally distributed random variable to each component. This can also be viewed as a replacement for mutation.

Fitness and its role in selecting parents and mutation candidates

In unconstrained optimization problems, you can use the value of the objective $f(\mathbf{x})$ as a measure of the "fitness" of an individual $\mathbf{x}$, but some transformation must be applied when the objective is being minimized (for example, use $-f(\mathbf{x})$). More generally, fitness can be any monotonically increasing function of the objective. Using the objective directly or some simple modification of it is rarely effective, however, because it is sensitive to objective function scaling. Consider two values of f: 10 and 20. Adding 1000 to f transforms these values to 1010 and 1020, whose percentage difference is much smaller. If the probability of being chosen to be a parent is equal to an individual's share of total population fitness, then before adding 1000, these probabilities are 1/3 and 2/3, and after they are 1010/2030 and 1020/2030, both close to 0.5. Reeves (1997) recommends ranking procedures, the simplest of which ranks individuals in order of their objective function values and sets fitness equal to that ranking. Once a measure of fitness has been chosen, a common procedure for selecting parents or mutation candidates is random selection from the population, using a probability distribution that assigns higher probabilities to individuals with higher fitness, such as that used in the previous example.

Updating the population

After a number of new solutions are produced by crossover (or more generally, recombination) and mutation operations, improved solutions must be incorporated into the population. The best solution found thus far is almost always retained. A common strategy replaces a certain fraction of the remaining individuals, either with improved offspring or with new individuals chosen to maintain diversity. Another strategy is *tournament selection*, in which new solutions and current population members compete in a "tournament." Each solution competes with K other solutions, which may be randomly selected, and, in each pairwise comparison, the solution with best fitness value wins. If P is the population size, the P solutions with the most wins become the new population.

Constraints

When there are constraints, GAs face a fundamental difficulty, namely that many crossover or mutation operators rarely yield feasible offspring, even if the parents are feasible. This can lead to a population with an excessive number of infeasible solutions. To alleviate this problem, GAs often include a penalty function in f (see Section 8.4) to measure fitness. A value must be chosen for the penalty weight, however. If this is too small, the original problem of too many infeasible solutions remains, and if it is too large, the search tends to reject points with small infeasibilities, even if they are close to an optimal solution.

For an excellent introduction to genetic algorithms, see the website constructed by Marek Obitko of the Czech Technical University at http://cs.felk.cvut.cz/~xobitko/ga/. It contains a genetic algorithm, coded as a Java Applet, which the user can run interactively, specifying his or her own objective if desired.

10.5.5 Using the Evolutionary Algorithm in the Premium Excel Solver

An evolutionary algorithm is included in the current release of Frontline Systems' Premium Excel Solver (for current information, see www.frontsys.com). It is invoked by choosing "Standard Evolutionary" from the Solver dropdown list in the Solver Parameters Dialog Box. The other nonlinear solver is "Standard GRG Nonlinear," which is the GRG2 solver described in Section 8.7. As discussed there, GRG2 is a gradient-based local solver, which will find the "nearest" local solution to its starting point. The evolutionary solver is much less likely to stop at a local minimum, as we illustrate shortly.

The "Options" box for the evolutionary solver is shown in Figure 10.7. The solver stops when either the time or iterations limit is reached, or when 99% of the population members have fitness values such that the fractional deviation between largest and smallest is less than the "Convergence" tolerance shown in the figure. The population size cannot be less than 10 or more than 200, and the initial population is chosen mainly by random sampling from within the hyperrectangle specified by the bounds on the variables. You are advised to define bounds for all variables, so the initial sampling can be performed from a hyperrectangle of limited

