CHAPTER 14

Metaheuristics

Several of the preceding chapters have described algorithms that can be used to obtain an optimal solution for various kinds of OR models, including certain types of linear programming, integer programming, and nonlinear programming models. These algorithms have proven to be invaluable for addressing a wide variety of practical problems. However, this approach doesn't always work. Some problems (and the corresponding OR models) are so complicated that it may not be possible to solve for an optimal solution. In such situations, it still is important to find a good feasible solution that is at least reasonably close to being optimal. Heuristic methods commonly are used to search for such a solution.

A **heuristic method** is a procedure that is likely to discover a very good feasible solution, but not necessarily an optimal solution, for the specific problem being considered. No guarantee can be given about the quality of the solution obtained, but a well-designed heuristic method usually can provide a solution that is at least nearly optimal (or conclude that no such solutions exist). The procedure also should be sufficiently efficient to deal with very large problems. The procedure often is a full-fledged *iterative algorithm,* where each iteration involves conducting a search for a new solution that might be better than the best solution found previously. When the algorithm is terminated after a reasonable time, the solution it provides is the best one that was found during any iteration.

Heuristic methods often are based on relatively simple common-sense ideas for how to search for a good solution. These ideas need to be carefully tailored to fit the specific problem of interest. Thus, heuristic methods tend to be ad hoc in nature. That is, each method usually is designed to fit a specific problem type rather than a variety of applications.

For many years, this meant that an OR team would need to start from scratch to develop a heuristic method to fit the problem at hand, whenever an algorithm for finding an optimal solution was not available. This all has changed in relatively recent years with the development of powerful metaheuristics. A **metaheuristic** is a general solution method that provides both a general structure and strategy guidelines for developing a specific heuristic method to fit a particular kind of problem. Metaheuristics have become one of the most important techniques in the toolkit of OR practitioners.

This chapter provides an elementary introduction to metaheuristics. After describing the general nature of metaheuristics in the first section, the following three sections will introduce and illustrate three commonly used metaheuristics.

14.1 THE NATURE OF METAHEURISTICS

To illustrate the nature of metaheuristics, let us begin with an example of a small but modestly difficult nonlinear programming problem:

An Example: A Nonlinear Programming Problem with Multiple Local Optima

Consider the following problem:

Maximize $f(x) = 12x^5 - 975x^4 + 28{,}000x^3 - 345{,}000x^2 + 1{,}800{,}000x,$

subject to

$0 \leq x \leq 31.$

Figure 14.1 graphs the objective function $f(x)$ over the feasible values of the single variable x. This plot reveals that the problem has three local optima, one at $x = 5$, another at $x = 20$, and the third at $x = 31$, where the global optimum is at $x = 20$.

The objective function $f(x)$ is sufficiently complicated that it would be difficult to determine where the global optimum lies without the benefit of viewing the plot in Fig. 14.1. Calculus could be used, but this would require solving a polynomial equation of the fourth degree (after setting the first derivative equal to zero) to determine where the critical points lie. It would even be difficult to ascertain that $f(x)$ has multiple local optima rather than just a global optimum.

This problem is an example of a *nonconvex programming* problem, a special type of nonlinear programming problem that typically has multiple local optima. Section 13.10

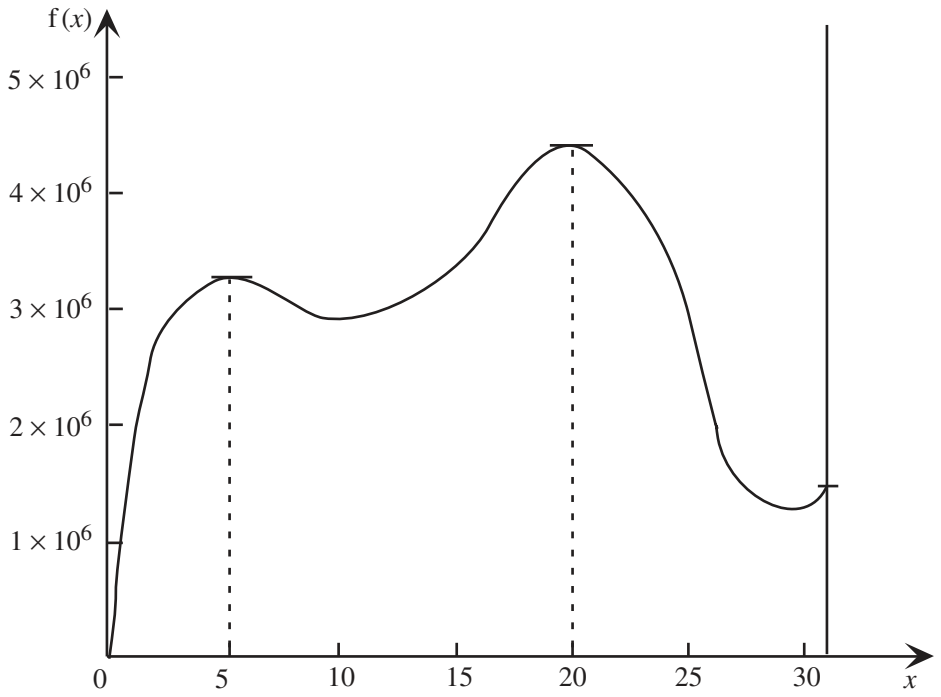


FIGURE 14.1
A plot of the value of the objective function over the feasible range, $0 \leq x \leq 31$, for the nonlinear programming example. The local optima are at $x = 5$, $x = 20$, and $x = 31$, but only $x = 20$ is a global optimum.

discusses nonconvex programming and even introduces a software package (Evolutionary Solver) that uses the kind of metaheuristic described in Sec. 14.4.

For nonlinear programming problems that appear to be somewhat difficult, like this one, a simple heuristic method is to conduct a **local improvement procedure.** Such a procedure starts with an initial trial solution and then, at each iteration, searches in the neighborhood of the current trial solution to find a better trial solution. This process continues until no improved solution can be found in the neighborhood of the current trial solution. Thus, this kind of procedure can be viewed as a *hill-climbing procedure* that keeps climbing higher on the plot of the objective function (assuming the objective is maximization) until it essentially reaches the top of the hill. A well-designed local improvement procedure usually will be successful in converging to a *local* optimum (the top of a hill), but it then will stop even if this local optimum is not a *global* optimum (the top of the tallest hill).

For example, the *gradient search procedure* described in Sec. 13.5 is a local improvement procedure. If it were to start with, say, $x = 0$ as the initial trial solution in Fig. 14.1, it would climb up the hill by trying successively larger values of x until it essentially reaches the top of the hill at $x = 5$, at which point it would stop. Figure 14.2 shows a typical sequence of values of $f(x)$ that would be obtained by such a local improvement procedure when starting from far down the hill.

Since the nonlinear programming example depicted in Fig. 14.1 involves only a single variable, the bisection method described in Sec. 13.4 also could be applied to this particular problem. This procedure is another example of a local improvement procedure, since each iteration starts from the current trial solution to search in its neighborhood (defined by a current lower bound and upper bound on the value of the variable) for a better solution. For example, if the search were to begin with a lower bound of $x = 0$ and an upper bound of $x = 6$ in Fig. 14.1, the sequence of trial solutions obtained by the bisection method would be $x = 3$, $x = 4.5$, $x = 5.25$, $x = 4.875$, and so forth as it converges to $x = 5$. The corresponding values of the objective function for these four trial solutions are 2.975 million, 3.286 million, 3.300 million, and 3.302 million, respectively. Thus, the second iteration provides a relatively large improvement over the first one (311,000), the third iteration gives a considerably smaller improvement (14,000), and the fourth iteration yields only a very small improvement (2000). As depicted in Fig. 14.2, this pattern is rather typical of local improvement procedures (although with some variation in the rate of convergence to the local maximum).

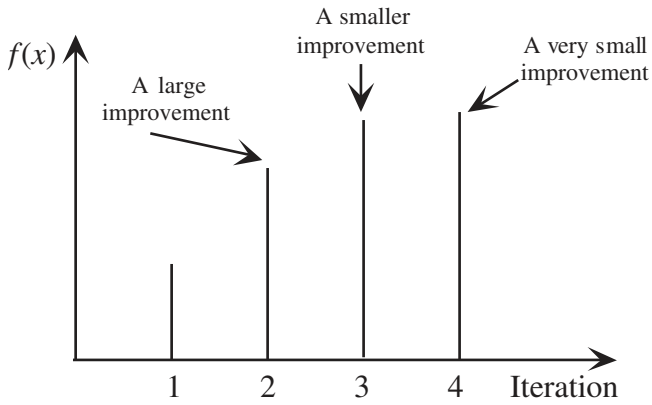


FIGURE 14.2
A typical sequence of objective function values for the solutions obtained by a local improvement procedure as it converges to a local optimum when it is applied to a maximization problem.

Just as with the gradient search procedure, this search with the bisection method would get trapped at the local optimum at $x = 5$, so it never would find the global optimum at $x = 20$. Like other local improvement procedures, both the gradient search procedure and the bisection method are designed only to keep improving on the current trial solutions within the local neighborhood of those solutions. Once they climb to the top of a hill, they must stop because they cannot climb any higher within the local neighborhood of the trial solution at the top of the hill. This illustrates the drawback of any local improvement procedure.

The drawback of a local improvement procedure: When a well-designed local improvement procedure is applied to an optimization problem with multiple local optima, the procedure will converge to one local optimum and then stop. Which local optimum it finds depends on where the procedure begins the search. Thus, the procedure will find the global optimum only if it happens to begin the search in the neighborhood of this global optimum.

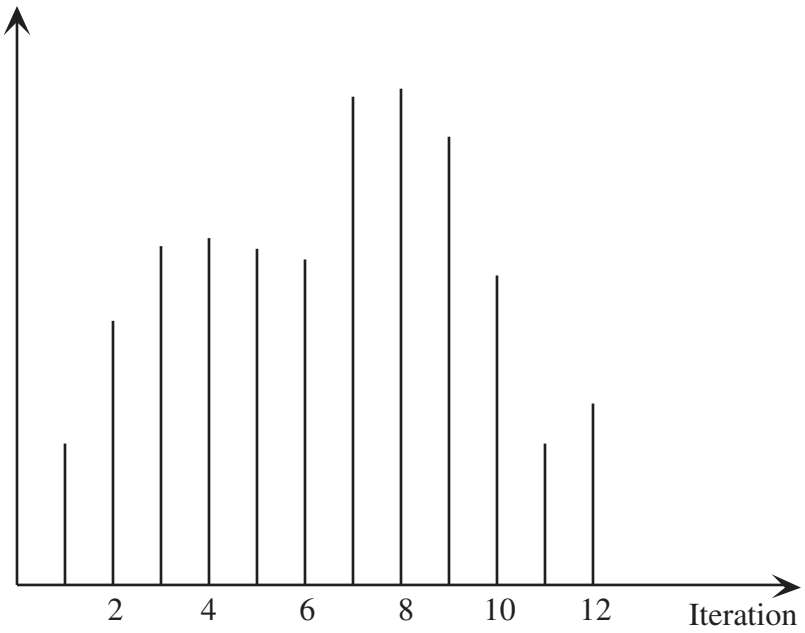
To try to overcome this drawback, one can restart the local improvement procedure a number of times from randomly selected initial trial solutions. Restarting from a new part of the feasible region often will lead to a new local optimum. Repeating this a number of times increases the chance that the best of the local optima obtained actually will be the global optimum. (As described in Sec. 13.10, this is what is done with either Solver or ASPE when using the *GRG Nonlinear* solving method and then selecting the *Use Multistart* option.) This approach works well on small problems, like the one-variable nonlinear programming example depicted in Fig. 14.1. However, it is much less successful on large problems with many variables and a complicated feasible region. When the feasible region has numerous "nooks and crannies" and restarting a local improvement procedure from only one of them will lead to the global optimum, restarting from randomly selected initial trial solutions becomes a haphazard way to reach the global optimum.

What is needed instead is a more structured approach that uses the information being gathered to guide the search toward the global optimum. This is the role that a metaheuristic plays.

The nature of metaheuristics: A metaheuristic is a general kind of solution method that orchestrates the interaction between local improvement procedures and higher level strategies to create a process that is capable of escaping from local optima and performing a robust search of a feasible region.

Thus, one key feature of a metaheuristic is its ability to escape from a local optimum. After reaching (or nearly reaching) a local optimum, different metaheuristics execute this escape in different ways. However, a common characteristic is that the trial solutions that immediately follow a local optimum are allowed to be inferior to this local optimum. Consequently, when a metaheuristic is applied to a maximization problem (such as the example depicted in Fig. 14.1), the objective function values for the sequence of trial solutions obtained typically would follow a pattern similar to that shown in Fig. 14.3. As with Fig. 14.2, the process begins by using a local improvement procedure to climb to the top of the current hill (iteration 4). However, rather than stopping there, the metaheuristic might guide the search a little way down the other side of this hill until it can start climbing to the top of the tallest hill (iteration 8). To verify that this appears to be the global optimum, a metaheuristic continues exploring further before stopping (iteration 12).

Figure 14.3 illustrates both an advantage and a disadvantage of a well-designed metaheuristic. The advantage is that it tends to move relatively quickly toward very good solutions, so it provides a very efficient way of dealing with large complicated problems. The disadvantage is that there is no guarantee that the best solution found will be an optimal solution or even a nearly optimal solution. Therefore, whenever a problem can be solved



■ **FIGURE 14.3**
A typical sequence of objective function values for the solutions obtained by a metaheuristic as it first converges to a local optimum (iteration 4) and then escapes to converge to (hopefully) the global optimum (iteration 8) of a maximization problem before concluding its search (iteration 12).

by an algorithm that can guarantee optimality, that should be done instead. The role of metaheuristics is to deal with problems that are too large and complicated to be solved by exact algorithms. All the examples in this chapter are too small to require the use of metaheuristics, since they are intended only to illustrate in a straightforward way how metaheuristics can approach far more complicated problems.

Section 14.3 will illustrate the application of a particular metaheuristic to the nonlinear programming example depicted in Fig. 14.1. Section 14.4 then will apply another metaheuristic to the integer programming version of this same example.

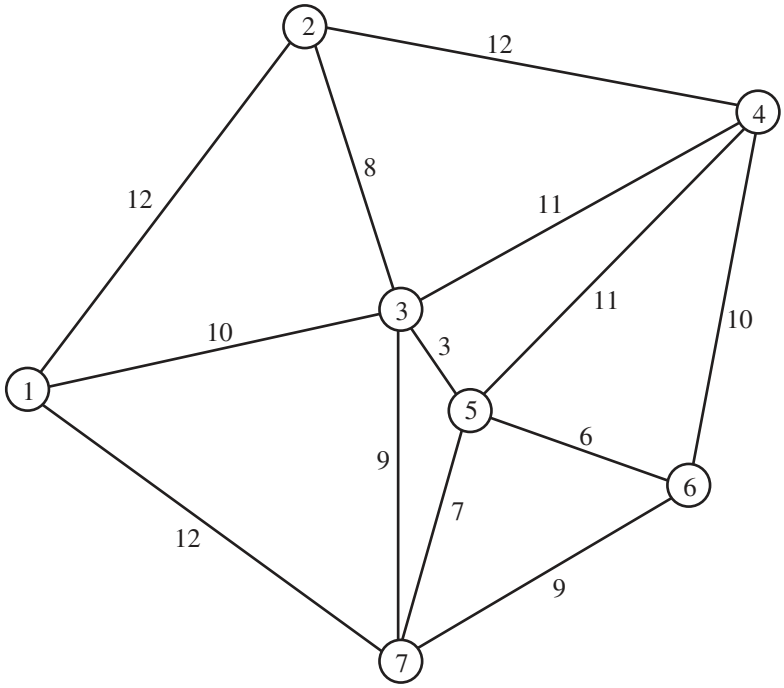
Although metaheuristics sometimes are applied to difficult nonlinear programming and integer programming problems, a more common area of application is to *combinatorial optimization* problems. Our next example is of this type.

An Example: A Traveling Salesman Problem

Perhaps the most famous classic combinatorial optimization problem is called the *traveling salesman problem*. It has been given this picturesque name because it can be described in terms of a salesman (or saleswoman) who must travel to a number of cities during one tour. Starting from his (or her) home city, the salesman wishes to determine which route to follow to visit each city exactly once before returning to his home city so as to minimize the total length of the tour.

Figure 14.4 shows an example of a small traveling salesman problem with seven cities. City 1 is the salesman's home city. Therefore, starting from this city, the salesman must choose a route to visit each of the other cities exactly once before returning to city 1. The number next to each link between each pair of cities represents the distance (or cost or time) between these cities. We assume that the distance is the same in either direction. (This is referred to as a *symmetric* traveling salesman problem.) Although there commonly is a direct link between every pair of cities, we are simplifying this example by assuming that the only direct links are those shown in the figure. The objective is to determine which route will minimize the total distance that the salesman must travel.

There have been a number of applications of traveling salesman problems that have nothing to do with salesmen. For example, when a truck leaves a distribution center to



■ **FIGURE 14.4**
The example of a traveling salesman problem that will be used for illustrative purposes throughout this chapter.

deliver goods to a number of locations, the problem of determining the shortest route for doing this is a traveling salesman problem. Another example involves the manufacture of printed circuit boards for wiring chips and other components. When many holes need to be drilled into a printed circuit board, the problem of finding the most efficient drilling sequence is a traveling salesman problem.

