8

Vehicle Routing

In this chapter we consider a typically operational problem, i.e., the optimal planning of routes for a set of vehicles; each vehicle is used for multiple deliveries within its route. Such a problem has a lot of variations and is known as VRP (*Vehicle Routing Problem*). In the simplest version of the problem, we have a set of customers located over some geographic region; each customer should be delivered a given amount of goods. Each customer is associated with a point in the region of interest; we know the distances between any pair of customer locations. Another point of interest is the deposit from which goods must be transported by a fleet of vehicles with limited capacity; the departure point of these vehicles is the deposit, and we also know the distance between the deposit and any customer location. We would like to deliver the required amount to all of the customers at minimum cost; the total cost function can depend, e.g., on the total miles traveled by the vehicles, on the total travel time, or on a combination of both. For the sake of simplicity, in most of the chapter we assume that only mileage is relevant. We are facing a twofold problem: On the one hand, we must assign a subset of customers to each vehicle, subject to capacity constraints; on the other one, we should plan a route for each vehicle, i.e., a sequence of customers, in order to minimize the traveled distance. Typically, such a problem makes sense over relatively short distances and time spans. The amount demanded by each customer is small enough, with respect to vehicles' capacity, to accommodate multiple deliveries; otherwise we would resort to a point-to-point transportation mode.

What we have outlined is just the basic VRP, as there are many complications in practice, in terms of both costs and constraints. Costs can be linked to both space and time; there can be a fixed cost for using a vehicle; as to

constraints, delivery might be subject to time windows; the vehicle fleet may be heterogeneous, and capacity can be multidimensional (volume and weight). Still, even the basic VRP is hard to solve to optimality, unless very sophisticated approaches are used. Hence, we will just describe basic principles that can be used for the development of heuristics. These principles should be regarded as building blocks for heuristics aimed at more realistic versions of VRP. Optimization modeling can also be used, but naive mixed-integer models have weak continuous relaxations; hence, use of commercial branch and bound packages is ineffective and ad hoc strategies must be employed, which are definitely outside of the scope of an introductory book. Still, optimization models can be used to address *parts* of a VRP within clever decomposition strategies (see section 8.3.2).

Since we are interested in distribution, we just deal with deliveries, but the VRP is formally equivalent to a problem in which we want to *collect* goods; a more complicated task pops up when we have a mixed delivery/collection problem, as is the case with some postal services offering package collection to subscribers. Yet another related problem deals with *fixed routing*, in which we have to determine a set of routes which will then be followed regularly. This is more of a tactical than an operational problem. As an example of a more strategic issue, we may consider fleet sizing problems.

VRP is a classic among network routing problems. In section 8.1 we give an introduction to routing problems. If we have one vehicle with infinite capacity, VRP boils down to the classical Traveling Salesperson Problem (TSP). Solution methods for TSP can be somehow adapted to deal with VRP; indeed, TSP is a component of VRP. This is why we devote section 8.2 to illustrate some basic heuristics for solving TSP. Then we use these heuristics as building blocks to cope with basic VRP in section 8.3. Finally, in section 8.4 we illustrate a few complications arising in more realistic versions of VRP.

As a general remark, for the sake of simplicity, in this chapter we assume *deterministic* problems; we do not associate any uncertainty with demand, as we consider short-term operational problems, whereby customers have placed orders and we must just deliver the required goods. However, demand uncertainty can play a role in more tactical problems such as fixed routing. Demand uncertainty may play a role even in the short-term; in fact, there are goods which are not ordered from the warehouse, but it is the driver himself which receives orders on the spot, when visiting retailers (as a practical example, consider how fresh milk and butter are delivered to small retail stores). By the same token, we do not consider uncertainty in the traveling time; in urban transportation, delivery may be heavily affected by traffic jams or accidents.