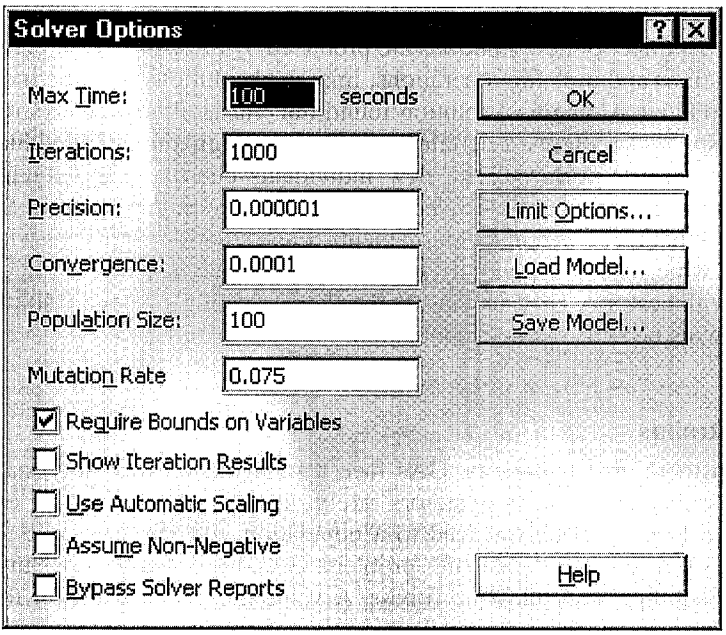

FIGURE 10.7
Options dialog for the evolutionary solver. Permission by Microsoft.

size. The initial decision variable values entered in the spreadsheet are also included in the initial population, perhaps several times, so the method benefits from a good starting point.

As in the GA template presented earlier, an iteration of the evolutionary algorithm consists of a crossover step involving two or more parents, mutation of a single population member (which is performed with the probability specified in the "Mutation Rate" box), and an optional local search. Note that the default mutation probability is 0.075, so if this value is used, mutation is fairly rare. Three mutation strategies are possible, one of which is selected if mutation is performed. A single variable in the single population element is selected for mutation. The three strategies alter the variables value as follows: (1) replace it by a random value from a uniform distribution; (2) move it to either its upper or lower bound; or (3) increase or decrease it by a randomly chosen amount, whose magnitude decreases as the iterations progress. In the population update, if a new element is "worse" than all population members, it is discarded. If not, the member to be replaced may not be the worst. Instead, a probabilistic replacement process is used, where the worst members have higher probabilities of being replaced. Computational experiments have shown that this leads to a more diverse population and to overall better performance than if the worst element were replaced each time. The measures of goodness used to define better and worse are complex, involving both objective values and penal-

TABLE 10.8
GRG results for Branin problem

Changing cells	Starting point 1	Starting point 2	Starting point 3
Initial x_1	1	−5	−5
x_2	1	5	10
Final x_1	3.141590675	9.000272447	−2.619502503
x_2	2.274999493	0.999727553	10
Final objective	0.397887358	2.550824843	2.791184064

ties for infeasibility. In some cases, infeasible points with good objective function values are accepted into the population. In others, an attempt is made to modify a solution to "repair" infeasibilities.

Table 10.8 shows the result of applying the "Standard GRG Solver" to a two-variable, one-constraint problem called the Branin problem that has three local optima and a global optimum with objective function value of 0.397. The objective function is constructed in three steps:

$$
\begin{aligned}
t_1 &= \left(\frac{x_1}{\pi}\right)\left[\left(\frac{1.275x_1}{\pi}\right) - 5\right] \\
t_2 &= (x_2 - t_1 - 6)^2 \\
t_3 &= 9.602113\cos(x_1) + 10 \\
f &= t_2 + t_3
\end{aligned}
\tag{10.7}
$$

and the problem is

$$
\begin{aligned}
&\text{Minimize:} \quad f \\
&\text{Subject to:} \quad x_1 + x_2 \leq 10
\end{aligned}
$$

Starting from (1, 1), GRG finds the global solution, but it finds the two inferior local optima starting from the points (–5, 5) or (–5, 10). . . . The evolutionary solver finds the global optimum to six significant figures from any starting point in 1000 iterations.