The difficulty of traveling salesman problems increases rapidly as the number of cities increases. For a problem with n cities and a link between every pair of cities, the number of feasible routes to be considered is $(n - 1)!/2$ since there are $(n - 1)$ possibilities for the first city after the home city, $(n - 2)$ possibilities for the next city, and so forth. The denominator of 2 arises because every route has an equivalent reverse route with exactly the same distance. Thus, while a 10-city traveling salesman problem has less than 200,000 feasible solutions to be considered, a 20-city problem has roughly 10^{16} feasible solutions, while a 50-city problem has about 10^{62}.

Surprisingly, powerful algorithms based on the branch-and-cut approach introduced in Sec. 12.8 have succeeded in solving to optimality certain huge traveling salesman problems with many hundreds (or even thousands) of cities. However, because of the enormous difficulty of solving large traveling salesman problems, heuristic methods guided by metaheuristics continue to be a popular way of addressing such problems.

These heuristic methods commonly involve generating a sequence of feasible trial solutions, where each new trial solution is obtained by making a certain type of small adjustment in the current trial solution. Several methods have been suggested for how to adjust the current trial solution. Because of its ease of implementation, one popular method uses the following type of adjustment.

A **sub-tour reversal** adjusts the sequence of cities visited in the current trial solution by selecting a subsequence of the cities and simply reversing the order in which that subsequence of cities is visited. (The subsequence being reversed can consist of as few as two cities, but also can have more.)

To illustrate a sub-tour reversal, suppose that the initial trial solution for our example in Fig. 14.4 is to visit the cities in numerical order:

1-2-3-4-5-6-7-1 Distance = 69

If we select, say, the subsequence 3-4 and reverse it, we obtain the following new trial solution:

1-2-4-3-5-6-7-1 Distance = 65

Thus, this particular sub-tour reversal has succeeded in reducing the distance for the complete tour from 69 to 65.

Figure 14.5 depicts this sub-tour reversal, which leads from the initial trial solution on the left to the new trial solution on the right. The dashed lines indicate the links that are deleted from the tour (on the left) or added to the tour (on the right) by sub-tour reversal. Note that the new trial solution deletes exactly two links from the previous tour and replaces them by exactly two new links to form the new tour. This is a characteristic of any sub-tour reversal (including those where the subsequence of cities being reversed consists of more than two cities). Thus, a particular sub-tour reversal is possible only if the corresponding two new links actually exist.

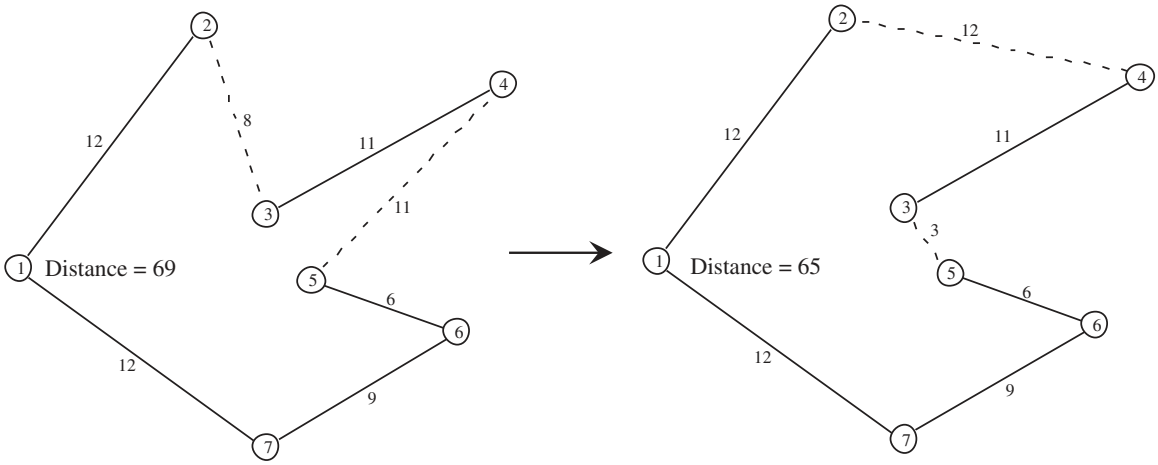
This success in obtaining an improved tour by simply performing a sub-tour reversal suggests the following heuristic method for seeking a good feasible solution for any traveling salesman problem.

The Sub-Tour Reversal Algorithm

Initialization. Start with any feasible tour as the initial trial solution.

Iteration. For the current trial solution, consider all possible ways of performing a sub-tour reversal (except exclude the reversal of the entire tour) that would provide an improved solution. Select the one that provides the largest decrease in the distance traveled to be the new trial solution. (Ties may be broken arbitrarily.)

■ **FIGURE 14.5**
A sub-tour reversal that replaces the tour on the left (the initial trial solution) by the tour on the right (the new trial solution) by reversing the order in which cities 3 and 4 are visited. This sub-tour reversal results in replacing the dashed lines on the left by the dashed lines on the right as the links that are traversed in the new tour.



Stopping rule. Stop when no sub-tour reversal will improve the current trial solution. Accept this solution as the final solution.

Now let us apply this algorithm to the example, starting with 1-2-3-4-5-6-7-1 as the initial trial solution. There are four possible sub-tour reversals that would improve upon this solution, as listed in the second, third, fourth, and fifth rows below:

	1-2-3-4-5-6-7-1	Distance = 69
Reverse 2-3:	1-3-2-4-5-6-7-1	Distance = 68
Reverse 3-4:	1-2-4-3-5-6-7-1	Distance = 65
Reverse 4-5:	1-2-3-5-4-6-7-1	Distance = 65
Reverse 5-6:	1-2-3-4-6-5-7-1	Distance = 66

The two solutions with Distance = 65 tie for providing the largest decrease in the distance traveled, so suppose that the first of these, 1-2-4-3-5-6-7-1 (as shown on the right side of Fig. 14.5), is chosen arbitrarily to be the next trial solution. This completes the first iteration.

The second iteration begins with the tour on the right side of Fig. 14.5 as the current trial solution. For this solution, there is only one sub-tour reversal that will provide an improvement, as listed in the second row below:

	1-2-4-3-5-6-7-1	Distance = 65
Reverse 3-5-6:	1-2-4-6-5-3-7-1	Distance = 64

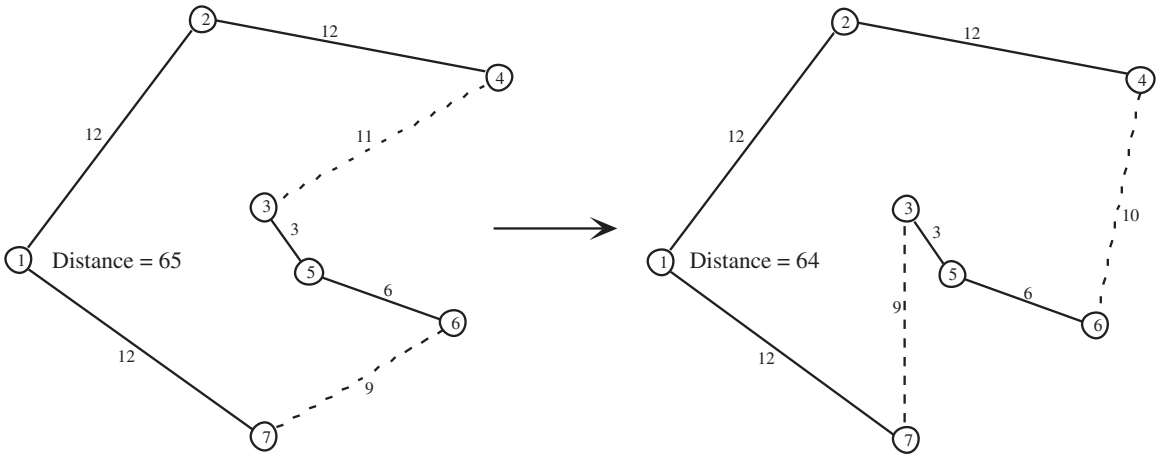
Figure 14.6 shows this sub-tour reversal, where the entire subsequence of cities 3-5-6 on the left now is visited in reverse order (6-5-3) on the right. Thus, the tour on the right now traverses the link 4-6 instead of 4-3, as well as the link 3-7 instead of 6-7, in order to use the reverse order 6-5-3 between cities 4 and 7. This completes the second iteration.

We next try to find a sub-tour reversal that will improve upon this new trial solution. However, there is none, so the sub-tour reversal algorithm stops with this trial solution as the final solution.

Is 1-2-4-6-5-3-7-1 the optimal solution? Unfortunately, no. The optimal solution turns out to be

1-2-4-6-7-5-3-1 Distance = 63

FIGURE 14.6
The sub-tour reversal of 3-5-6 that leads from the trial solution on the left to an improved trial solution on the right.



(or 1-3-5-7-6-4-2-1 by reversing the direction of this entire tour)

However, this solution cannot be reached by performing a sub-tour reversal that improves 1-2-4-6-5-3-7-1.

The sub-tour reversal algorithm is another example of a *local improvement procedure*. It improves upon the current trial solution at each iteration. When it can no longer find a better solution, it stops because the current trial solution is a local optimum. In this case, 1-2-4-6-5-3-7-1 is indeed a *local optimum* because there is no better solution within its local neighborhood that can be reached by performing a sub-tour reversal.

What is needed to provide a better chance of reaching a global optimum is to use a metaheuristic that will enable the process to escape from a local optimum. You will see how three different metaheuristics do this with this same example in the next three sections.

14.2 TABU SEARCH

Tabu search is a widely used metaheuristic that uses some common-sense ideas to enable the search process to escape from a local optimum. After introducing its basic concepts, we will go through a simple example and then return to the traveling salesman example.

Basic Concepts

Any application of tabu search includes as a subroutine a *local search procedure* that seems appropriate for the problem being addressed. (A **local search procedure** operates just like a local improvement procedure except that it may not require that each new trial solution must be better than the preceding trial solution.) The process begins by using this procedure as a local *improvement* procedure in the usual way (i.e., only accepting an improved solution at each iteration) to find a local optimum. A key strategy of tabu search is that it then continues the search by allowing *non-improving moves* to the best solutions in the neighborhood of the local optimum. Once a point is reached where better solutions can be found in the neighborhood of the current trial solution, the local improvement procedure is reapplied to find a new local optimum.

Using the analogy of hill climbing, this process is sometimes referred to as the **steepest ascent/mildest descent approach** because each iteration selects the available move that goes furthest up the hill, or, when an upward move is not available, selects a move that drops least down the hill. If all goes well, the process will follow a pattern like that shown in Fig. 14.3, where a local optimum is left behind in order to climb to the global optimum.

The danger with this approach is that after moving away from a local optimum, the process will cycle right back to the same local optimum. To avoid this, a tabu search temporarily forbids moves that would return to (or perhaps toward) a solution recently visited. A **tabu list** records these forbidden moves, which are referred to as *tabu moves*. (The only exception to forbidding such a move is if it is found that a tabu move actually is better than the best feasible solution found so far.)

This use of *memory* to guide the search by using tabu lists to record some of the recent history of the search is a distinctive feature of tabu search. This feature has roots in the field of artificial intelligence.

Tabu search also can incorporate some more advanced concepts. One is *intensification,* which involves exploring a portion of the feasible region more thoroughly than usual after it has been identified as a particularly promising portion for containing very good solutions. Another concept is *diversification,* which involves forcing the search into previously unexplored areas of the feasible region. (Long-term memory is used to help implement both concepts.) However, we will focus on the basic form of tabu search summarized next without delving into these additional concepts.

An Application Vignette

Founded in 1886, **Sears, Roebuck and Company** (now commonly referred to as just **Sears**) grew to become the largest multiline retailer in the United States by the mid-20th century. It continues today to rank among the largest retailers in the world selling merchandise and services. By 2013, it had more than 2,500 full-line and specialty retail stores in the United States and Canada. It also provides the largest home-delivery service of furniture and appliances in these countries with approximately 4 million deliveries a year. Sears manages a fleet of over 1,000 delivery vehicles that includes contract carriers and Sears-owned vehicles. It also operates a U.S. fleet of about 12,500 service vehicles and the associated technicians, who make approximately 14 million on-site service calls annually to repair and install appliances and provide home improvement.

The cost of operating this huge home-delivery and home-service business runs in the *billions of dollars per year*. With many thousands of vehicles being used to make many tens of thousands of calls on customers *daily*, the efficiency of this operation has a major impact on the company's profitability.

With so many calls on customers to be made with so many vehicles, a huge number of decisions must be made *each* day. Which stops should be assigned to each vehicle's route? What should the order of the stops be (which considerably impacts the total distance and time for the route) for each vehicle? How can all these decisions be made so as to minimize total operational costs while providing satisfactory service to the customers?

It became clear that operations research was needed to address this problem. The natural formulation is as a *vehicle-routing problem with time windows* (VRPTW), for which both exact and heuristic algorithms have been developed. Unfortunately, the Sears problem is so huge that it is a very difficult combinatorial optimization problem that is beyond the reach of standard algorithms for VRPTW. Therefore, a new algorithm was developed that was based on using *tabu search* for making both the decisions on which vehicle's route serves which stops and what the sequence is of stops within a route.

The resulting new vehicle-routing-and-scheduling system, based largely on tabu search, led to *over* **$9 million** *in one-time savings* and *over* **$42 million** in *annual savings* for Sears. It also provided a number of intangible benefits, including (most importantly) *improved service to customers*.

Source: D. Weigel, and B. Cao: "Applying GIS and OR Techniques to Solve Sears Technician-Dispatching and Home-Delivery Problems," *Interfaces*, **29**(1): 112–130, Jan.–Feb. 1999. (A link to this article is provided on our website, www.mhhe.com/hillier.)

Outline of a Basic Tabu Search Algorithm

Initialization. Start with a feasible initial trial solution.

Iteration. Use an appropriate local search procedure to define the feasible moves into the local neighborhood of the current trial solution. Eliminate from consideration any move on the current tabu list unless that move would result in a better solution than the best trial solution found so far. Determine which of the remaining moves provides the best solution. Adopt this solution as the next trial solution, regardless of whether it is better or worse than the current trial solution. Update the tabu list to forbid cycling back to what had been the current trial solution. If the tabu list already had been full, delete the oldest member of the tabu list to provide more flexibility for future moves.

Stopping rule. Use some stopping criterion, such as a fixed number of iterations, a fixed amount of CPU time, or a fixed number of consecutive iterations without an improvement in the best objective function value. (The latter criterion is a particularly popular one.) Also stop at any iteration where there are no feasible moves into the local neighborhood of the current trial solution. Accept the best trial solution found on any iteration as the final solution.

This outline leaves a number of questions unanswered:

1. Which local search procedure should be used?
2. How should that procedure define the *neighborhood structure* that specifies which solutions are immediate neighbors (reachable in a single iteration) of any current trial solution?
3. What is the form in which tabu moves should be represented on the tabu list?
4. Which tabu move should be added to the tabu list in each iteration?
5. How long should a tabu move be retained on the tabu list?
6. Which stopping rule should be used?

These all are important details that need to be worked out to fit the specific type of problem being addressed, as illustrated by the following examples. Tabu search only provides a general structure and strategy guidelines for developing a specific heuristic method to fit a specific situation. The selection of its parameters is a key part of developing a successful heuristic method.

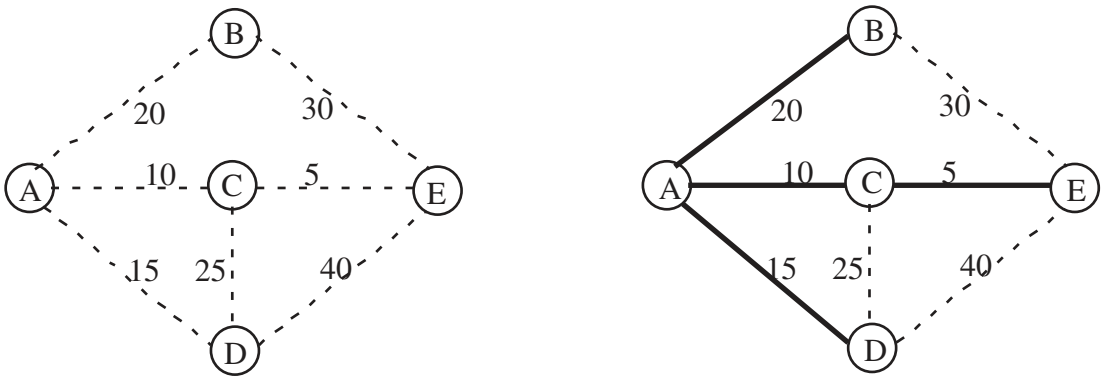
The following examples illustrate the use of tabu search.

A Minimum Spanning Tree Problem with Constraints

Section 10.4 describes the minimum spanning tree problem. In brief, starting with a network that has its nodes but no links between the nodes yet, the problem is to determine which links should be inserted into the network. The objective is to minimize the total cost (or length) of the inserted links that will provide a path between every pair of nodes. For a network with n nodes, $(n - 1)$ links (with no cycles) are needed to provide a path between every pair of nodes. Such a network is referred to as a *spanning tree*.