8.1 NETWORK ROUTING PROBLEMS: THE TSP

Network routing is a general header for a very wide class of problems. Within distribution logistics, we typically adopt network routing models to tackle

service scheduling problem, aimed at finding the optimal use of transportation resources (e.g., trucks) to deliver some goods to a set of customers located on a region, which is modeled as a network.[1]

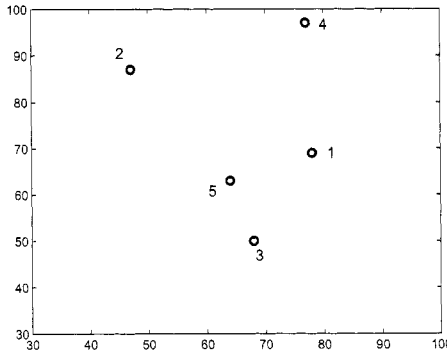
From section 2.2, we recall that a network is a graph with additional information. A graph consists of a set of nodes and arcs. In our case, nodes correspond to locations (retail stores or vehicle deposits). An information which may be associated with each node corresponding to a customer is the amount of demand. Formally, arcs are ordered pairs of nodes; they can be used to represent the possibility of traveling from one node to another one, and the information associated with the arc can be distance, traveling time, or cost. We also recall that a graph can be directed or undirected. In a directed graph, we have oriented arcs, i.e., node pairs are ordered. An oriented arc is typically represented as an arrow, whereas a line is used when the orientation is irrelevant. We should also mention that for undirected graphs we should use the term *vertex*, rather than node, and *edge*, rather than arc, since latter terms are reserved to directed graphs. However, we will use just one pair of terms to keep it simple.

In vehicle routing problems, arcs are oriented if the distance (or traveling time, or cost) from node i to node j does not equal the distance from j to i. This may sound odd, but in a urban transportation problem one-way street may have that effect. On a geographical scale, if the nodes represent Los Angeles and Boston, we may argue that the distance is symmetric.[2] In this chapter we only deal with symmetric problems for the sake of simplicity.

To make things concrete, let us consider the five points depicted in the left part of figure 8.1. Think of those points as cities, or points within a city, that must be visited in order to deliver goods to customers. In the right part of the figure, we give the coordinates of each point, with respect to an arbitrary point of reference. The essential information is the distance between nodes. The real-life distance between two points may be hard to compute, because of roads, natural obstacles, etc.; if we assume that the plain Euclidean distance is a good proxy for distance, we get the distance matrix illustrated in the right part of figure 8.2 (distances have been rounded to the nearest integer). A distance matrix is a handy way to collect distance information. In our case, the distance matrix is symmetric by construction; hence, we may just show the upper triangle of the matrix, as we did in figure 8.2. The left part of the figure illustrates the corresponding network, with undirected arcs depicted as lines joining nodes; in a sense, this representation is abstract, in that node placement in the figure has no physical interpretation.

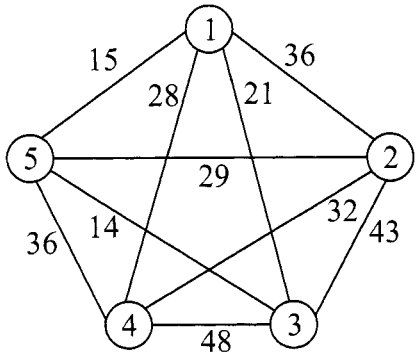
[1] Actually, network routing can refer to quite different problems, such as optimizing the layout in VLSI (Very Large Scale Integrated) circuits.

[2] It is worth noting that costs in transportation problems might not be symmetric even if distances are. If there is more goods flow in one of the two directions, e.g., from Detroit to New York, demand/offer mechanisms may induce asymmetric transportation fares.



Point	X	Y
1	78	69
2	47	87
3	68	50
4	77	97
5	64	63

Fig. 8.1 Map and coordinates of five points on a region.



	1	2	3	4	5
1	–	36	21	28	15
2		–	43	32	29
3			–	48	14
4				–	36
5					–

Fig. 8.2 Network and distance matrix for a symmetric network with five nodes.