This problem is very small, however, with only two decision variables. As the number of decision variables increases, the number of iterations required by evolutionary solvers to achieve high accuracy increases rapidly. To illustrate this, consider the linear project selection problem shown in Table 10.9. The optimal solution is also shown there, found by the LP solver. This problem involves determining the optimal level of investment for each of eight projects, labeled *A* through *H*, for which fractional levels are allowed. Each project has an associated net present value (NPV) of its projected net profits over the next 5 years and a different cost in each of the 5 years, both of which scale proportionately to the fractional level of investment. Total costs in each year are limited by forecasted budgets (funds available in

TABLE 10.9
Project selection problem with budget constraints

		Project									Optimal solution (Σ projects)	Funds available
		A	*B*	*C*	*D*	*E*	*F*	*G*	*H*			
Net Present Value		**151**	**197**	**119**	**70**	**130**	**253**	**165**	**300**	**Total NPV**	**839.11**	
Costs:	**Year 1**	20	100	20	30	50	40	50	80	Year 1	230	230
	Year 2	20	10	10	30	10	20	40	30	Year 2	90	100
	Year 3	20	0	10	30	10	20	10	20	Year 3	50	50
	Year 4	20	0	10	20	10	20	10	0	Year 4	30	50
	Year 5	10	30	10	10	10	20	10	0	Year 5	50	50
Optimal Decisions		0	0.667	0.222	0	0	1	0.778	1			

Abbreviation: NPV = net present value.

TABLE 10.10
Final objective values obtained by evolutionary solver

Run	Iterations	Best objective value found
1	1,000	769.91
2	1,000	771.77
3	1,000	789.23
4	5,000	804.67
5	5,000	784.99
6	10,000	811.21
7	10,000	792.47
8	28,244	806.69

the last column), and the problem is to maximize the total value from all projects subject to the budget constraints. The summed NPV and the summed annual cost for each year at the optimum is in the next to last column. Note that the optimal solution in the bottom row of the table is at an extreme point because there are eight variables and eight active constraints (including those that require each decision variable to be between zero and one).

Table 10.10 shows the performance of the evolutionary solver on this problem in eight runs, starting from an initial point of zero. The first seven runs used the iteration limits shown, but the eighth stopped when the default time limit of 100 seconds was reached. For the same number of iterations, different final objective function values are obtained in each run because of the random mechanisms used in the mutation and crossover operations and the randomly chosen initial population. The best value of 811.21 is not obtained in the run that uses the most iterations or computing time, but in the run that was stopped after 10,000 iterations. This final value differs from the true optimal value of 839.11 by 3.32%, a significant difference, and the final values of the decision variables are quite different from the optimal values shown in Table 10.9.

If constraints that the decision variables be binary are added, however, the evolutionary solver reaches the optimal objective value of 767 in two runs with a 5000-iteration limit, and in one of two runs with a 1000-iteration limit. This is because only $2^8 = 256$ possible solutions need to be explored. Hence, if high accuracy is required, general-purpose evolutionary algorithms seem best suited to small problems with continuous variables, but they can find good solutions to larger problems, including integer and mixed-integer problems. Of course, evolutionary solvers (or any other search method) can be combined with local solvers like GRG, simply by starting the local solver at the final point obtained by the search procedure. If the problem is smooth and this point is near a global optimum, the local solver may well find the global solution to high accuracy. A local solver or heuristic can also be combined with scatter search, as described in the next section.

10.5.6 Scatter Search

Scatter search, described in Glover and Laguna (1997) and Glover (1998), is a population-based search method that primarily uses deterministic principles to strategically guide the search. Its steps are shown here, stated for problems whose only constraints are bounds on the variables. These bounds are taken into account when generating trial and combined solutions. It may, however, be applied to problems with more general constraints by augmenting the objective function with a penalty function (see Section 8.4).