The left-hand side of Fig. 14.7 shows a network with five nodes, where the dashed lines represent the potential links that could be inserted into the network and the number next to each dashed line represents the cost associated with inserting that particular link. Thus, the problem is to determine which four of these links (with no cycles) should be inserted into the network to minimize the total cost of these links. The right-hand side of the figure shows the desired *minimum spanning tree,* where the dark lines represent the links

■ **FIGURE 14.7**
(*a*) The data for a minimum spanning tree problem before choosing the links to be included in the network and (*b*) the optimal solution for this problem where the dark lines represent the chosen links.



that have been inserted into the network with a total cost of 50. This optimal solution is obtained easily by applying the "greedy" algorithm presented in Sec. 10.4.

To illustrate the use of tabu search, let us now add a couple complications to this example by supposing that the following constraints also must be observed when choosing the links to include in the network.

Constraint 1: Link AD can be included only if link DE also is included.
Constraint 2: At most one of the three links—AD, CD, and AB—can be included.

Note that the previously optimal solution on the right-hand side of Fig. 14.7 violates both of these constraints because (1) link AD is included even though DE is not and (2) both AD and AB are included.

By imposing such constraints, the greedy algorithm presented in Sec. 10.4 can no longer be used to find the new optimal solution. For such a small problem, this solution probably could be found rather quickly by inspection. However, let us see how tabu search could be used on either this problem or much larger problems to search for an optimal solution.

The easiest way to take the constraints into account is to charge a huge penalty, such as the following, for violating them:

1. Charge a penalty of 100 if constraint 1 is violated.
2. Charge a penalty of 100 if two of the three links specified in constraint 2 are included. Increase this penalty to 200 if all three of the links are included.

A penalty of 100 is large enough to ensure that the constraints will not be violated for a spanning tree that minimizes the total cost, including the penalty, provided only that there exist some feasible solutions. Doubling this penalty if constraint 2 is badly violated provides an incentive for at least reducing how many of the three links are included during an iteration of the tabu search.

There are a variety of ways to answer the six questions that are needed to specify how the tabu search will be conducted. (See the list of questions that follows the outline of a basic tabu search algorithm.) Here is one straightforward way of answering the questions.

1. **Local search procedure:** At each iteration, choose the best immediate neighbor of the current trial solution that is not ruled out by its tabu status.
2. **Neighborhood structure:** An immediate neighbor of the current trial solution is one that is reached by adding a single link and then deleting one of the other links in the cycle that is formed by the addition of this link. (The deleted link must come from this cycle in order to still have a spanning tree.)
3. **Form of tabu moves:** List the links that should not be deleted.
4. **Addition of a tabu move:** At each iteration, after choosing the link to be added to the network, also add this link to the tabu list.
5. **Maximum size of tabu list:** Two. Whenever a tabu move is added to a full list, delete the older of the two tabu moves that already were on the list. (Since a spanning tree for the problem being considered only includes four links, the tabu list must be kept very small to provide some flexibility in choosing the link to be deleted at each iteration.)
6. **Stopping rule:** Stop after three consecutive iterations without an improvement in the best objective function value. (Also stop at any iteration where the current trial solution has no immediate neighbors that are not ruled out by their tabu status.)

Having specified these details, we now can proceed to apply the tabu search algorithm to the example. To get started, a reasonable choice for the initial trial solution is the optimal solution for the unconstrained version of the problem that is shown in Fig. 14.7(*b*).

Because this solution violates both of the constraints (but with the inclusion of only two of the three links specified in constraint 2), penalties of 100 need to be imposed twice. Therefore, the total cost of this solution is

$$\begin{aligned}\text{Cost} &= 20 + 10 + 5 + 15 + 200 \text{ (constraint penalties)} \\ &= 250.\end{aligned}$$

Iteration 1. The three options for adding a link to the network in Fig. 14.7(*b*) are BE, CD, and DE. If BE were to be chosen, the cycle formed would be BE-CE-AC-AB, so the three options for deleting a link would be CE, AC, and AB. (At this point, no links have yet been added to the tabu list.) If CE were to be deleted, the change in the cost would be $30 - 5 = 25$ with no change in the constraint penalties, so the total cost would increase from 250 to 275. Similarly, if AC were to be deleted instead, the total cost would increase from 250 to $250 + (30 - 10) = 270$. However, if link AB were to be the one deleted, the link costs would change by $30 - 20 = 10$ and the constraint penalties would decrease from 200 to 100 because constraint 2 would no longer be violated, so the total cost would become $50 + 10 + 100 = 160$. These results are summarized in the first three rows of Table 14.1.

The next two rows summarize the calculations if CD were to be the link that is added to the network. In this case, the cycle created is CD-AD-AC, so AD and AC are the only options for deleting a link. AC would be a particularly bad choice because constraint 1 would still be violated (a penalty of 100), and a penalty of 200 now would need to be charged for violating constraint 2 since all three of the links specified in the constraint would be included in the network. Deleting AD instead would have the virtue of satisfying constraint 1 and not increasing the extent to which constraint 2 is violated.

The last three rows of the table show the options if DE were the added link. The cycle created by adding this link would be DE-CE-AC-AD, so CE, AC, and AD would be the options for deletion. All three would satisfy constraint 1, but deleting AD would satisfy constraint 2 as well. By completely eliminating constraint penalties, the total cost for this option would become only $50 + (40 - 15) = 75$. Since this is the smallest cost for all eight available options for moving to an immediate neighbor of the current trial solution, we choose this particular move by adding DE and deleting AD. This choice is indicated in the iteration 1 portion of Fig. 14.8 and the resulting spanning tree for beginning iteration 2 is shown to the right.

To complete the iteration, since DE was added to the network, it becomes the first link placed on the tabu list. This will prevent deleting DE next and cycling back to the trial solution that began this iteration.

TABLE 14.1 The options for adding a link and deleting another link in iteration 1

Add	Delete	Cost
BE	CE	75 + 200 = 275
BE	AC	70 + 200 = 270
BE	AB	60 + 100 = 160
CD	AD	60 + 100 = 160
CD	AC	65 + 300 = 365
DE	CE	85 + 100 = 185
DE	AC	80 + 100 = 180
DE	AD	75 + 0 = 75 ← Minimum

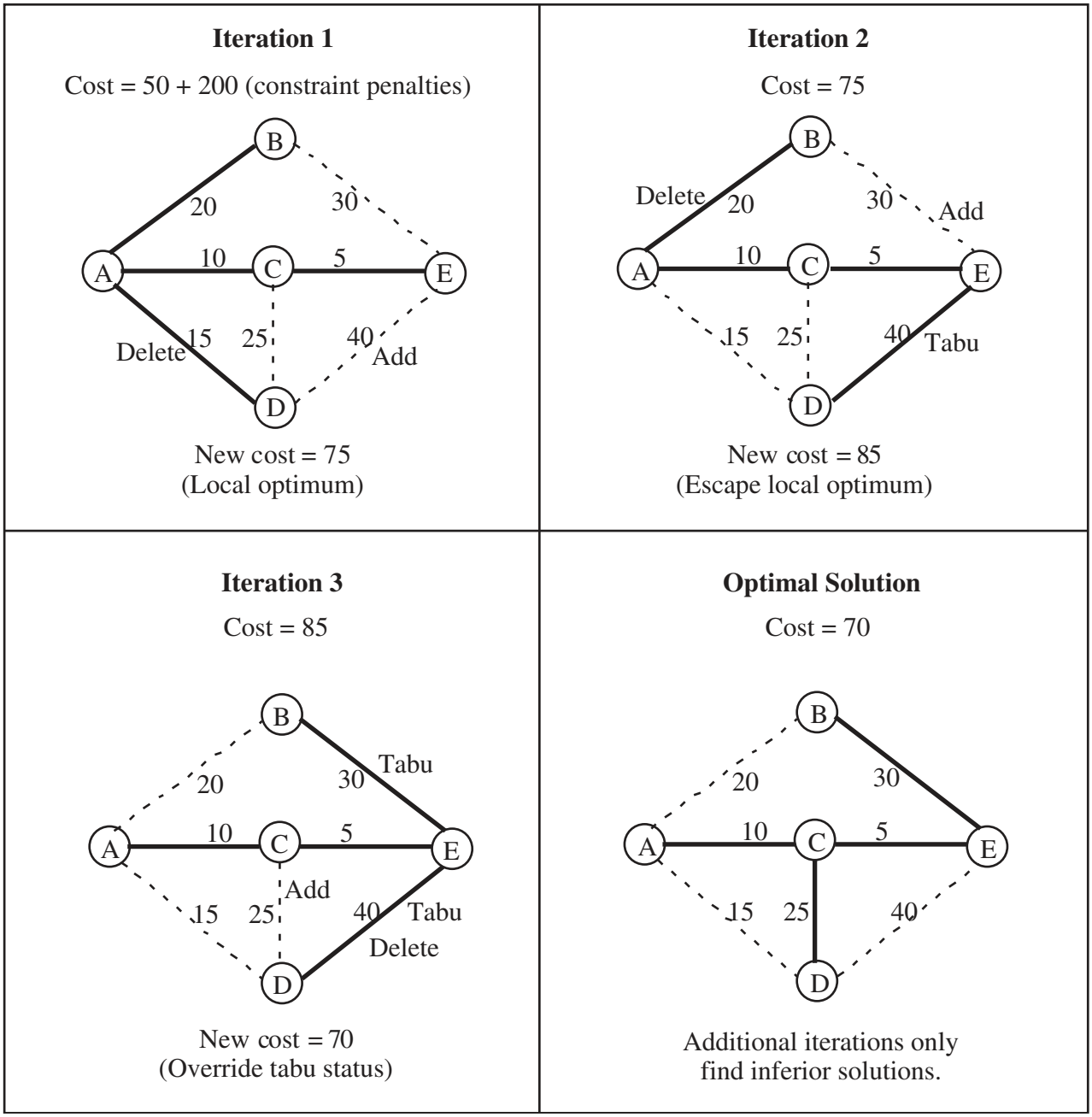


FIGURE 14.8
Application of a tabu search algorithm to the minimum spanning tree problem shown in Fig. 14.7 after also adding two constraints.

To summarize, the following decisions have been made during this first iteration:

Add link DE to the network.
Delete link AD from the network.
Add link DE to the tabu list.

Iteration 2. The upper right-hand portion of Fig. 14.8 indicates that the corresponding decisions made during iteration 2 are the following:

Add link BE to the network.
Automatically place this added link on the tabu list.
Delete link AB from the network.

Table 14.2 summarizes the calculations that led to these decisions by finding that the move in the sixth row provides the smallest cost.

The moves listed in the first and seventh rows of the table involve deleting DE, which is on the tabu list. Therefore, these moves would have been considered only if they would result in a better solution than the best trial solution found so far, which has a cost of 75. The calculation in the seventh row shows that this move would not provide a better solution. A calculation is not even needed for the first row because this move would cycle back to the preceding trial solution.

Note that the move in the sixth row is made even though it results in a new trial solution that has a larger cost (85) than for the preceding trial solution (75) that initiated iteration 2. What this means is that the preceding trial solution was a local optimum because all of its immediate neighbors (those that can be reached by making one of the moves listed in Table 14.2) have a larger cost. However, moving to the best of the immediate neighbors allows us to escape the local optimum and continue the search for the global optimum.

Before moving to iteration 3, we should interject an observation about what more advanced forms of tabu search might do here when selecting the best immediate neighbor. More general tabu search methods can change the meaning of a "best neighbor," depending on history, by using additional forms of memory to support intensification and diversification processes. As mentioned earlier, intensification focuses the search in a particularly promising region of solutions identified previously and diversification drives the search into promising new regions.

Iteration 3. The lower left-hand portion of Fig. 14.8 summarizes the decisions made during iteration 3.

Add link CD to the network.
Automatically place this added link on the tabu list.
Delete link DE from the network.

Table 14.3 shows that this move leads to the best immediate neighbor of the trial solution that initiated this iteration.

TABLE 14.2 The options for adding a link and deleting another link in iteration 2

Add	Delete	Cost
AD	DE*	(Tabu move)
AD	CE	85 + 100 = 185
AD	AC	80 + 100 = 180
BE	CE	100 + 0 = 100
BE	AC	95 + 0 = 95
BE	AB	85 + 0 = 85 ← Minimum
CD	DE*	60 + 100 = 160
CD	CE	95 + 100 = 195

*A tabu move. Will be considered only if it would result in a better solution than the best trial solution found previously.

TABLE 14.3 The options for adding a link and deleting another link in iteration 3

Add	Delete	Cost
AB	BE*	(Tabu move)
AB	CE	100 + 0 = 100
AB	AC	95 + 0 = 95
AD	DE*	60 + 100 = 160
AD	CE	95 + 0 = 95
AD	AC	90 + 0 = 90
CD	DE*	70 + 0 = 70 ← Minimum
CD	CE	105 + 0 = 105

*A tabu move. Will be considered only if it would result in a better solution than the best trial solution found previously.

An interesting feature of this move is that it is made even though it is a tabu move. The reason it is made is that, in addition to being the best immediate neighbor, it also results in a solution that is better (a cost of 70) than the best trial solution found previously (a cost of 75). This enables the tabu status of the move to be overridden. (Tabu search also can incorporate a variety of more advanced criteria for overriding tabu status.)

One more adjustment needs to be made in the tabu list before beginning the next iteration:

Delete link DE from the tabu list.

This is done for two reasons. First, the tabu list consists of links that normally should not be deleted from the network during the current iteration (with the exception noted above), but DE is no longer in the network. Second, since the size of the tabu list has been set at two and two other links (BE and CD) have been added to the list more recently, DE automatically would have been deleted from the list at this point anyway.

Continuation. The current trial solution shown in the lower right-hand portion of Fig. 14.8 is, in fact, the optimal solution (the global optimum) for the problem. However, the tabu search algorithm has no way of knowing this, so it would continue on for a while. Iteration 4 would begin with this trial solution and with links BE and CD on the tabu list. After completing this iteration and two more, the algorithm would terminate because three consecutive iterations did not improve on the best previous objective function value (a cost of 70).

With a well-designed tabu search algorithm, the best trial solution found after the algorithm has run a modest number of iterations is likely to be a good feasible solution. It might even be an optimal solution, but no such guarantee can be given. Selecting a stopping rule that provides a relatively long run of the algorithm increases the chance of reaching the global optimum.

Having gotten our feet wet by designing and applying a tabu search algorithm to this small example, let us now apply a similar tabu search algorithm to the example of a traveling salesman problem presented in Sec. 14.1.

The Traveling Salesman Problem Example

There are some close parallels between a minimum spanning tree problem and a traveling salesman problem. In both cases, the problem is to choose which links to include in the solution. (Recall that a solution for a traveling salesman problem can be described

as the sequence of links that the salesman traverses in the tour of the cities.) In both cases, the objective is to minimize the total cost or distance associated with the fixed number of links that are included in the solution. And in both cases, there is an intuitive local search procedure available that involves adding and deleting links in the current trial solution to obtain the new trial solution.

For minimum spanning tree problems, the local search procedure described in the preceding subsection involves adding and deleting only a *single* link at each iteration. The corresponding procedure described in Sec. 14.1 for traveling salesman problems involves using *sub-tour reversals* to add and delete a *pair* of links at each iteration.

Because of the close parallels between these two types of problems, the design of a tabu search algorithm for traveling salesman problems can be quite similar to the one just described for the minimum spanning problem example. In particular, using the outline of a basic tabu search algorithm presented earlier, the six questions following the outline can be answered in a similar way below.

1. **Local search algorithm:** At each iteration, choose the best immediate neighbor of the current trial solution that is not ruled out by its tabu status.
2. **Neighborhood structure:** An immediate neighbor of the current trial solution is one that is reached by making a *sub-tour reversal,* as described in Sec. 14.1 and illustrated in Fig. 14.5. Such a reversal requires adding two links and deleting two other links from the current trial solution. (We rule out a sub-tour reversal that simply reverses the direction of the tour provided by the current trial solution.)
3. **Form of tabu moves:** List the links such that a particular sub-tour reversal would be tabu if *both* links to be deleted in this reversal are on the list. (This will prevent quickly cycling back to a previous trial solution.)
4. **Addition of a tabu move:** At each iteration, after choosing the two links to be added to the current trial solution, also add these two links to the tabu list.
5. **Maximum size of tabu list:** Four (two from each of the two most recent iterations). Whenever a pair of links is added to a full list, delete the two links that already have been on the list the longest.
6. **Stopping rule:** Stop after three consecutive iterations without an improvement in the best objective function value. (Also stop at any iteration where the current trial solution has no immediate neighbors that are not ruled out by their tabu status.)

To apply this tabu search algorithm to our example (see Fig. 14.4), let us begin with the same initial trial solution, 1-2-3-4-5-6-7-1, as in Sec. 14.1. Recall how starting the sub-tour reversal algorithm (a local improvement algorithm) with this initial trial solution led in two iterations (see Figs. 14.5 and 14.6) to a local optimum at 1-2-4-6-5-3-7-1, at which point that algorithm stopped. Except for adding a tabu list, the tabu search algorithm starts off in exactly the same way, as summarized below:

Initial trial solution: 1-2-3-4-5-6-7-1 Distance = 69
Tabu list: Blank at this point.

Iteration 1: Choose to reverse 3-4 (see Fig. 14.5).
Deleted links: 2-3 and 4-5
Added links: 2-4 and 3-5
Tabu list: Links 2-4 and 3-5
New trial solution: 1-2-4-3-5-6-7-1 Distance = 65

Iteration 2: Choose to reverse 3-5-6 (see Fig. 14.6).
Deleted links: 4-3 and 6-7 (OK since not on tabu list)
Added links: 4-6 and 3-7

Tabu list: Links 2-4, 3-5, 4-6, and 3-7
New trial solution: 1-2-4-6-5-3-7-1 Distance = 64

However, rather than terminating, the tabu search algorithm now escapes from this local optimum (shown on the right side of Fig. 14.6 and the left side of Fig. 14.9) by moving next to the best immediate neighbor of the current trial solution even though its distance is longer. Considering the limited availability of links between pairs of nodes (cities) in Fig. 14.4, the current trial solution has only the two immediate neighbors listed below:

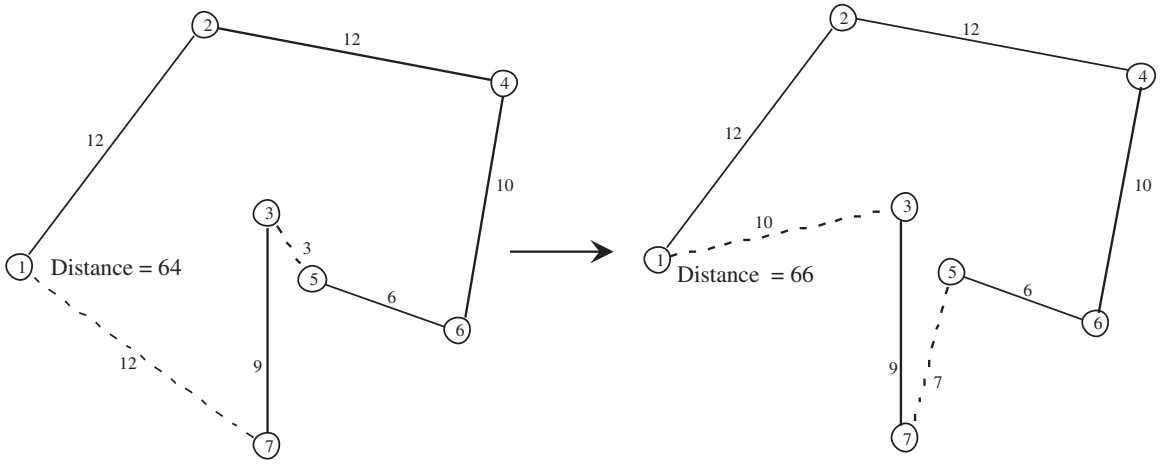
Reverse 6-5-3: 1-2-4-3-5-6-7-1 Distance = 65
Reverse 3-7: 1-2-4-6-5-7-3-1 Distance = 66

(We are ruling out reversing 2-4-6-5-3-7 to obtain 1-7-3-5-6-4-2-1 because this is simply the same tour in the opposite direction.) However, we must rule out the first of these immediate neighbors because it would require deleting links 4-6 and 3-7, which is tabu since *both* of these links are on the tabu list. (This move could still be allowed if it would improve upon the best trial solution found so far, but it does not.) Ruling out this immediate neighbor prevents us from simply cycling back to the preceding trial solution. Therefore, by default, the second of these immediate neighbors is chosen to be the next trial solution, as summarized below:

Iteration 3: Choose to reverse 3-7 (see Fig. 14.9).
Deleted links: 5-3 and 7-1
Added links: 5-7 and 3-1
Tabu list: 4-6, 3-7, 5-7, and 3-1
(2-4 and 3-5 are now deleted from the list.)
New trial solution: 1-2-4-6-5-7-3-1 Distance = 66

The sub-tour reversal for this iteration can be seen in Fig. 14.9, where the dashed lines show the links being deleted (on the left) and added (on the right) to obtain the new trial solution. Note that one of the deleted links is 5-3 even though it was on the tabu list at the end of iteration 2. This is OK since a sub-tour reversal is tabu only if *both* of the deleted links are on the tabu list. Also note that the updated tabu list at the end of iteration 3 has

FIGURE 14.9
The sub-tour reversal of 3-7 in iteration 3 that leads from the trial solution on the left to the new trial solution on the right.



deleted the two links that had been on the list the longest (the ones added during iteration 1) since the maximum size of the tabu list has been set at four.

The new trial solution has the four immediate neighbors listed below:

Reverse 2-4-6-5-7: 1-7-5-6-4-2-3-1 Distance = 65
Reverse 6-5: 1-2-4-5-6-7-3-1 Distance = 69
Reverse 5-7: 1-2-4-6-5-7-3-1 Distance = 63
Reverse 7-3: 1-2-4-6-5-3-7-1 Distance = 64

However, the second of these immediate neighbors is tabu because *both* of the deleted links (4-6 and 5-7) are on the tabu list. The fourth immediate neighbor (which is the preceding trial solution) also is tabu for the same reason. Thus, the only viable options are the first and third immediate neighbors. Since the latter neighbor has the shorter distance, it becomes the next trial solution, as summarized below:

Iteration 4: Choose to reverse 5-7 (see Fig. 14.10).
Deleted links: 6-5 and 7-3
Added links: 6-7 and 5-3
Tabu list: 5-7, 3-1, 6-7, and 5-3
(4-6 and 3-7 are now deleted from the list.)
New trial solution: 1-2-4-6-7-5-3-1 Distance = 63

Figure 14.10 shows this sub-tour reversal. The tour for the new trial solution on the right has a distance of only 63, which is less than for any of the preceding trial solutions. In fact, this new solution happens to be the optimal solution.

Not knowing this, the tabu search algorithm would attempt to execute more iterations. However, the only immediate neighbor of the current trial solution is the trial solution that was obtained at the preceding iteration. This would require deleting links 6-7 and 5-3, both of which are on the tabu list, so we are prevented from cycling back to the preceding trial solution. Since no other immediate neighbors are available, the stopping rule terminates the algorithm at this point with 1-2-4-6-7-5-3-1 (the best of the trial solutions) as the final solution. Although there is no guarantee that the algorithm's final solution is an optimal solution, we are fortunate that it turned out to be optimal in this case.

The metaheuristics area in your IOR Tutorial includes a procedure for applying this particular tabu search algorithm to other small traveling salesman problems.

This particular algorithm is just one example of a possible tabu search algorithm for traveling salesman problems. Various details of the algorithm could be modified in a number of reasonable ways. For example, the method typically doesn't stop when all available moves are forbidden by their tabu status, but instead just selects a "least tabu" move. Also, an important feature of general tabu search methods includes the use of multiple neighborhoods, relying on basic neighborhoods as long as they bring progress, and then including more advanced neighborhoods when the rate of finding improved solutions diminishes. The most significant additional element of tabu search is its use of intensification and diversification strategies, as mentioned earlier. But the general outline of a basic "short-term memory" tabu search approach would remain roughly the same as we have illustrated.

Both examples considered in this section fall into the category of combinatorial optimization problems involving networks. This is a particularly common area of application for tabu search algorithms. The general outline of these algorithms incorporates the principles presented in this section, but the details are worked out to fit the structure of the specific problems being considered.

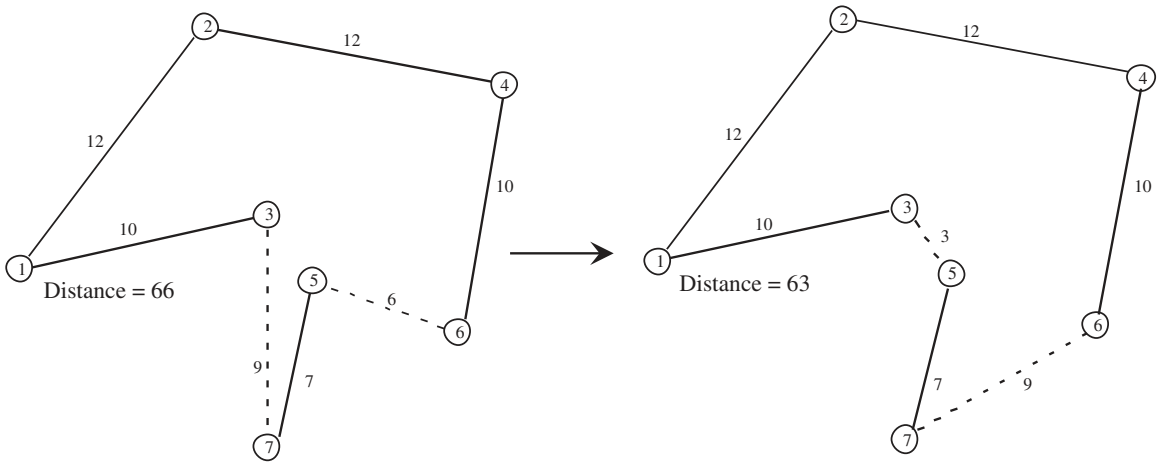


FIGURE 14.10
The sub-tour reversal of 5-7 in iteration 4 that leads from the trial solution on the left to the new trial solution on the right (which happens to be the optimal solution).

14.3 SIMULATED ANNEALING

Simulated annealing is another widely used metaheuristic that enables the search process to escape from a local optimum. To better compare and contrast it with tabu search, we will apply it to the same traveling salesman problem example before returning to the nonlinear programming example introduced in Sec. 14.1. But first, let us examine the basic concepts of simulated annealing.

Basic Concepts

Figure 14.1 in Sec. 14.1 introduced the concept that finding the global optimum of a complicated maximization problem is analogous to determining which of a number of hills is the tallest hill and then climbing to the top of that particular hill. Unfortunately, a mathematical search process does not have the benefit of keen eyesight that would enable spotting a tall hill in the distance. Instead, it is like hiking in a dense fog where the only clue for the direction to take next is how much the next step in any direction would take you up or down.

One approach, adopted into tabu search, is to climb the current hill in the steepest direction until reaching its top and then start climbing slowly downward while searching for another hill to climb. The drawback is that a lot of time (iterations) is spent climbing each hill encountered rather than searching for the tallest hill.

Instead, the approach used in simulated annealing is to focus mainly on searching for the tallest hill. Since the tallest hill can be anywhere in the feasible region, the early emphasis is on taking steps in random directions (except for rejecting some, but not all, steps that would go downward rather than upward) in order to explore as much of the feasible region as possible. Because most of the accepted steps are upward, the search will gradually gravitate toward those parts of the feasible region containing the tallest hills. Therefore, the search process gradually increases the emphasis on climbing upward by rejecting an increasing proportion of steps that go downward. Given enough time, the process often will reach and climb to the top of the tallest hill.

To be more specific, each iteration of the simulated annealing search process moves from the current trial solution to an immediate neighbor in the local neighborhood of this

solution, just as for tabu search. However, the difference from tabu search lies in how an immediate neighbor is selected to be the next trial solution. Let

Z_c = objective function value for the *current* trial solution,
Z_n = objective function value for the current candidate to be the next trial solution,
T = a parameter that measures the tendency to accept the current candidate to be the next trial solution if this candidate is not an improvement on the current trial solution.

The rule for selecting which immediate neighbor will be the next trial solution follows:

Move selection rule: Among all the immediate neighbors of the current trial solution, select one randomly to become the current candidate to be the next trial solution. Assuming the objective is *maximization* of the objective function, accept or reject this candidate to be the next trial solution as follows:

If $Z_n \geq Z_c$, always accept this candidate.
If $Z_n < Z_c$, accept the candidate with the following probability:

$$\text{Prob\{acceptance\}} = e^x \text{ where } x = \frac{Z_n - Z_c}{T}$$

(If the objective is *minimization* instead, reverse Z_n and Z_c in the above formulas.) If this candidate is rejected, repeat this process with a new randomly selected immediate neighbor of the current trial solution. (If no immediate neighbors remain, terminate the algorithm.)

Thus, if the current candidate under consideration is better than the current trial solution, it always is accepted to be the next trial solution. If it is worse, the probability of acceptance depends on how much worse it is (and on the size of T). Table 14.4 shows a sampling of these probability values, ranging from a very high probability when the current candidate is only slightly worse (relative to T) than the current trial solution to an extremely small probability when it is much worse. In other words, the move selection rule usually will accept a step that is only slightly downhill, but seldom will accept a steep downward step. Starting with a relatively large value of T (as simulated annealing does) makes the probability of acceptance relatively large, which enables the search to proceed in almost random directions. Gradually decreasing the value of T as the search continues (as simulated annealing does) gradually decreases the probability of acceptance, which increases the emphasis on mostly climbing upward. Thus, the choice of the values of T over time controls the degree of randomness in the process for allowing downward steps.

TABLE 14.4 Some sample probabilities that the move selection rule will accept a downward step when the objective is maximization

$x = \frac{Z_n - Z_c}{T}$	Prob{acceptance} = e^x
−0.01	0.990
−0.1	0.905
−0.25	0.779
−0.5	0.607
−1	0.368
−2	0.135
−3	0.050
−4	0.018
−5	0.007

This random component, not present in basic tabu search, provides more flexibility for moving toward another part of the feasible region in the hope of finding a taller hill.

The usual method of implementing the move selection rule to determine whether a particular downward step will be accepted is to compare a **random number** between 0 and 1 to the probability of acceptance. Such a random number can be thought of as a random observation from a uniform distribution between 0 and 1. (All references to random numbers throughout the chapter will be to such random numbers.) There are a number of methods of generating these random numbers (as will be described in Sec. 20.3). For example, the Excel function RAND() generates such random numbers upon request. (The beginning of the Problems section also describes how you can use the random digits given in Table 20.3 to obtain the random numbers you will need for some of your homework problems.) After generating a random number, it is used as follows to determine whether to accept a downward step:

> If *random number* $<$ Prob{acceptance}, accept a downward step.
> Otherwise, reject the step.

Why does simulated annealing use the particular formula for Prob{acceptance} specified by the move selection rule? The reason is that simulated annealing is based on the analogy to a *physical annealing process*. This process initially involves melting a metal or glass at a high temperature and then slowly cooling the substance until it reaches a low-energy stable state with desirable physical properties. At any given temperature T during this process, the energy level of the atoms in the substance is fluctuating but tending to decrease. A mathematical model of how the energy level fluctuates assumes that changes occur randomly except that only some of the increases are accepted. In particular, the probability of accepting an increase when the temperature is T has the same form as for Prob{acceptance} in the move selection rule for simulated annealing.

The analogy for an optimization problem in minimization form is that the energy level of the substance at the current state of the system corresponds to the objective function value at the current feasible solution of the problem. The objective of having the substance reach a stable state with an energy level that is as small as possible corresponds to having the problem reach a feasible solution with an objective function value that is as small as possible.

Just as for a physical annealing process, a key question when designing a simulated annealing algorithm for an optimization problem is to select an appropriate **temperature schedule** to use. (Because of the analogy to physical annealing, we now are referring to T in a simulated annealing algorithm as the temperature.) This schedule needs to specify the initial, relatively large value of T, as well as the subsequent progressively smaller values. It also needs to specify how many moves (iterations) should be made at each value of T. The selection of these parameters to fit the problem under consideration is a key factor in the effectiveness of the algorithm. Some preliminary experimentation can be used to guide this selection of the parameters of the algorithm. We later will specify one specific temperature schedule that seems reasonable for the two examples considered in this section, but many others could be considered as well.

With this background, we now can provide an outline of a basic simulated annealing algorithm.

Outline of a Basic Simulated Annealing Algorithm

Initialization. Start with a feasible initial trial solution.

Iteration. Use the *move selection rule* to select the next trial solution. (If none of the immediate neighbors of the current trial solution are accepted, the algorithm is terminated.)

Check the temperature schedule. When the desired number of iterations have been performed at the current value of T, decrease T to the next value in the temperature schedule and resume performing iterations at this next value.

Stopping rule. When the desired number of iterations have been performed at the smallest value of T in the temperature schedule (or when none of the immediate neighbors of the current trial solution are accepted), stop. Accept the best trial solution found at any iteration (including for larger values of T) as the final solution.

Before applying this algorithm to any particular problem, a number of details need to be worked out to fit the structure of the problem.

1. How should the initial trial solution be selected?
2. What is the *neighborhood structure* that specifies which solutions are immediate neighbors (reachable in a single iteration) of any current trial solution?
3. What device should be used in the move selection rule to *randomly* select one of the immediate neighbors of the current trial solution to become the current candidate to be the next trial solution?
4. What is an appropriate temperature schedule?

We will illustrate some reasonable ways of addressing these questions in the context of applying the simulated annealing algorithm to the following two examples.

The Traveling Salesman Problem Example

We now return to the particular traveling salesman problem that was introduced in Sec. 14.1 and displayed in Fig. 14.4.