Steps of Scatter Search

1. Create an initial diverse trial set of solutions.
2. Apply an *improvement method* to some or all trial solutions. Save the r best solutions found as members of the initial *reference set, R.*
3. Repeat steps 1 and 2 until some designated number of reference set solutions have been found.
4. Select subsets of the reference set to use in step 5.
5. For each subset chosen in step 4, use a *solution combination method* to produce one or more combined solutions.
6. Starting from each of the combined solutions in step 5, use the improvement method to create a set of enhanced solutions.
7. If an enhanced solution is better than any member of the reference set, insert it in the reference set and delete the worst member of the set.
8. Return to step 4, and repeat until some stopping condition is met. Such conditions may be based on elapsed time or iterations, or on lack of improvement in the objective.

Explanation of scatter search steps

Step 1 starts with a large set of "seed" solutions, which may be created by heuristics or by random generation. One possible implementation then generates a diverse subset of these by choosing some initial seed solution, then selecting a second one that maximizes the distance from the initial one. The third one maximizes the distance from the nearest of the first two, and so on.

The improvement method used in steps 2 and 6 may be one of the following:

- A heuristic descent method like that outlined in Figure 10.5 if the problem is combinatorial.
- A local NLP solver like the GRG or SQP algorithms described in Chapter 8; in this case, the problem must be a constrained, possibly nonconvex problem with continuous variables.
- Simply an evaluation of the objective and constraint functions.

Steps 4 through 6 are the scatter search counterparts to the crossover and mutation operators in genetic algorithms, and the reference set corresponds to the GA

population. The solution combination method produces combined solutions that are linear combinations of those in the subsets produced in step 4. However, variables that are required to take on integer values are subjected to generalized rounding processes, that is, processes for which the rounding of each successive variable depends on the outcomes of previous roundings. In the simplest case, two subsets, each containing a single solution, are chosen, with one solution selected to have a good objective value (to intensify the search in the neighborhood of good solutions) and the second chosen to be far from the first (to diversify the search). In this case, taking linear combinations of these two solutions produces new ones that are on the line segment between *and beyond* the two "parent" solutions. These are then used as starting points for the improvement method.

Scatter search has been implemented in software called OPTQUEST (see www.opttek.com). OPTQUEST is available as a callable library written in C, which can be invoked from any C program, or as a dynamic linked library (DLL), which can be called from a variety of languages including C, Visual Basic, and Java. The callable library consists of a set of functions that (1) input the problem size and data, (2) set options and tolerances, (3) perform steps 1 through 3 to create an initial reference set, (4) retrieve a trial solution from OPTQUEST to be input to the improvement method, and (5) input the solution resulting from the improvement method back into OPTQUEST, which uses it as the input to step 7 of the scatter search protocol. The improvement method is provided by the user. We use the term *improvement* loosely here because the user can simply provide an evaluation of the objective and constraint functions.

Optimizing simulations

OPTQUEST has also been combined with several Monte Carlo and discrete-event simulators. The Monte Carlo simulators include an Excel add-on called Crystal Ball (see www.decisioneering.com). It allows a user to define a subset of spreadsheet cells as random input variables with specified probability distributions and to designate several output cells that depend on these inputs and on other nonrandom input cells. The program then samples a specified number of times from the input distributions, evaluates the output cells, and computes statistics and histograms of the distributions of each output cell. In an optimization, a set of (nonrandom) input cells are designated as decision cells, some statistic associated with an output cell (typically its mean) is selected to be the objective function, and other statistics of other outputs may be taken as constraints. OPTQUEST is then applied to vary the decision variables in order to optimize the objective subject to the constraints. For each trial solution suggested by OPTQUEST, a complete simulation is run, and the designated cell statistics are returned to OPTQUEST. As an example, one can minimize the average of total holding plus set-up cost in an inventory problem with random demand, by choosing an optimal reorder level and order quantity. In such problems, the average value returned by Crystal Ball is only an estimate of the true average, so it contains some random error, which can be reduced by using a larger sample size. OPTQUEST is able to process these noisy objective values and still return a good approximation to an optimal solution.

TABLE 10.11
OPTQUEST applied to problem in Table 10.8

Iteration	Best objective	*x*1	*x*2
1	27.7029	1	1
4	8.8072	8.00106	1.99894
19	0.471901	3.19623	1.98847
89	0.406846	3.18067	2.28509
150	0.401044	3.15053	2.3207
205	0.39855	3.14473	2.29735
335	0.398046	3.14732	2.26958
459	0.397898	3.14194	2.2716
565	0.397887	3.14159	2.2751

TABLE 10.12
OPTQUEST applied to problem in Table 10.9

Iteration	Best objective	*A*	*B*	*E*
1	0.00	0	0	0
3	300.00	0	0	0
10	565.00	1	0	1
35	604.76	0.13	0.67	0.89
67	609.51	0.89	0.53	0.72
78	721.47	0.07	0.40	0.78
80	730.94	0.08	0.48	0.77
82	742.77	0.09	0.58	0.76
84	757.56	0.10	0.70	0.75
159	766.88	0.09	0.72	0.77
161	769.21	0.09	0.72	0.78
1005	794.13	0.28	0.69	0.72
2084	794.35	0.28	0.69	0.72
2963	794.73	0.27	0.69	0.72
4024	794.82	0.27	0.69	0.72
4996	797.25	0.24	0.70	0.72
Optimal	**839.11**	**0.00**	**0.67**	**0.00**

OPTQUEST examples

Crystal Ball can deal with spreadsheets that contain no random variables, and OPTQUEST can be applied to deterministic optimization problems arising from such spreadsheets. Table 10.11 shows the performance of OPTQUEST applied to the two-variable, one-constraint problem defined in Equations (10.7), which was solved by an evolutionary algorithm in Section 10.5 to six-digit accuracy in 1000 iterations. OPTQUEST finds the same solution with similar effort.

Table 10.12 shows OPTQUEST's progress on the project selection LP, whose optimal solution is given in Table 10.9. Initial progress is rapid, but it slows rapidly after about 1000 iterations, and after 5000 iterations the best objective value found is 797.25, about 5% short of the optimal value of 839.11. The values of variables *A*,

TABLE 10.13
Classification of metaheuristic search procedures

Metaheuristic	Classification
Genetic algorithms	M/S/P
Scatter search	A/N/P
Simulated annealing	M/S/1
Tabu search	A/N/1

B, and *E* are also shown. Although *B* is reasonably near its optimal value, *A* and *E* are far from theirs. This performance is comparable to that of the evolutionary algorithm in the Extended Excel Solver, shown in Table 10.10. If the decision variables in this problem must be binary, however, then OPTQUEST finds the optimal solution, whose objective value is 767, in only 116 iterations. The evolutionary algorithm found this same optimal solution in one of two runs using 1000 iterations.

Classifying metaheuristics

Glover and Laguna (1997) classify metaheuristics according to a three-attribute scheme as shown in Table 10.13. In the first position, "A" denotes the use of adaptive memory, and "M" means memoryless. An important feature of tabu and scatter search is remembering attributes of past solutions to guide the search in an adaptive way, that is, the length and operation of the memory may vary as the search progresses. Genetic algorithms and simulated annealing are viewed as not having *adaptive* memory, although GAs do retain information on the past through the population itself. An "N" in the second position indicates that a systematic neighborhood search is used to find an improved solution, and "S" indicates that a randomized sampling procedure is employed. Although traditional GA and SA methods use random sampling, some recent SA and evolutionary algorithms either replace this with a neighborhood search or initiate a search from a point found by a randomized procedure. A "1" in the third position indicates that the method uses a population of size 1, that is, it moves from a current solution to a new one; "P" indicates that a population of size *P* is used.