The metaheuristics area in your IOR Tutorial includes a procedure for applying the basic simulated annealing algorithm to small traveling salesman problems like this example. This procedure answers the four questions in the following way:

1. **Initial trial solution:** You may enter any feasible solution (sequence of cities on the tour), perhaps by randomly generating the sequence, but it is helpful to enter one that appears to be a good feasible solution. For the example, the feasible solution 1-2-3-4-5-6-7-1 is a reasonable choice.
2. **Neighborhood structure:** An immediate neighbor of the current trial solution is one that is reached by making a *sub-tour reversal,* as described in Sec. 14.1 and illustrated in Fig. 14.5. (However, the sub-tour reversal that simply reverses the direction of the tour provided by the current trial solution is ruled out.)
3. **Random selection of an immediate neighbor:** Selecting a sub-tour to be reversed requires selecting the slot in the current sequence of cities where the sub-tour currently begins and then the slot where the sub-tour currently ends. The beginning slot can be anywhere except the first and last slots (reserved for the home city) and the next-to-last slot. The ending slot must be somewhere after the beginning slot, excluding the last slot. (Both beginning in the second slot and ending in the next-to-last slot also is ruled out since this would simply reverse the direction of the tour.) As will be illustrated shortly, random numbers are used to give equal probabilities to selecting any of the eligible beginning slots and then any of the eligible ending slots. If this selection of the beginning and ending slots turns out to be infeasible (because the links needed to complete the sub-tour reversal are not available), this process is repeated until a feasible selection is made.
4. **Temperature schedule:** Five iterations are performed at each of five values of T (T_1, T_2, T_3, T_4, T_5) in turn, where

$$T_1 = 0.2Z_c \text{ when } Z_c \text{ is the objective function value for the initial trial solution,}$$
$$T_2 = 0.5T_1,$$
$$T_3 = 0.5T_2,$$
$$T_4 = 0.5T_3,$$
$$T_5 = 0.5T_4.$$

This particular temperature schedule is only illustrative of what could be used. $T_1 = 0.2Z_c$ is a reasonable choice because T_1 should tend to be fairly large compared to typical values of $|Z_n - Z_c|$, which will encourage an almost random search through the feasible region to find where the search should be focused. However, by the time the value of T is reduced to T_5, almost no nonimproving moves will be accepted, so the emphasis will be on improving the value of the objective function.

When dealing with larger problems, more than five iterations probably would be performed at each value of T. Furthermore, the values of T would probably be reduced more slowly than with the temperature schedule prescribed above.

Now let us elaborate on how the random selection of an immediate neighbor is made. Suppose we are dealing with the initial trial solution of 1-2-3-4-5-6-7-1 in our example.

Initial trial solution: 1-2-3-4-5-6-7-1 $\quad Z_c = 69 \quad T_1 = 0.2Z_c = 13.8$

The sub-tour that will be reversed can begin anywhere between the second slot (currently designating city 2) and the sixth slot (currently designating city 6). These five slots can be given equal probabilities by having the following values of a random number between 0 and 1 correspond to choosing the slot indicated below.

0.0000–0.1999: Sub-tour begins in slot 2.
0.2000–0.3999: Sub-tour begins in slot 3.
0.4000–0.5999: Sub-tour begins in slot 4.
0.6000–0.7999: Sub-tour begins in slot 5.
0.8000–0.9999: Sub-tour begins in slot 6.

Suppose that the random number generated happens to be 0.2779.

0.2779: Choose a sub-tour that begins in slot 3.

By beginning in slot 3, the sub-tour that will be reversed needs to end somewhere between slots 4 and 7. These four slots are given equal probabilities by using the following correspondence with a random number.

0.0000–0.2499: Sub-tour ends in slot 4.
0.2500–0.4999: Sub-tour ends in slot 5.
0.5000–0.7499: Sub-tour ends in slot 6.
0.7500–0.9999: Sub-tour ends in slot 7.

Suppose that the random number generated for this purpose happens to be 0.0461.

0.0461: Choose to end the sub-tour in slot 4.

Since slots 3 and 4 currently designate that cities 3 and 4 are the third and fourth cities visited in the tour, the sub-tour of cities 3-4 will be reversed.

Reverse 3-4 (see Fig. 14.5): 1-2-4-3-5-6-7-1 $\quad Z_n = 65$

This immediate neighbor of the current (initial) trial solution becomes the current candidate to be the next trial solution. Since

$$Z_n = 65 < Z_c = 69,$$

this candidate is better than the current trial solution (remember that the objective here is to *minimize* the total distance of the tour), so this candidate is automatically accepted to be next trial solution.

This choice of a sub-tour reversal was a fortunate one because it led to a feasible solution. This does not always happen in traveling salesman problems like our example where certain pairs of cities are not directly connected by a link. For example, if the random numbers had called for reversing 2-3-4-5 to obtain the tour 1-5-4-3-2-6-7-1, Fig. 14.4

shows that this is an infeasible solution because there is no link between cities 1 and 5 as well as no link between cities 2 and 6. When this happens, new pairs of random numbers would need to be generated until a feasible solution is obtained. (A more sophisticated procedure also can be constructed to generate random numbers only for relevant links.)

To illustrate a case where the current candidate to be the next trial solution is worse than the current trial solution, suppose that the second iteration results in reversing 3-5-6 (as in Fig. 14.6) to obtain 1-2-4-6-5-3-7-1, which has a total distance of 64. Then suppose that the third iteration begins by reversing 3-7 (as in Fig. 14.9) to obtain 1-2-4-6-5-7-3-1 (which has a total distance of 66) as the current candidate to be the next trial solution. Since 1-2-4-6-5-3-7-1 (with a total distance of 64) is the current trial solution for iteration 3, we now have

$$Z_c = 64, \qquad Z_n = 66, \qquad T_1 = 13.8.$$

Therefore, since the objective here is *minimization,* the probability of accepting 1-2-4-6-5-7-3-1 as the next trial solution is

$$\begin{aligned}\text{Prob}\{\text{acceptance}\} &= e^{(Z_c - Z_n)/T_1} \\ &= e^{-2/13.8} \\ &= 0.865.\end{aligned}$$

If the next random number generated is less than 0.865, this candidate solution will be accepted as the next trial solution. Otherwise, it will be rejected.

Table 14.5 shows the results of using IOR Tutorial to apply the complete simulated annealing algorithm to this problem. Note that iterations 14 and 16 tie for finding the best

TABLE 14.5 One application of the simulated annealing algorithm in IOR Tutorial to the traveling salesman problem example

Iteration	*T*	Trial Solution Obtained	Distance
0		1-2-3-4-5-6-7-1	69
1	13.8	1-3-2-4-5-6-7-1	68
2	13.8	1-2-3-4-5-6-7-1	69
3	13.8	1-3-2-4-5-6-7-1	68
4	13.8	1-3-2-4-6-5-7-1	65
5	13.8	1-2-3-4-6-5-7-1	66
6	6.9	1-2-3-4-5-6-7-1	69
7	6.9	1-3-2-4-5-6-7-1	68
8	6.9	1-2-3-4-5-6-7-1	69
9	6.9	1-2-3-5-4-6-7-1	65
10	6.9	1-2-3-4-5-6-7-1	69
11	3.45	1-2-3-4-6-5-7-1	66
12	3.45	1-3-2-4-6-5-7-1	65
13	3.45	1-3-7-5-6-4-2-1	66
14	3.45	1-3-5-7-6-4-2-1	63 ← Minimum
15	3.45	1-3-7-5-6-4-2-1	66
16	1.725	1-3-5-7-6-4-2-1	63 ← Minimum
17	1.725	1-3-7-5-6-4-2-1	66
18	1.725	1-3-2-4-6-5-7-1	65
19	1.725	1-2-3-4-6-5-7-1	66
20	1.725	1-3-2-4-6-5-7-1	65
21	0.8625	1-3-7-5-6-4-2-1	66
22	0.8625	1-3-2-4-6-5-7-1	65
23	0.8625	1-2-3-4-6-5-7-1	66
24	0.8625	1-3-2-4-6-5-7-1	65
25	0.8625	1-3-7-5-6-4-2-1	66

trial solution, 1-3-5-7-6-4-2-1 (which happens to be the optimal solution along with the equivalent tour in the reverse direction, 1-2-4-6-7-5-3-1), so this solution is accepted as the final solution. You might find it interesting to apply this software to the same problem yourself. Due to the randomness built into the algorithm, the sequence of trial solutions obtained will be different each time. Because of this feature, practitioners sometimes will reapply a simulated annealing algorithm to the same problem several times to increase the chance of finding an optimal solution. (Problem 14.3-2 asks you to do this for this same example.) The initial trial solution also may be changed each time to help facilitate a more thorough exploration of the entire feasible region.

If you would like to see **another example** of how random numbers are used to perform an iteration of the basic simulated annealing algorithm for a traveling salesman problem, one is provided in the Solved Examples section of the book's website.

Before going on to the next example, we should pause at this point to mention a couple of ways in which advanced features of tabu search can be combined fruitfully with simulated annealing. One way is by applying the *strategic oscillation* feature of tabu search to the temperature schedule of simulated annealing. Strategic oscillation adjusts the temperature schedule by decreasing the temperatures more rapidly than usual but then strategically moving the temperatures back and forth across levels where the best solutions were found. Another way involves applying the candidate-list strategies of tabu search to the move selection rule of simulated annealing. The idea here is to scan multiple neighbors to see if an improving move is found before applying the randomized rule for accepting or rejecting the current candidate to be the next trial solution. These changes have sometimes produced significant improvements.

As these ideas for applying features of tabu search to simulated annealing suggest, a *hybrid algorithm* that combines the ideas of different metaheuristics can sometimes perform better than an algorithm that is based solely on a single metaheuristic. Although we are presenting three commonly used metaheuristics separately in this chapter, experienced practitioners occasionally will pick and choose among the ideas of these and other metaheuristics in designing their heuristic methods.

The Nonlinear Programming Example

Now reconsider the example of a small nonlinear programming problem (only a single variable) that was introduced in Sec. 14.1. The problem is to

$$\text{Maximize} \qquad f(x) = 12x^5 - 975x^4 + 28{,}000x^3 - 345{,}000x^2 + 1{,}800{,}000x,$$

subject to

$$0 \leq x \leq 31.$$

The graph of $f(x)$ in Fig. 14.1 reveals that there are local optima at $x = 5$, $x = 20$, and $x = 31$, but only $x = 20$ is a global optimum.

The metaheuristics area in IOR Tutorial includes a procedure for applying the simulated annealing algorithm to small nonlinear programming problems of the form,

$$\text{Maximize} \qquad f(x_1, \ldots, x_n)$$

subject to

$$L_j \leq x_j \leq U_j, \quad \text{for } j = 1, \ldots, n,$$

where $n = 1$ or 2, and where L_j and U_j are constants ($0 \leq L_j < U_j \leq 63$) representing the bounds on x_j. (Having relatively tight bounds on the individual variables is highly desirable for the efficiency of a simulated annealing algorithm, as well as for genetic algorithms discussed in the next section.) One or two linear functional constraints on the variables $\mathbf{x} = (x_1, \ldots, x_n)$ also can be included when $n = 2$. For the example, we have

$$n = 1, \qquad L_1 = 0, \qquad U_1 = 31,$$

with no linear functional constraints.

This procedure in IOR Tutorial designs the details of the simulated annealing algorithm for such nonlinear programming problems as follows.

1. **Initial trial solution:** You may enter any feasible solution, but it is helpful to enter one that appears to be a good feasible solution. In the absence of any clues about where the good feasible solutions might lie, it is reasonable to set each variable x_j midway between its lower bound L_j and upper bound U_j in order to start the search in the middle of the feasible region. (For this reason, $x = 15.5$ is a reasonable choice for the initial trial solution for the example.)
2. **Neighborhood structure:** Any feasible solution is considered to be an immediate neighbor of the current trial solution. However, the method described below for selecting an immediate neighbor to become the current candidate to be the next trial solution gives a preference to feasible solutions that are relatively close to the current trial solution, while still allowing for the possibility of moving to a different part of the feasible region to continue the search.
3. **Random selection of an immediate neighbor:** Set

$$\sigma_j = \frac{U_j - L_j}{6}, \qquad \text{for } j = 1, \ldots, n.$$

Then, given the current trial solution $(x_1, \ldots, x_n)$,

$$\text{reset } x_j = x_j + N(0, \sigma_j), \qquad \text{for } j = 1, \ldots, n,$$

where $N(0, \sigma_j)$ is a random observation from a *normal distribution* with mean zero and standard deviation σ_j. If this does not result in a feasible solution, then repeat this process (starting again from the current trial solution) as many times as needed to obtain a feasible solution.

4. **Temperature schedule:** As for traveling salesman problems, five iterations are performed at each of five values of T (T_1, T_2, T_3, T_4, T_5) in turn, where

$T_1 = 0.2Z_c$ when Z_c is the objective function value for the initial trial solution,
$T_2 = 0.5T_1$,
$T_3 = 0.5T_2$,
$T_4 = 0.5T_3$,
$T_5 = 0.5T_4$.

The reason for setting $\sigma_j = (U_j - L_j)/6$ when selecting an immediate neighbor is that when the variable x_j is midway between L_j and U_j, any new feasible value of the variable is within three standard deviations of the current value. This gives a significant probability that the new value will move most of the way to one of its bounds even though there is a much higher probability that the new value will be relatively close to the current value. There are a number of methods for generating a random observation $N(0, \sigma_j)$ from a normal

distribution (as will be discussed briefly in Sec. 20.4). For example, the Excel function, NORMINV(RAND(),0,σ_j), generates such a random observation. For your homework, here is a straightforward way of generating the random observations you need. Obtain a random number r and then use the normal table in Appendix 5 to find the value of $N(0, \sigma_j)$ such that $P\{X \leq N(0, \sigma_j)\} = r$ when X is a normal random variable with mean 0 and standard deviation σ_j.

To illustrate how the algorithm designed in this way would be applied to the example, let us start with $x = 15.5$ as the initial trial solution. Thus,

$$Z_c = f(15.5) = 3{,}741{,}121 \qquad \text{and} \qquad T_1 = 0.2Z_c = 748{,}224.$$

Since

$$\sigma = \frac{U - L}{6} = \frac{31 - 0}{6} = 5.167,$$

the next step is to generate a random observation $N(0, 5.167)$ from a normal distribution with mean zero and this standard deviation. To do this, we first obtain a random number, which happens to be 0.0735. Going to the normal table in Appendix 5, $P\{\text{standard normal} \leq -1.45\} = 0.0735$, so $N(0, 5.167) = -1.45(5.167) = -7.5$. The current candidate to be the next trial solution then is obtained by resetting x as

$$\begin{aligned} x &= 15.5 + N(0, 5.167) = 15.5 - 7.5 \\ &= 8, \end{aligned}$$

so that

$$Z_n = f(x) = 3{,}055{,}616.$$

Because

$$\frac{Z_n - Z_c}{T} = \frac{3{,}055{,}616 - 3{,}741{,}121}{748{,}224} = -0.916$$

the probability of accepting $x = 8$ as the next trial solution is

$$\text{Prob}\{\text{acceptance}\} = e^{-0.916} = 0.400.$$

Therefore, $x = 8$ will be accepted only if the corresponding random number between 0 and 1 happens to be less than 0.400. Thus, $x = 8$ is fairly likely to be rejected. (In somewhat later iterations when T is much smaller, $x = 8$ would almost certainly be rejected.) This is fortunate since Fig. 14.1 reveals that the search should focus on the portion of the feasible region between $x = 10$ and $x = 30$ in order to start climbing the tallest hill.

Table 14.6 provides the results that were obtained by using IOR Tutorial to apply the complete simulated annealing algorithm to this nonlinear programming problem. Note how the trial solutions obtained vary fairly widely over the feasible region during the early iterations, but then start approaching the top of the tallest hill more consistently during the later iterations when T has been reduced to much smaller values. Therefore, of the 25 iterations, the best trial solution of $x = 20.031$ (as compared to the optimal solution of $x = 20$) was not obtained until iteration 21.

Once again, you might find it interesting to apply this software to the same problem yourself to see what is yielded by new sequences of random numbers and random observations from normal distributions. (Problem 14.3-6 asks you to do this several times.)

TABLE 14.6 One application of the simulated annealing algorithm in IOR Tutorial to the nonlinear programming example

Iteration	T	Trial Solution Obtained	$f(x)$
0		$x = 15.5$	3,741,121.0
1	748,224	$x = 17.557$	4,167,533.956
2	748,224	$x = 14.832$	3,590,466.203
3	748,224	$x = 17.681$	4,188,641.364
4	748,224	$x = 16.662$	3,995,966.078
5	748,224	$x = 18.444$	4,299,788.258
6	374,112	$x = 19.445$	4,386,985.033
7	374,112	$x = 21.437$	4,302,136.329
8	374,112	$x = 18.642$	4,322,687.873
9	374,112	$x = 22.432$	4,113,901.493
10	374,112	$x = 21.081$	4,345,233.403
11	187,056	$x = 20.383$	4,393,306.255
12	187,056	$x = 21.216$	4,330,358.125
13	187,056	$x = 21.354$	4,313,392.276
14	187,056	$x = 20.795$	4,370,624.01
15	187,056	$x = 18.895$	4,348,060.727
16	93,528	$x = 21.714$	4,259,787.734
17	93,528	$x = 19.463$	4,387,360.1
18	93,528	$x = 20.389$	4,393,076.988
19	93,528	$x = 19.83$	4,398,710.575
20	93,528	$x = 20.68$	4,378,591.085
21	46,764	$x = 20.031$	4,399,955.913 ← Maximum
22	46,764	$x = 20.184$	4,398,462.299
23	46,764	$x = 19.9$	4,399,551.462
24	46,764	$x = 19.677$	4,395,385.618
25	46,764	$x = 19.377$	4,383,048.039

14.4 GENETIC ALGORITHMS

Genetic algorithms provide a third type of metaheuristic that is quite different from the first two. This type tends to be particularly effective at exploring various parts of the feasible region and gradually evolving toward the best feasible solutions.

After introducing the basic concepts for this type of metaheuristic, we will apply a basic genetic algorithm to the same nonlinear programming example just considered above with the additional constraint that the variable is restricted to integer values. We then will apply this approach to the same traveling salesman problem example considered in each of the preceding sections.