10.6 OTHER SOFTWARE FOR GLOBAL OPTIMIZATION

In addition to the Premium Excel Solver and Optquest, there are many other software systems for constrained global optimization; see Pintér (1996b), Horst and Pardalos (1995), and Pintér (1999) for further information. Perhaps the most widely used of these is LGO (Pintér, 1999), (Pintér, 2000), which is intended for smooth problems with continuous variables. It is available as an interactive development environment with a graphical user interface under Microsoft Windows, or as a callable library, which can be invoked from an application written by the user in

Fortran, C/C++, Visual Basic, or Delphi. The user provides the model coded as a corresponding subroutine or function.

LGO operates in two phases. The first is the global phase, which attempts to find a point which is a good approximation to a global optimum. It uses an adaptive deterministic as well as a random sampling technique, with an option to apply these within a branch-and-bound procedure. The ensuing local phase starts from this point and finds an improved point, which is the "nearest" local optimum, using a combination of local gradient-based NLP algorithms.

REFERENCES

Avriel, M. *Nonlinear Programming*. Prentice-Hall, Englewood Cliffs, NJ (1976).

Barnes, J. W.; and L. K. Vanston. "Scheduling Jobs with Linear Delay Penalties and Sequence Dependent Setup Costs." *Oper Res* **29:** (1) 146–161 (1981).

Crainic, T. G.; and G. Laporte, eds. *Fleet Management and Logistics*. Kluwer Academic Publishers, Boston/Dordrecth/London (1998).

Floquet, P.; L. Pibouleau; and S. Domenech. "Separation Sequence Synthesis: How to Use a Simulated Annealing Procedure." *Comput Chem Eng* **18:** 1141–1148 (1994).

Floudas, C. A. "Global Optimization in Design and Control of Chemical Process Systems." *J Process Cont* **10:** 125–134 (2000a).

Floudas, C. A. *Deterministic Global Optimization: Theory, Methods, and Applications*. Kluwer Academic Publishers, Norwell, MA (2000b).

Fogel, D. B. "A Comparison of Evolutionary Programming and Genetic Algorithms on Selected Constrained Optimization Problems." *Simulation* **64:** 397–404 (1995).

Glover, F. *A Template for Scatter Search and Path Relinking*, working paper, School of Business, University of Colorado, Boulder, CO, 80309 (1998).

Glover, F.; and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA (1997).

Hiriart-Urruty, J. D.; and C. Lemarechal. *Convex Analysis and Minimization Algorithms*. Springer-Verlag, Berlin (1993).

Holland, J. H. *Adaptations in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI (1975), reissued by MIT Press, Cambridge, MA (1992).

Horst, R.; and P. M. Pardalos. *Handbook of Global Optimization*. Kluwer Academic Publishers, Dordrecht/Boston/London (1995).

Johnston, D. S.; C. R. Aragon; L. A. McGeoch; et al. "Optimization by Simulated Annealing: An Experimental Evaluation: Part 1, Graph Partitioning." *Oper Res* **37:** 865–892 (1989).

Kearfott, R. B. *Rigorous Global Search: Continuous Problems*. Kluwer Academic Publishers, Norwell, MA (1996).

Laguna, M.; J. W. Barnes; and F. Glover. "Tabu Search Methods for a Single Machine Scheduling Problem." *J Intell Manufact* **2:** 63 (1991).

Locatelli, M.; and F. Schoen. "Random Linkage: A Family of Acceptance/Rejection Algorithms for Global Optimization." *Math Prog* **85:** 379–396 (1999).

Pintér, J. D. *Global Optimization in Action (Continuous and Lipschitz Optimization: Algorithms, Implementations, and Applications)*. Kluwer Academic Publishers, Norwell, MA (1996a).

Pintér, J. D. "Continuous Global Optimization Software: a Brief Review." *Optima* **52:** 1–8 (1996b).

Pintér, J. D. "Continuous Global Optimization." *Interactive Transactions of ORMS* **2** (1999). Available online at http://catt.bus.okstate.edu/itorms.

Pintér, J. D. *Computational Global Optimization in Nonlinear Systems: An Interactive Tutorial.* Published for INFORMS by Lionheart Publishing, Atlanta (2000). Available online at www.lionhrtpub.com/books.

Reeves, C. R. "Genetic Algorithms for the Operations Researcher." *INFORMS J Comput* **9**(3): 231–250 (1997).

Rinnooy Kan, A. H. G.; and G. T. Timmer. "Stochastic Global Optimization Methods, Part 2: Multi Level Methods." *Math Prog* **39:** 57–78 (1987).

Rinnooy Kan, A. H. G.; and G. T. Timmer. "Global Optimization," Chapter 9 In *Handbooks in OR and MS, vol. 1.* G. L. Nemhauser et al., eds. Elsevier Science Publishers B. V., Amsterdam, The Netherlands (1989).

SUPPLEMENTARY REFERENCES

Adjiman, C. S.; and C. A. Floudas. "Rigorous Convex Underestimators for General Twice-Differentiable Problems." *J Global Optim* **9:** 23 (1996).

Adjiman, C. S.; I. P. Androulakis; C. D. Maranas; and C. A. Floudas. "A Global Optimization Method aBB for Process Design." *Comput Chem Eng* **20:** S419–424 (1996).

Adjiman, C. S.; I. P. Androulakis; and C. A. Floudas. "A Global Optimization Method aBB, for General Twice-Differentiable Constrained NLPs II. Implementation and Computational Results." *Comput Chem Eng* **22:** 1159–1179 (1998).

Adjiman, C. S.; S. Dallwig; C. A. Floudas; and A. Neumaier. "A Global Optimization Method, aBB, for General Twice-Differentiable Constrained NLPs—I, Theoretical Advances." *Comput Chem Eng* **22:** 1137–1158 (1998).

Angeline, P. J.; and K. E. Kinnear Jr. *Advances in Genetic Programming, vol. 2.* MIT Press, Cambridge, MA (1998).

Azzaro-Pantel, C.; L. Bernal-Haro; P. Baudet; S. Demenech, et al. "A Two-Stage Methodology for Short-term Batch Plant Scheduling: Discrete-Event Simulation and Genetic Algorithm." *Comput Chem Eng* **22:** 1461–1481 (1998).

Choi, H.; J. W. Ko; and V. Manousiouthakis. "A Stochastic Approach to Global Optimization of Chemical Processes." *Comput Chem Eng* **23:** 1351–1358 (1999).

Esposito, W. R.; and C. A. Floudas. "Parameter Estimation in Nonlinear Algebraic Models via Global Optimization." *Comput Chem Eng* **22:** S213–220 (1998).

Fogel, D. B. "A Comparison of Evolutionary Programming and Genetic Algorithms on Selected Constrained Optimization Problems." *Simulation* **64:** 3499 (1995).

Friese, T.; P. Ulbig; and S. Schulz. "Use of Evolutionary Algorithms for the Calculation of Group Contribution Parameters in Order to Predict Thermodynamic Properties. Part 1: Genetic Algorithms." *Comput Chem Eng* **22:** 1559–1572 (1998).

Garrard, A.; and E. S. Fraga. "Mass Exchange Network Synthesis Using Genetic Algorithms." *Comput Chem Eng* **22:** 1837–1850 (1998).

Greeff, D. J.; and C. Aldrich. "Empirical Modelling of Chemical Process Systems with Evolutionary Programming." *Comput Chem Eng* **22:** 995–1005 (1998).

Gross, B.; and P. Roosen. "Total Process Optimization in Chemical Engineering with Evolutionary Algorithms." *Comput Chem Eng* **22:** S229–236 (1998).