Basic Concepts

Just as simulated annealing is based on an analogy to a natural phenomenon (the physical annealing process), genetic algorithms are greatly influenced by another form of a natural phenomenon. In this case, the analogy is to the biological *theory of evolution* formulated by Charles Darwin in the mid-19th century. Each species of plants and animals has great individual variation. Darwin observed that those individuals with variations that impart a survival advantage through improved adaptation to the environment are most likely to survive to the next generation. This phenomenon has since been referred to as *survival of the fittest*.

The modern field of genetics provides a further explanation of this process of evolution and the *natural selection* involved in the survival of the fittest. In any species that

An Application Vignette

Intel Corporation is the world's largest semiconductor chip maker. With well over 80,000 employees and annual revenues over $53 billion, it has over 5000 products serving a wide variety of markets.

With so many products, one key to the continuing success of the company is an effective system for continually updating the design and scheduling of its product line. It can maximize its revenues only by introducing products into markets with the right features, at the right price, and at the right time. Therefore, a major operations research study was undertaken to optimize how this is done. The resulting model incorporated market requirements and financials, design-engineering capabilities, manufacturing costs, and multiple-time dynamics. This model then was embedded in a decision support system that soon was used by hundreds of Intel employees representing most major Intel groups and many distinct job functions.

The algorithmic heart of this decision support system is a *genetic algorithm* that handles resource constraints, scheduling, and financial optimization. This algorithm uses a fitness function to evaluate candidate solutions and then performs the usual genetic operators of mutation and crossover. It also calls on a combination of heuristic methods and mathematical optimization techniques to optimize product composition. This algorithm and its associated database enabled a new business process that is shifting Intel divisions to a unified focus on global profit maximization.

This dramatic application of operations research revolving around a genetic algorithm led to OR professionals from Intel winning the prestigious 2011 Daniel H. Wagner Prize for Excellence in Operations Research Practice.

Source: Rash, E., and K. Kempf, "Product Line Design and Scheduling at Intel," *Interfaces*, **42**(5): 425–436, September–October 2012. (A link to this article is provided on our website, www.mhhe.com/hillier.)

reproduces by sexual reproduction, each offspring inherits some of the *chromosomes* from each of the two parents, where the *genes* within the chromosomes determine the individual features of the child. A child who happens to inherit the better features of the parents is slightly more likely to survive into adulthood and then become a parent who passes on some of these features to the next generation. The population tends to improve slowly over time by this process. A second factor that contributes to this process is a random, low-level mutation rate in the DNA of the chromosomes. Thus, a *mutation* occasionally occurs that changes the features of a chromosome that a child inherits from a parent. Although most mutations have no effect or are disadvantageous, some mutations provide desirable improvements. Children with desirable mutations are slightly more likely to survive and contribute to the future gene pool of the species.

These ideas transfer over to dealing with optimization problems in a rather natural way. Feasible solutions for a particular problem correspond to members of a particular species, where the fitness of each member now is measured by the value of the objective function. Rather than processing a single trial solution at a time (as with basic forms of tabu search and simulated annealing), we now work with an entire *population* of trial solutions.[1] For each iteration (generation) of a genetic algorithm, the current **population** consists of the set of trial solutions currently under consideration. These trial solutions are thought of as the currently living members of the species. Some of the youngest members of the population (including especially the fittest members) survive into adulthood and become **parents** (paired at random) who then have **children** (new trial solutions) who share some of the features (genes) of both parents. Since the fittest members of the population are more likely to become parents than others, a genetic algorithm tends to generate *improving populations* of trial solutions as it proceeds. **Mutations** occasionally occur so that certain children also can acquire features (sometimes desirable features) that are not possessed by either parent. This helps a genetic algorithm to explore a new, perhaps better part of the feasible region than previously considered. Eventually, survival of the fittest should tend to lead a genetic algorithm to a trial solution (the best of any considered) that is at least nearly optimal.

[1]One of the intensification strategies of tabu search also maintains a population of best solutions. The population is used to create linking paths between its members and to relaunch the search along these paths.

Although the analogy of the process of biological evolution defines the core of any genetic algorithm, it is not necessary to adhere rigidly to this analogy in every detail. For example, some genetic algorithms (including the one outlined below) allow the same trial solution to be a parent repeatedly over multiple generations (iterations). Thus, the analogy needs to be only a starting point for defining the details of the algorithm to best fit the problem under consideration.

Here is a rather typical outline of a genetic algorithm that we will employ for the two examples.

Outline of a Basic Genetic Algorithm

Initialization. Start with an initial population of feasible trial solutions, perhaps by generating them randomly. Evaluate the *fitness* (the value of the objective function) for each member of this current population.

Iteration. Use a random process that is biased toward the more fit members of the current population to select some of the members (an even number) to become parents. Pair up the parents randomly and then have each pair of parents give birth to two children (new *feasible* trial solutions) whose features (genes) are a random mixture of the features of the parents, except for occasional mutations. (Whenever the random mixture of features and any mutations result in an *infeasible* solution, this is a *miscarriage,* so the process of attempting to give birth then is repeated until a child is born that corresponds to a *feasible* solution.) Retain the children and enough of the best members of the current population to form the new population of the same size for the next iteration. (Discard the other members of the current population.) Evaluate the fitness for each new member (the children) in the new population.

Stopping rule. Use some stopping rule, such as a fixed number of iterations, a fixed amount of CPU time, or a fixed number of consecutive iterations without any improvement in the best trial solution found so far. Use the best trial solution found on any iteration as the final solution.

Before this algorithm can be implemented the following questions need to be answered:

1. What should the population size be?
2. How should the members of the current population be selected to become parents?
3. How should the features of the children be derived from the features of the parents?
4. How should mutations be injected into the features of the children?
5. Which stopping rule should be used?

The answers to these questions depend greatly on the structure of the specific problem being addressed. The metaheuristics area in the IOR Tutorial does include two versions of the algorithm. One is for very small integer nonlinear programming problems like the example considered next. The other is for small traveling salesman problems. Both versions answer some of the questions in the same way, as described below:

1. **Population size:** Ten. (This size is reasonable for the small problems for which this software is designed, but much larger populations commonly are used for large problems.)
2. **Selection of parents:** From among the five most fit members of the population (according to the value of the objective function), select four randomly to become parents. From among the five least fit members, select two randomly to become parents. Pair up the six parents randomly to form three couples.
3. **Passage of features (genes) from parents to children:** This process is highly problem dependent and so differs for the two versions of the algorithm in the software, as described later for the two examples.

4. **Mutation rate:** The probability that an inherited feature of a child mutates into an opposite feature is set at 0.1 in the software. (Much smaller mutation rates commonly are used for large problems.)
5. **Stopping rule:** Stop after five consecutive iterations without any improvement in the best trial solution found so far.

Now we are ready to apply the algorithm to the two examples.

The Integer Version of the Nonlinear Programming Example

We return again to the small nonlinear programming problem that was introduced in Sec. 14.1 (see Fig. 14.1) and then addressed using a simulated annealing algorithm at the end of the preceding section. However, we now add the additional constraint that the problem's single variable x must have an integer value. Because the problem already has the constraint that $0 \leq x \leq 31$, this means that the problem has 32 feasible solutions, $x = 0, 1, 2, \ldots, 31$. (Having such bounds is very important for a genetic algorithm, since it reduces the search space to the relevant region.) Thus, we now are dealing with an *integer* nonlinear programming problem.

When applying a genetic algorithm, *strings of binary digits* often are used to represent the solutions of the problem. Such an *encoding* of the solutions is a particularly convenient one for the various steps of a genetic algorithm, including the process of parents giving birth to children. This encoding is easy to do for our particular problem because we simply can write each value of x in base 2. Since 31 is the maximum feasible value of x, only five binary digits are required to write any feasible value. We always will include all five binary digits even when the leading digit or digits are zeroes. Thus, for example,

$x = 3$ is 00011 in base 2,
$x = 10$ is 01010 in base 2,
$x = 25$ is 11001 in base 2.

Each of the five binary digits is referred to as one of the **genes** of the solution, where the two possible values of the binary digit describe which of two possible features is being carried in that gene to help form the overall genetic makeup. When both parents have the same feature, it will be passed down to each child (except when a mutation occurs). However, when the two parents carry opposite features on the same gene, which feature a child will inherit becomes random.

For example, suppose that the two parents are

P1: 00011 and
P2: 01010.

Since the first, third, and fourth digits agree, the children then automatically become (barring mutations)

C1: 0x01x and
C2: 0x01x,

where x indicates that this particular digit is not known yet. Random numbers are used to identify these unknown digits, where a natural correspondence is

0.0000–0.4999 corresponds to the digit being 0,
0.5000–0.9999 corresponds to the digit being 1.

For example, suppose that the next four random numbers generated are 0.7265, 0.5190, 0.0402, and 0.3639 so that the two unknown digits for the first child are both 1s and the two unknown digits for the second child are both 0s. The children then become (barring mutations)

C1: 01011 and
C2: 00010.

This particular method of generating the children from the parents is known as *uniform crossover*. It is perhaps the most intuitive of the various alternative methods that have been proposed.

We now need to consider the possibility of mutations that would affect the genetic makeup of the children.

Since the probability of a mutation in any gene (flipping the binary digit to the opposite value) has been set at 0.1 for our algorithm, we can let the random numbers

0.0000–0.0999 correspond to a mutation,
0.1000–0.9999 correspond to no mutation.

For example, suppose that in the next 10 random numbers generated, only the eighth one is less than 0.1000. This indicates that no mutation occurs in the first child, but the third gene (digit) in the second child flips its value. Therefore, the final conclusion is that the two children are

C1: 01011 and
C2: 00110.

Returning to base 10, the two parents correspond to the solutions, $x = 3$ and $x = 10$, whereas their children would have been (barring mutations) $x = 11$ and $x = 2$. However, because of the mutation, the children become $x = 11$ and $x = 6$.

For this particular example, any integer value of x such that $0 \leq x \leq 31$ (in base 10) is a feasible solution, so every 5-digit number in base 2 also is a feasible solution. Therefore, the above process of creating children never results in a *miscarriage* (an infeasible solution). However, if the upper bound on x were, say, $x \leq 25$ instead, then miscarriages would occur occasionally. Whenever a miscarriage occurs, the solution is discarded and the entire process of creating a child is repeated until a feasible solution is obtained.

This example includes only a single variable. For a nonlinear programming problem with multiple variables, each member of the population again would use base 2 to show the value of each variable. The above process of generating children from parents then would be done in the same way one variable at a time.

Table 14.7 shows the application of the complete algorithm to this example through both the initialization step (part *a* of the table) and iteration 1 (part *b* of the table). In the initialization step, each of the members of the initial population were generated by generating five random numbers and using the correspondence between a random number and a binary digit given earlier to obtain the five binary digits in turn. The corresponding value of x in base 10 then is plugged into the objective function given at the beginning of Sec. 14.1 to evaluate the fitness of that member of the population.

The five members of the initial population that have the highest degree of fitness (in order) are members 10, 8, 4, 1, and 7. To randomly select four of these members to become parents, a random number is used to select one member to be rejected, where 0.0000– 0.1999 corresponds to ejecting the first member listed (member 10), 0.2000–0.3999 corresponds to rejecting the second member, and so forth. In this case, the random number was 0.9665, so the fifth member listed (member 7) does not become a parent.

From among the five less fit members of the initial population (members 2, 1, 6, 5, and 9), random numbers now are used to select which two of these members will become parents. In this case, the random numbers were 0.5634 and 0.1270. For the first random number, 0.0000–0.1999 corresponds to selecting the first member listed (member 2), 0.2000–0.3999 corresponds to selecting the second member, and so forth, so the third member listed (member 6) is the one selected in this case. Since only four members (2, 1, 5, and 9) now remain for selecting the last parent, the corresponding intervals for the second random number are 0.0000–0.2499, 0.2500–0.4999, 0.5000–0.7499, and

TABLE 14.7 Application of the genetic algorithm to the integer nonlinear programming example through (*a*) the initialization step and (*b*) iteration 1

	Member	Initial Population	Value of x	Fitness
(a)	1	0 1 1 1 1	15	3,628,125
	2	0 0 1 0 0	4	3,234,688
	3	0 1 0 0 0	8	3,055,616
	4	1 0 1 1 1	23	3,962,091
	5	0 1 0 1 0	10	2,950,000
	6	0 1 0 0 1	9	2,978,613
	7	0 0 1 0 1	5	3,303,125
	8	1 0 0 1 0	18	4,239,216
	9	1 1 1 1 0	30	1,350,000
	10	1 0 1 0 1	21	4,353,187

	Member	Parents	Children	Value of x	Fitness
(b)	10	1 0 1 0 1	0 0 1 0 1	5	3,303,125
	2	0 0 1 0 0	1 0 0 0 1	17	4,064,259
	8	1 0 0 1 0	1 0 0 1 1	19	4,357,164
	4	1 0 1 1 1	1 0 1 0 0	20	4,400,000
	1	0 1 1 1 1	0 1 0 1 1	11	2,980,637
	6	0 1 0 0 1	0 1 1 1 1	15	3,628,125

0.7500–0.9999. Because 0.1270 falls in the first of these intervals, the first remaining member listed (member 2) is selected to be a parent.

The next step is to pair up the six parents—members 10, 8, 4, 1, 6, and 2. Let us begin by using a random number to determine the mate of the first member listed (member 10). The random number 0.8204 indicated that it should be paired up with the fifth of the other five parents listed (member 2). To pair up the next member listed (member 8), the next random number was 0.0198, which is in the interval 0.0000–0.3333, so the first of the three remaining parents listed (member 4) is chosen to be the mate of member 8. This then leaves the two remaining parents (members 1 and 6) to become the last couple.

Part (*b*) of Table 14.7 shows the children that were reproduced by these parents by using the process illustrated earlier in this subsection. Note that mutations occurred in the third gene of the second child and the fourth gene of the fourth child. By and large, the six children have a relatively high degree of fitness. In fact, for each pair of parents, both of the children turned out to be more fit than one of the parents. This does not always occur but is fairly common. In the case of the second pair of parents, both of the children happen to be more fit than both parents. Fortuitously, both of these children ($x = 19$ and $x = 20$) actually are superior to *any* of the members of the preceding population given in part (*a*) of the table. To form the new population for the next iteration, all six children are retained along with the four most fit members of the preceding population (members 10, 8, 4, and 1).

Subsequent iterations would proceed in a similar fashion. Since we know from the discussion in Sec. 14.1 (see Fig. 14.1) that $x = 20$ (the best trial solution generated in iteration 1) actually is the optimal solution for this example, subsequent iterations would not provide any further improvement. Therefore, the stopping rule would terminate the algorithm after five more iterations and provide $x = 20$ as the final solution.

Your IOR Tutorial includes a procedure for applying this same genetic algorithm to other very small integer nonlinear programming problems. (The form and size restrictions are the same as specified in Sec. 14.3 for nonlinear programming problems.)

You might find it interesting to apply this procedure in IOR Tutorial to this same example. Because of the randomness inherent in the algorithm, different intermediate results are obtained each time that it is applied. (Problem 14.4-3 asks you to apply the algorithm to this example several times.)

Although this was a discrete example, genetic algorithms can also be applied to continuous problems such as a nonlinear programming problem without an integer constraint. In this case, the value of a continuous variable would be represented (or closely approximated) by a decimal number in base 2. For example, $x = 23\frac{5}{8}$ is 10111.10100 in base 2, and $x = 23.66$ is closely approximated by 10111.10101 in base 2. All the binary digits on both sides of the decimal point can be treated just as before to have parents reproduce children, and so forth.

The Traveling Salesman Problem Example

Sections 14.2 and 14.3 illustrated how a tabu search algorithm and a simulated annealing algorithm would be applied to the particular traveling salesman problem introduced in Sec. 14.1 (see Fig. 14.4). Now let us see how our genetic algorithm can be applied using this same example.

Rather than using binary digits in this case, we will continue to represent each solution (tour) in the natural way as a sequence of cities visited. For example, the first solution considered in Sec. 14.1 is the tour of the cities in the following order: 1-2-3-4-5-6-7-1, where city 1 is the home base where the tour must begin and end. We should point out, however, that genetic algorithms for traveling salesman problems frequently use other methods for *encoding* solutions. In general, clever methods of representing solutions (often by using strings of binary digits) can make it easier to generate children, create mutations, maintain feasibility, and so forth, in a natural way. The development of an appropriate *encoding scheme* is a key part of developing an effective genetic algorithm for any application.

A complication with this particular example is that, in a sense, it is too easy. Because of the rather limited number of links between pairs of cities in Fig. 14.4, this problem barely has 10 distinct feasible solutions if we rule out a tour that is simply a previously considered tour in the reverse direction. Therefore, it is not possible to have an initial population with 10 distinct trial solutions such that the resulting six parents then reproduce distinct children that also are distinct from the members of the initial population (including the parents).

Fortunately, a genetic algorithm can still operate reasonably well when there is a modest amount of duplication in the trial solutions in a population or in two consecutive populations. For example, even when both parents in a couple are identical, it still is possible for their children to differ from the parents because of mutations.

The genetic algorithm for traveling salesman problems in your IOR Tutorial does not do anything to avoid duplication in the trial solutions considered. Each of the 10 trial solutions in the initial population is generated in turn as follows. Starting from the home base city, random numbers are used to select the next city from among those that have a link to the home base city (cities 2, 3, and 7 in Fig. 14.4). Random numbers then are used to select the third city from among the remaining cities that have a link to the second city. This process is continued until either every city is included once in the tour (plus a return to the home base city from the last city) or a dead end is reached because there is no link from the current city to any of the remaining cities that still need to be visited. In the latter case, the entire process for generating a trial solution is restarted from the beginning with new random numbers.