Hanagandi, V.; and M. Nikolaou. "A Hybrid Approach to Global Optimization Using a Clustering Algorithm in a Genetic Search Framework." *Comput Chem Eng* **22:** 1913–1925 (1998).

Haupt, R. L. *Practical Genetic Algorithms.* Wiley, New York (1998).

Jung, J. H.; C. H. Lee; and I-B. Lee. "A Genetic Algorithm for Scheduling of Multi-Product Batch Processes." *Comput Chem Eng* **22:** 1725–1730 (1998).

Karr, C. L.; and L. M. Freeman. *Industrial Applications of Genetic Algorithms.* CRC Press, Boca Raton, FL (1998).

Löhl, T.; C. Schulz; and S. Engell. "Sequencing of Batch Operations for Highly Coupled Production Process: Genetic Algorithms Versus Mathematical Programming." *Comput Chem Eng* **22:** S579–585 (1998).

Mitchell, M. *An Introduction to Genetic Algorithms.* The MIT Press, Cambridge, MA (1998).

Pham, Q.T. "Dynamic Optimization of Chemical Engineering Processes by an Evolutionary Method." *Comput Chem Eng* **22:** 1089–1097 (1998).

Sen, S.; S. Narasimhan; and K. Deb. "Sensor Network Design of Linear Processes Using Genetic Algorithms." *Comput Chem Eng* **22:** 385–390 (1998).

Wang, K.; Löhl, T.; Stobbe, M.; and S. Engell. "A Genetic Algorithm for Online-scheduling of a Multiproduct Polymer Batch Plant." *Comput Chem Eng* **24:** 393–400 (2000).

Zamora, J. M.; and I. E. Grossmann. "Continuous Global Optimization of Structured Process Systems Models." *Comput Chem Eng* **22:** 1749–1770 (1998).

PART III

APPLICATIONS OF OPTIMIZATION

THIS SECTION OF the book is devoted to representative applications of the optimization techniques presented in Chapters 4 through 10. Chapters 11 through 16 include the following major application areas:

1. Heat transfer and energy conservation (Chapter 11)
2. Separations (Chapter 12)
3. Fluid flow (Chapter 13)
4. Reactors (Chapter 14)
5. Large-scale plant design and operations (Chapter 15)
6. Integrated planning, scheduling, and control (Chapter 16)

Each chapter presents several detailed studies illustrating the application of various optimization techniques. The following matrix shows the classification of the examples with respect to specific techniques. Truly optimal design of process plants cannot be performed by considering each unit operation separately. Hence, in Chapter 15 we discuss the optimization of large-scale plants, including those represented by flowsheet simulators.

We have not included any homework problems in Chapters 11 through 16. As a general suggestion for classroom use, parameters or assumptions in each example can be changed to develop a modified problem. By changing the numerical method employed or the computer code one can achieve a variety of problems.

Classification of optimization applications (example number is in parentheses) by technique

Methods	Chapter 11	12	13	14	15	16
Analytical solution	Waste heat recovery (11.1)		Pipe diameter (13.1)			Material balance reconciliation (16.4)
One-dimensional search	Multistage evaporator (11.3)	Reflux ratio of distillation column (12.4)	Fixed-bed filter (13.3)			
Unconstrained optimization		Nonlinear regression of VLE data (12.3)	Minimum work of compression (13.2)			
Linear programming	Boiler/turbo generator system (11.4)			Thermal cracker (14.1)		Planning and scheduling (16.1)
Nonlinear programming		Staged-Distillation column (12.1) Liquid extraction column (12.2)	Gas transmission network (13.4)	Ammonia reactor (14.2) Alkylation reactor (14.3) CVD reactor (14.5)	Refrigeration process (15.2) Extractive distillation (15.3) Operating margin (15.4)	Reactor control (16.3)
Mixed integer programming	Heat exchanger (11.2)		Gas transmission network (13.4)	Protein folding (14.4) Reaction synthesis (14.6)		Batch scheduling (16.2)