Random numbers are also used to reproduce children from a pair of parents. To illustrate this process, consider the following pair of parents:

P1: 1-2-3-4-5-6-7-1
P2: 1-2-4-6-5-7-3-1

As we describe the process of generating a child from these parents, we also summarize the results in Table 14.8 to help you follow the progression.

TABLE 14.8 Illustration of the process of generating a child for the traveling salesman problem example

Parent P1: 1-2-3-4-5-6-7-1
Parent P2: 1-2-4-6-5-7-3-1

Link	Options	Random Selection	Tour
1	1-2, 1-7, 1-2, 1-3	1-2	1-2
2	2-3, 2-4	2-4	1-2-4
3	4-3, 4-5, 4-6	4-3	1-2-4-3
4	3-5*, 3-7	3-5*	1-2-4-3-5
5	5-6, 5-6, 5-7	5-6	1-2-4-3-5-6
6	6-7	6-7	1-2-4-3-5-6-7
7	7-1	7-1	1-2-4-3-5-6-7-1

*A link that completes a sub-tour reversal

Ignoring the possibility of mutations for the time being, here is the main idea for how to generate a child.

> **Inheriting Links:** Genes correspond to the links in a tour. Therefore, each of the links (genes) inherited by a child should come from one parent or the other (or both). (One other possibility described later is that a parent also can pass down a sub-tour reversal.) These links being inherited are randomly selected one at a time until a complete tour (the child) has been generated.

To start this process with the above parents, since a tour must begin in city 1, a child's initial link must come from one of the parent's links that connect city 1 to another city. For parent P1, these are links 1-2 and 1-7. (Link 1-7 qualifies since it is equivalent to take the tour in either direction.) For parent P2, the corresponding links are 1-2 (again) and 1-3. The fact that both parents have link 1-2 doubles the probability that it will be inherited by a child. Therefore, when using a random number to determine which link the child will inherit, the interval 0.0000–0.4999 (or any interval of this size) corresponds to inheriting link 1-2 whereas the intervals 0.50000–0.7499 and 0.7500–0.9999 then would correspond to the choice of link 1-7 and link 1-3, respectively. Suppose 1-2 is selected, as shown in the first row of Table 14.8. After 1-2, one parent next uses link 2-3 whereas the other uses 2-4. Therefore, in generating the child, a random choice should be made between these two options. Suppose 2-4 is selected. (See the second row of Table 14.8.) There now are three options for the link to follow 1-2-4 because the first parent uses two links (4-3 and 4-5) to connect city 4 in its tour and the second parent uses link 4-6 (link 4-2 is ignored because city 2 already is in the child's tour). When randomly selecting one of these options, suppose 4-3 is chosen to form 1-2-4-3 as the beginning of the child's tour thus far, as shown in the third row of Table 14.8.

We now come to an additional feature of this process for generating a child's tour, namely, using a *sub-tour reversal* from a parent.

> **Inheriting a Sub-Tour Reversal:** One other possibility for a link inherited by a child is a link that is needed to complete a sub-tour reversal that the child's tour is making in a portion of a parent's tour.

To illustrate how this possibility can arise, note that the next city beyond 1-2-4-3 needs to be one of the cities not yet visited (city 5, 6, or 7), but the first parent does not have a link from city 3 to any of these other cities. The reason is that the child is using a sub-tour reversal (reversing 3-4) of this parent's tour, 1-2-3-4-5-6-7-1. Completing this sub-tour reversal requires adding the link 3-5, so this becomes one of the options for the next

link in the child's tour. The other option is link 3-7 provided by the second parent (link 3-1 is not an option because city 1 must come at the very end of the tour). One of these two options is selected randomly. Suppose the choice is link 3-5, which provides 1-2-4-3-5 as the child's tour thus far, as shown in the fourth row of Table 14.8.

To continue this tour, the options for the next link are 5-6 (provided by both parents) and 5-7 (provided by the second parent). Suppose that the random choice among 5-6, 5-6, and 5-7 is 5-6, so that the tour thus far is 1-2-4-3-5-6. (See the fifth row of Table 14.8.) Since the only city not yet visited is city 7, link 6-7 is automatically added next, followed by link 7-1 to return to home base. Thus, as shown in the last row of Table 14.8, the complete tour for the child is

C1: 1-2-4-3-5-6-7-1

Figure 14.5 in Sec. 14.1 displays how closely this child resembles the first parent, since the only difference is the sub-tour reversal obtained by reversing 3-4 in the parent.

If link 5-7 had been chosen instead to follow 1-2-4-3-5, the tour would have been completed automatically as 1-2-4-3-5-7-6-1. However, there is no link 6-1 (see Fig. 14.4), so a dead end is reached at city 6. When this happens, a *miscarriage* occurs and the entire process needs to be restarted from the beginning with new random numbers until a child with a complete tour is obtained. Then this process is repeated to obtain the second child.

We now need to add one more feature—the possibility of mutations—to complete the description of the process of generating children.

> **Mutations of Inherited Links:** Whenever a particular link normally would be inherited from a parent of a child, there is a small possibility that a mutation will occur that will reject that link and instead randomly select one of the other links from the current city to another city not already on the tour, regardless of whether that link is used by either parent.

Our genetic algorithm for traveling salesman problems implemented in your IOR Tutorial uses a probability of 0.1 that a mutation will occur each time the next link in the child's tour needs to be selected. Thus, whenever the corresponding random number is less than 0.1000, the choice of the link made in the normal manner described above is rejected (if any other possible choice exists). Instead, all the other links from the current city to a city not already in the tour (including links not provided by either parent) are identified, and one of these links is randomly selected to be the next link in the tour. For example, suppose that a mutation occurs when generating the very first link for the child. Even though 1-2 had been the random choice as the first link, this link now would be rejected because of the mutation. Since city 1 also has links to cities 3 and 7 (see Fig. 14.4), either link 1-3 or link 1-7 would be randomly selected to be the first tour. (Since the parents end their tours by using one or the other of these links, this can be viewed in this case as starting the child's tour by reversing the direction of one of the parents' tours.)

We now can outline the general procedure for generating a child from a pair of parents.

Procedure for Generating a Child

1. **Initialization:** To start, designate the home base city as the *current city*.
2. **Options for the next link:** Identify all the links from the current city to another city not already in the child's tour that are used by either parent in either direction. Also, add any link that is needed to complete a sub-tour reversal that the child's tour is making in a portion of a parent's tour.
3. **Selection of the next link:** Use a random number to randomly select one of the options identified in step 2.
4. **Check for a mutation:** If the next random number is less than 0.1000, a mutation occurs and the link selected in step 3 is rejected (unless there is no other link from the current

city to another city not already in the tour). If the link is rejected, identify all the other links from the current city to another city not already in the tour (including links not used by either parent). Use a random number to randomly select one of these other links.

5. **Continuation:** Add the link selected in step 3 (if no mutation occurs) or in step 4 (if a mutation occurs) to the end of the child's current incomplete tour and redesignate the city at the end of this link as the *current city*. If there still remains more than one city not included on the tour (plus the return to the home base city), return to steps 2–4 to select the next link. Otherwise, go to step 6.
6. **Completion:** With only one city remaining that has not yet been added to the child's tour, add the link from the current city to this remaining city. Then add the link from this last city back to the home base city to complete the tour for the child. However, if the needed link does not exist, a miscarriage occurs and the procedure must restart again from step 1.

This procedure is applied for each pair of parents to obtain each of their two children.

The genetic algorithm for traveling salesman problems in your IOR Tutorial incorporates this procedure for generating children as part of the overall algorithm outlined near the beginning of this section. Table 14.9 shows the results from applying this algorithm to the example through the initialization step and the first iteration of the overall algorithm. Because of the randomness built into the algorithm, its intermediate results (and perhaps the final best solution as well) will vary each time the algorithm is run to its completion. (To explore this further, Prob. 14.4-7 asks you to use your IOR Tutorial to apply the complete algorithm to this example several times.)

The fact that the example has only a relatively small number of distinct feasible solutions is reflected in the results shown in Table 14.9. Members 1, 4, 6, and 10 are identical, as are members 2, 7, and 9 (except that member 2 takes its tour in the reverse direction). Therefore, the random generation of the 10 members of the initial population resulted in only five distinct feasible solutions. Similarly, four of the six children generated (members 12, 14, 15, and 16) are identical to one of its parents (except that member 14 takes its tour in the opposite direction of its first parent). Two of the children (members

TABLE 14.9 One application of the genetic algorithm in IOR Tutorial to the traveling salesman problem example through (*a*) the initialization step and (*b*) iteration 1

(a)

Member	Initial Population	Distance
1	1-2-4-6-5-3-7-1	64
2	1-2-3-5-4-6-7-1	65
3	1-7-5-6-4-2-3-1	65
4	1-2-4-6-5-3-7-1	64
5	1-3-7-5-6-4-2-1	66
6	1-2-4-6-5-3-7-1	64
7	1-7-6-4-5-3-2-1	65
8	1-3-7-6-5-4-2-1	69
9	1-7-6-4-5-3-2-1	65
10	1-2-4-6-5-3-7-1	64

(b)

Member	Parents	Children	Member	Distance
1	1-2-4-6-5-3-7-1	1-2-4-5-6-7-3-1	11	69
7	1-7-6-4-5-3-2-1	1-2-4-6-5-3-7-1	12	64
2	1-2-3-5-4-6-7-1	1-2-4-5-6-7-3-1	13	69
6	1-2-4-6-5-3-7-1	1-7-6-4-5-3-2-1	14	65
4	1-2-4-6-5-3-7-1	1-2-4-6-5-3-7-1	15	64
5	1-3-7-5-6-4-2-1	1-3-7-5-6-4-2-1	16	66

12 and 15) have a better fitness (shorter distance) than one of its parents, but neither improved upon both of its parents. None of these children provide an optimal solution (which has a distance of 63). This illustrates the fact that a genetic algorithm may require many generations (iterations) on some problems before the survival-of-the-fittest phenomenon results in clearly superior populations.

The Solved Examples section of the book's website provides **another example** of applying this genetic algorithm to a traveling salesman problem. This problem has a somewhat larger number of distinct feasible solutions than the above example, so there is a greater diversity in its initial population, the resulting parents, and their children.

Genetic algorithms are well suited for dealing with the traveling salesman problem and good progress has been made on developing considerably more sophisticated versions than the one described above. In fact, at the time of this writing, a particularly powerful version that has successfully obtained high-quality solutions for problems with up to 200,000 cities (!) has just been announced.[2]

14.5 CONCLUSIONS

Some optimization problems (including various combinatorial optimization problems) are sufficiently complex that it may not be possible to solve for an optimal solution with the kinds of exact algorithms presented in previous chapters. In such cases, heuristic methods are commonly used to search for a good (but not necessarily optimal) feasible solution. Several metaheuristics are available that provide a general structure and strategy guidelines for designing a specific heuristic method to fit a particular problem. A key feature of these metaheuristic procedures is their ability to escape from local optima and perform a robust search of a feasible region.

This chapter has introduced three prominent types of metaheuristics. *Tabu search* moves from the current trial solution to the best neighboring trial solution at each iteration, much like a local improvement procedure, except that it allows a nonimproving move when an improving move is not available. It then incorporates short-term memory of the past search to encourage moving toward new parts of the feasible region rather than cycling back to previously considered solutions. In addition, it may employ intensification and diversification strategies based on long-term memory to focus the search on promising continuations. *Simulated annealing* also moves from the current trial solution to a neighboring trial solution at each iteration while occasionally allowing nonimproving moves. However, it selects the neighboring trial solution randomly and then uses the analogy to a physical annealing process to determine if this neighbor should be rejected as the next trial solution if it is not as good as the current trial solution. The third type of metaheuristic, *genetic algorithms,* works with an entire population of trial solutions at each iteration. It then uses the analogy to the biological theory of evolution, including the concept of survival of the fittest, to discard some of the trial solutions (especially the poorer ones) and replace them by some new ones. This replacement process has pairs of surviving members of the population pass on some of their features to pairs of new members just as if they were parents reproducing children.

For the sake of concreteness, we have described one basic algorithm for each metaheuristic and then adapted this algorithm to two specific types of problems (including the traveling salesman problem), using simple examples. However, many variations of each algorithm also have been developed by researchers and used by practitioners to better fit the characteristics of the complex problems being addressed. For example, literally dozens

[2]Nagata, Y., and S. Kobayashi: "A Powerful Genetic Algorithm Using Edge Assembly Crossover for the Traveling Salesman Problem," *INFORMS Journal on Computing*, **25**(2): 346–369, Spring 2013.

of variations of the basic genetic algorithm for traveling salesman problems presented in Sec. 14.4 (including different procedures for generating children) have been proposed, and research is continuing to determine what is most effective. (Some of the best methods for traveling salesman problems use special "k-opt" and "ejection chain" strategies that are carefully tailored to take advantage of the problem structure.) Therefore, the important lessons from this chapter are the basic concepts and intuition incorporated into each metaheuristic rather than the details of the particular algorithms presented here.

There are several other important types of metaheuristics in addition to the three that are featured in this chapter. These include, for example, ant colony optimization, scatter search, and artificial neural networks. (These suggestive names give a hint of the key idea that drives each of these metaheuristics.) Selected Reference 3 provides a thorough coverage of both these other metaheuristics and the three presented here.

Some heuristic algorithms actually are a hybrid of different types of metaheuristics in order to combine their better features. For example, short-term tabu search (without a diversification component) is very good at finding local optima but not as good at thoroughly exploring the various parts of a feasible region to find the part containing the global optimum, whereas a genetic algorithm has the opposite characteristics. Therefore, an improved algorithm sometimes can be obtained by beginning with a genetic algorithm to try to find the tallest hills (when the objective is maximization) and then switch to a basic tabu search at the very end to climb quickly to the top of these hills. The key for designing an effective heuristic algorithm is to incorporate whatever ideas work best for the problem at hand rather than adhering rigidly to the philosophy of a particular metaheuristic.

SELECTED REFERENCES

1. Coello, C., D. A. Van Veldhuizen, and G. B. Lamont: *Evolutionary Algorithms for Solving Multi-Objective Problems*, Kluwer Academic Publishers (now Springer), Boston, 2002.
2. Gen, M., and R. Cheng, *Genetic Algorithms and Engineering Optimization,* Wiley, New York, 2000.
3. Gendreau, M., and J.-Y. Potvin (eds): *Handbook of Metaheuristics*, 2nd ed., Springer, New York, 2010.
4. Glover, F.: "Tabu Search: A Tutorial," *Interfaces,* **20**(4): 74–94, July–August 1990.
5. Glover, F., and M. Laguna: *Tabu Search,* Kluwer Academic Publishers (now Springer), Boston, MA, 1997.
6. Gutin, G., and A. Punnen (eds.): *The Traveling Salesman Problem and Its Variations,* Kluwer Academic Publishers (now Springer), Boston, MA, 2002.
7. Haupt, R. L., and S. E. Haupt: *Practical Genetic Algorithms,* Wiley, Hoboken NJ, 1998.
8. Michalewicz, Z., and D. B. Fogel: *How To Solve It: Modern Heuristics,* Springer, Berlin, 2002.
9. Mitchell, M.: *An Introduction to Genetic Algorithms,* MIT Press, Cambridge, MA, 1998.
10. Reeves, C. R.: "Genetic Algorithms for the Operations Researcher," *INFORMS Journal on Computing,* **9:** 231–250, 1997. (Also see pp. 251–265 for commentaries on this feature article.)
11. Sarker, R., M. Mohammadian, and X. Yao (eds.): *Evolutionary Optimization,* Kluwer Academic Publishers (now Springer), Boston, MA, 2002.
12. Talbi, E.: *Metaheuristics*: From Design to Implementation, Wiley, Hoboken, NJ, 2009.

LEARNING AIDS FOR THIS CHAPTER ON OUR WEBSITE (www.mhhe.com/hillier)

Solved Examples:

Examples for Chapter 14

Automatic Procedures in IOR Tutorial:

Tabu Search Algorithm for Traveling Salesman Problems
Simulated Annealing Algorithm for Traveling Salesman Problems

Simulated Annealing Algorithm for Nonlinear Programming Problems
Genetic Algorithm for Integer Nonlinear Programming Problems
Genetic Algorithm for Traveling Salesman Problems

Glossary for Chapter 14

See Appendix 1 for documentation of the software.

PROBLEMS

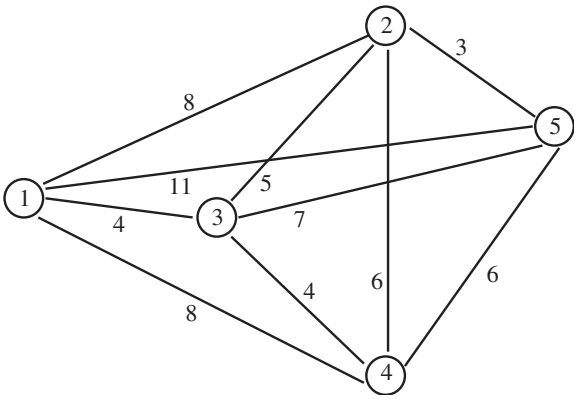
The symbol A to the left of some of the problems (or their parts) has the following meaning:

A: You should use the corresponding automatic procedure in IOR Tutorial. The printout will record the results obtained at each iteration.

An asterisk on the problem number indicates that at least a partial answer is given in the back of the book.

Instructions for Obtaining Random Numbers
For each problem or its part where random numbers are needed, obtain them from the consecutive random digits in Table 20.3 in Sec. 20.3 as follows. Start from the front of the top row of the table and form *five-digit* random numbers by placing a decimal point in front of each group of five random digits (0.09656, 0.96657, etc.) in the order that you need random numbers. Always restart from the front of the top row for each new problem or its part.

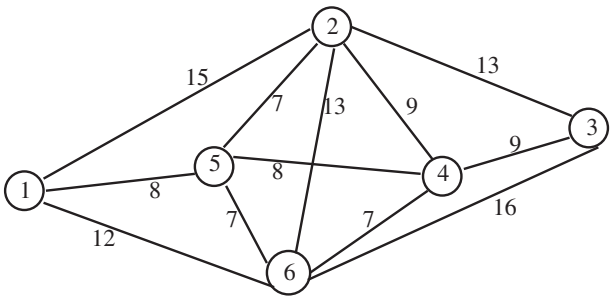
14.1-1. Consider the traveling salesman problem shown below, where city 1 is the home city.



(a) List all the possible tours, except exclude those that are simply the reverse of previously listed tours. Calculate the distance of each of these tours and thereby identify the optimal tour.
(b) Starting with 1-2-3-4-5-1 as the initial trial solution, apply the sub-tour reversal algorithm to this problem.
(c) Apply the sub-tour reversal algorithm to this problem when starting with 1-2-4-3-5-1 as the initial trial solution.
(d) Apply the sub-tour reversal algorithm to this problem when starting with 1-4-2-3-5-1 as the initial trial solution.

14.1-2. Reconsider the example of a traveling salesman problem shown in Fig. 14.4.
(a) When the sub-tour reversal algorithm was applied to this problem in Sec. 14.1, the first iteration resulted in a tie for which of two sub-tour reversals (reversing 3-4 or 4-5) provided the largest decrease in the distance of the tour, so the tie was broken arbitrarily in favor of the first reversal. Determine what would have happened if the second of these reversals (reversing 4-5) had been chosen instead.
(b) Apply the sub-tour reversal algorithm to this problem when starting with 1-2-4-5-6-7-3-1 as the initial trial solution.

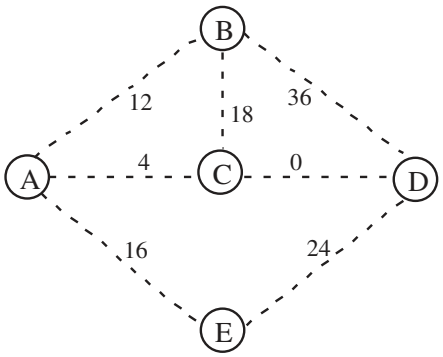
14.1-3. Consider the traveling salesman problem shown below, where city 1 is the home city.



(a) List all the possible tours, except exclude those that are simply the reverse of previously listed tours. Calculate the distance of each of these tours and thereby identify the optimal solution.
(b) Starting with 1-2-3-4-5-6-1 as the initial trial solution, apply the sub-tour reversal algorithm to this problem.
(c) Apply the sub-tour reversal algorithm to this problem when starting with 1-2-5-4-3-6-1 as the initial trial solution.

14.2-1. Read the referenced article that fully describes the OR study summarized in the application vignette presented in Sec. 14.2. Briefly describe how tabu search was applied in this study. Then list the various financial and nonfinancial benefits that resulted from this study.

14.2-2.* Consider the minimum spanning tree problem depicted below, where the dashed lines represent the potential links that could be inserted into the network and the number next to each dashed line represents the cost associated with inserting that particular link.



This problem also has the following two constraints:

Constraint 1: No more than one of the three links—AB, BC, and AE—can be included.

Constraint 2: Link AB can be included only if link BD also is included.

Starting with the initial trial solution where the inserted links are AB, AC, AE, and CD, apply the basic tabu search algorithm presented in Sec. 14.2 to this problem.

14.2-3. Reconsider the example of a constrained minimum spanning tree problem presented in Sec. 14.2 (see Fig. 14.7(*a*) for the data before introducing the constraints). Starting with a different initial trial solution, namely, the one with links AB, AD, BE, and CD, apply the basic tabu search algorithm again to this problem.

14.2-4. Reconsider the example of an unconstrained minimum spanning tree problem given in Sec. 10.4. Suppose that the following constraints are added to the problem:

Constraint 1: Either link AD or link ET must be included.

Constraint 2: At most one of the three links—AO, BC, and DE—can be included.

Starting with the optimal solution for the unconstrained problem given at the end of Sec. 10.4 as the initial trial solution, apply the basic tabu search algorithm to this problem.

14.2-5. Reconsider the traveling salesman problem shown in Prob. 14.1-1. Starting with 1-2-4-3-5-1 as the initial trial solution, apply the basic tabu search algorithm by hand to this problem.

A **14.2-6.** Consider the 8-city traveling salesman problem whose links have the associated distances shown in the following table (where a dash indicates the absence of a link).

City	2	3	4	5	6	7	8
1	14	15	—	—	—	—	17
2		13	14	20	—	—	21
3			11	21	17	9	9
4				11	10	8	20
5					15	18	—
6						9	—
7							13

City 1 is the home city. Starting with each of the initial trial solutions listed below, apply the basic tabu search algorithm in your IOR Tutorial to this problem. In each case, count the number of times that the algorithm makes a nonimproving move. Also point out any tabu moves that are made anyway because they result in the best trial solution found so far.

(a) Use 1-2-3-4-5-6-7-8-1 as the initial trial solution.
(b) Use 1-2-5-6-7-4-8-3-1 as the initial trial solution.
(c) Use 1-3-2-5-6-4-7-8-1 as the initial trial solution.

A **14.2-7.** Consider the 10-city traveling salesman problem whose links have the associated distances shown in the following table.

City	2	3	4	5	6	7	8	9	10
1	13	25	15	21	9	19	18	8	15
2		26	21	29	21	31	23	16	10
3			11	18	23	28	44	34	35
4				10	13	19	34	24	29
5					12	11	37	27	36
6						10	25	14	25
7							32	23	35
8								10	16
9									14

City 1 is the home city. Starting with each of the initial trial solutions listed below, apply the basic tabu search algorithm in your IOR Tutorial to this problem. In each case, count the number of times that the algorithm makes a nonimproving move. Also point out any tabu moves that are made anyway because they result in the best trial solution found so far.

(a) Use 1-2-3-4-5-6-7-8-9-10-1 as the initial trial solution.
(b) Use 1-3-4-5-7-6-9-8-10-2-1 as the initial trial solution.
(c) Use 1-9-8-10-2-4-3-6-7-5-1 as the initial trial solution.

14.3-1. While applying a simulated annealing algorithm to a certain problem, you have come to an iteration where the current value of T is $T = 2$ and the value of the objective function for the current trial solution is 30. This trial solution has four immediate neighbors and their objective function values are 29, 34, 31, and 24. For each of these four immediate neighbors in turn, you wish to determine the probability that the move selection rule would accept this immediate

neighbor if it is randomly selected to become the current candidate to be the next trial solution.

(a) Determine this probability for each of the immediate neighbors when the objective is *maximization* of the objective function.

(b) Determine this probability for each of the immediate neighbors when the objective is *minimization* of the objective function.

A **14.3-2.** Because of its use of random numbers, a simulated annealing algorithm will provide slightly different results each time it is run. Table 14.5 shows one application of the basic simulated annealing algorithm in IOR Tutorial to the example of a traveling salesman problem depicted in Fig. 14.4. Starting with the same initial trial solution (1-2-3-4-5-6-7-1), use your IOR Tutorial to apply this same algorithm to this same example five more times. How many times does it again find the optimal solution (1-3-5-7-6-4-2-1 or, equivalently, 1-2-4-6-7-5-3-1)?

14.3-3. Reconsider the traveling salesman problem shown in Prob. 14.1-1. Using 1-2-3-4-5-1 as the initial trial solution, you are to follow the instructions below for applying the basic simulated annealing algorithm presented in Sec. 14.3 to this problem.

(a) Perform the first iteration by hand. Follow the instructions given at the beginning of the Problems section to obtain the needed random numbers. Show your work, including the use of the random numbers.

A **(b)** Use your IOR Tutorial to apply this algorithm. Observe the progress of the algorithm and record for each iteration how many (if any) candidates to be the next trial solution are rejected before one is accepted. Also count the number of iterations where a nonimproving move is accepted.

A **14.3-4.** Follow the instructions of Prob. 14.3-3 for the traveling salesman problem described in Prob. 14.2-6, using 1-2-3-4-5-6-7-8-1 as the initial trial solution.

A **14.3-5.** Follow the instructions of Prob. 14.3-3 for the traveling salesman problem described in Prob. 14.2-7, using 1-9-8-10-2-4-3-6-7-5-1 as the initial trial solution.

A **14.3-6.** Because of its use of random numbers, a simulated annealing algorithm will provide slightly different results each time it is run. Table 14.6 shows one application of the basic simulated annealing algorithm in IOR Tutorial to the nonlinear programming example introduced in Sec. 14.1. Starting with the same initial trial solution ($x = 15.5$), use your IOR Tutorial to apply this same algorithm to this same example five more times. What is the best solution found in these five applications? Is it closer to the optimal solution ($x = 20$ with $f(x) = 4{,}400{,}000$) than the best solution shown in Table 14.6?

14.3-7. Consider the following nonconvex programming problem.

$$\text{Maximize} \qquad f(x) = x^3 - 60x^2 + 900x + 100,$$

subject to

$$0 \le x \le 31.$$

(a) Use the first and second derivatives of $f(x)$ to determine the critical points (along with the end points of the feasible region) where x is either a local maximum or a local minimum.

(b) Roughly plot the graph of $f(x)$ by hand over the feasible region.

(c) Using $x = 15.5$ as the initial trial solution, perform the first iteration of the basic simulated annealing algorithm presented in Sec. 14.3 by hand. Follow the instructions given at the beginning of the Problems section to obtain the needed random numbers. Show your work, including the use of the random numbers.

A **(d)** Use your IOR Tutorial to apply this algorithm, starting with $x = 15.5$ as the initial trial solution. Observe the progress of the algorithm and record for each iteration how many (if any) candidates to be the next trial solution are rejected before one is accepted. Also count the number of iterations where a nonimproving move is accepted.

14.3-8. Consider the example of a nonconvex programming problem presented in Sec. 13.10 and depicted in Fig. 13.18.

(a) Using $x = 2.5$ as the initial trial solution, perform the first iteration of the basic simulated annealing algorithm presented in Sec. 14.3 by hand. Follow the instructions given at the beginning of the Problems section to obtain the random numbers. Show your work, including the use of the random numbers.

A **(b)** Use your IOR Tutorial to apply this algorithm, starting with $x = 2.5$ as the initial trial solution. Observe the progress of the algorithm and record for each iteration how many (if any) candidates to be the next trial solution are rejected before one is accepted. Also count the number of iterations where a nonimproving move is accepted.

A **14.3-9.** Follow the instructions of Prob. 14.3-8 for the following nonconvex programming problem when starting with $x = 25$ as the initial trial solution.

$$\text{Maximize} \qquad f(x) = x^6 - 136x^5 + 6800x^4 - 155{,}000x^3 + 1{,}570{,}000x^2 - 5{,}000{,}000x,$$

subject to

$$0 \le x \le 50.$$

A **14.3-10.** Follow the instructions of Prob. 14.3-8 for the following nonconvex programming problem when starting with $(x_1, x_2) = (18, 25)$ as the initial trial solution.

$$\text{Maximize} \qquad f(x_1, x_2) = x_1^5 - 81x_1^4 + 2330x_1^3 - 28{,}750x_1^2 + 150{,}000x_1 + 0.5x_2^5 - 65x_2^4 + 2950x_2^3 - 53{,}500x_2^2 + 305{,}000x_2,$$

subject to

$$x_1 + 2x_2 \le 110$$
$$3x_1 + x_2 \le 120$$

and

$$0 \le x_1 \le 36, \qquad 0 \le x_2 \le 50.$$

14.4-1. For each of the following pairs of parents, generate their two children when applying the basic genetic algorithm presented

in Sec. 14.4 to an integer nonlinear programming problem involving only a single variable x, which is restricted to integer values over the interval $0 \leq x \leq 63$. (Follow the instructions given at the beginning of the Problems section to obtain the needed random numbers, and then show your use of these random numbers.)

(a) The parents are 010011 and 100101.

(b) The parents are 000010 and 001101.

(c) The parents are 100000 and 101000.

14.4-2.* Consider an 8-city traveling salesman problem (cities 1, 2, . . . , 8) where city 1 is the home city and links exist between all pairs of cities. For each of the following pairs of parents, generate their two children when applying the basic genetic algorithm presented in Sec. 14.4. (Follow the instructions given at the beginning of the Problems section to obtain the needed random numbers, and then show your use of these random numbers.)

(a) The parents are 1-2-3-4-7-6-5-8-1 and 1-5-3-6-7-8-2-4-1.

(b) The parents are 1-6-4-7-3-8-2-5-1 and 1-2-5-3-6-8-4-7-1.

(c) The parents are 1-5-7-4-6-2-3-8-1 and 1-3-7-2-5-6-8-4-1.

A **14.4-3.** Table 14.7 shows the application of the basic genetic algorithm described in Sec. 14.4 to an integer nonlinear programming example through the initialization step and the first iteration.

(a) Use your IOR Tutorial to apply this same algorithm to this same example, starting from another randomly selected initial population and proceeding to the end of the algorithm. Does this application again obtain the optimal solution ($x = 20$), just as was found during the first iteration in Table 14.7?

(b) Because of its use of random numbers, a genetic algorithm will provide slightly different results each time it is run. Use your IOR Tutorial to apply the basic genetic algorithm described in Sec. 14.4 to this same example five more times. How many times does it again find the optimal solution ($x = 20$)?

14.4-4. Reconsider the nonconvex programming problem shown in Prob. 14.3-7. Suppose now that the variable x is restricted to be an integer.

(a) Perform the initialization step and the first iteration of the basic genetic algorithm presented in Sec. 14.4 by hand. Follow the instructions given at the beginning of the Problems section to obtain the needed random numbers. Show your work, including the use of the random numbers.

A **(b)** Use your IOR Tutorial to apply this algorithm. Observe the progress of the algorithm and record the number of times that a pair of parents give birth to a child whose fitness is better than for both parents. Also count the number of iterations where the best solution found is better than any previously found.

A **14.4-5.** Follow the instructions of Prob. 14.4-4 for the nonconvex programming problem shown in Prob. 14.3-9 when the variable x is restricted to be an integer.

A **14.4-6.** Follow the instructions of Prob. 14.4-4 for the nonconvex programming problem shown in Prob. 14.3-10 when both of the variables x_1 and x_2 are restricted to be integer.

A **14.4-7.** Table 14.9 shows the application of the basic genetic algorithm described in Sec. 14.4 to the example of a traveling salesman problem depicted in Fig. 14.4 through the initialization step and first iteration of the algorithm.

(a) Use your IOR Tutorial to apply this same algorithm to this same example, starting from another randomly selected initial population and proceeding to the end of the algorithm. Does this application find the optimal solution (1-3-5-7-6-4-2-1 or, equivalently, 1-2-4-6-7-5-3-1)?

(b) Because of its use of random numbers, a genetic algorithm will provide slightly different results each time it is run. Use your IOR Tutorial to apply the basic genetic algorithm described in Sec. 14.4 to this same example five more times. How many times does it find the optimal solution?

14.4-8. Reconsider the traveling salesman problem shown in Prob. 14.1-1.

(a) Perform the initialization step and the first iteration of the basic genetic algorithm presented in Sec. 14.4 by hand. Follow the instructions given at the beginning of the Problems section to obtain the needed random numbers. Show your work, including the use of the random numbers.

A **(b)** Use your IOR Tutorial to apply this algorithm. Observe the progress of the algorithm and record the number of times that a pair of parents gives birth to a child whose tour has a shorter distance than for both parents. Also count the number of iterations where the best solution found has a shorter distance than any previously found.

A **14.4-9.** Follow the instructions of Prob. 14.4-8 for the traveling salesman problem described in Prob. 14.2-6.

A **14.4-10.** Follow the instructions of Prob. 14.4-8 for the traveling salesman problem described in Prob. 14.2-7.

14.4-11. Read the referenced article that fully describes the OR study summarized in the application vignette presented in Sec. 14.4. Briefly describe how a genetic algorithm was applied in this study. Then list the various financial and nonfinancial benefits that resulted from this study.

A **14.5-1.** Use your IOR Tutorial to apply the basic algorithm for all three metaheuristics presented in this chapter to the traveling salesman problem described in Prob. 14.2-6. (Use 1-2-3-4-5-6-7-8-1 as the initial trial solution for the tabu search and simulated annealing algorithms.) Which metaheuristic happened to provide the best solution on this particular problem?

A **14.5-2.** Use your IOR Tutorial to apply the basic algorithm for all three metaheuristics presented in this chapter to the traveling salesman problem described in Prob. 14.2-7. (Use 1-2-3-4-5-6-7-8-9-10-1 as the initial trial solution for the tabu search and simulated annealing algorithms.) Which metaheuristic happened to provide the best solution on this particular problem?