Given a network, a **node routing** problem calls for finding the optimal way of visiting nodes, with respect to a given criterion, subject to certain restrictions. A prototypical node routing problem is the *Traveling Salesperson Problem*, or TSP for short. In this problem, the salesperson lives in a city and must visit all of the other cities in the network before coming back home; each city (or customer) must be visited once. The traveling route is closed, and it is typically referred to as a tour. Among the many possible tours, she would like to find one with a minimal total traveled distance. The network in figure 8.2 may represent represents a simple TSP, whereby the salesperson lives in city 1 and must visit the other four cities in a clever sequence. The distance matrix can be interpreted literally, but it could also represent travel times; whatever the case, we interpret the labels associated to arcs as *costs*; the cost may also depend on both time and space. The cost for going from city i to city j is c_{ij}, and we have already noted that the matrix in the figure

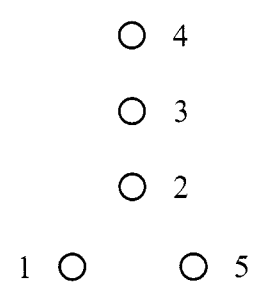


Fig. 8.3 Simple network to illustrate issues in representing network distances.

is symmetric. When $c_{ij} = c_{ji}$, for all i and j, we have a symmetric TSP; otherwise we have an asymmetric TSP, also denoted by ATSP.

As we pointed out, one of the cities should be regarded as the city where the salesperson lives and must get back to; however, due to the cyclical nature of the problem, which city is the starting point is irrelevant. For instance, the tours $(1, 2, 3, 4, 5, 1)$ and $(2, 3, 4, 5, 1, 2)$ have the same total length. As a consequence, if there are n cities, there are $n!$ permutations of them, but only $(n-1)!$ possible solutions for the TSP; actually, in the symmetric case, there are only half of that, because we may travel any tour in two ways, obtaining the same mileage. Formally, a cycle visiting all of the nodes of a graph exactly once is called a *Hamiltonian cycle*. So, TSP calls for finding the shortest Hamiltonian cycle.

The restriction that a city must be visited exactly once may sound illogical. After all, if three cities (say i, k, and j) are geographically arranged on a line, it may be advantageous to travel from the first one, to the second one, then to the third one, and finally travel back to the first city passing through the second one. In more concrete terms, if there is a convenient freeway joining three cities, it might well be the case that we travel twice through a city. Consider for instance the network of figure 8.3, and assume for simplicity that all of the distances between neighboring cities are 1. The optimal solution of the TSP is obviously to start from city 1; go through cities 2, 3, and 4; then go to city 5, going through 3 and 2 again; finally, get back home. This may not look like a Hamiltonian cycle, but it is if we build the network in a more abstract way. From the point of view of an abstract network, like the one in figure 8.2, we travel from city 4 to city 5 "directly," along an arc of length 3, which is the sum of the distances between cities 4 and 3, 3 and 2, and finally 2 and 5. This may be the only way of reaching city 5 from city 4, or maybe just the optimal way. The bottom line is that the abstract network representation includes a "full" distance matrix, with no empty entries even though some cities are not directly linked. The distance matrix consists of *optimal* distances between pairs of nodes.

The distance matrix, since it is an "optimal" distance matrix rather than the direct translation of a map, must satisfy a rather obvious requirement,

which is called **triangularity** property:

$$c_{ij} \leq c_{ik} + c_{kj}. \tag{8.1}$$

The distance from i to j cannot be larger than the distance from i to k plus the distance from k to j. We may have an equality in the aforementioned case of three cities arranged in sequence on a line. If we are dealing with a full matrix of triangular distances, we may look for a Hamiltonian cycle in the associated network; a city will be visited twice if it is optimal in the real world, but we do not see this on the abstract network model, which should not be taken too literally.

We consider solution methods for the symmetric TSP in section 8.2. Then, in section 8.3, we generalize the TSP by associating a demand information with each node. If customers must be served by finite-capacity vehicles, it is unlikely that all of the customers may be served by just one tour. If the overall demand cannot fit one vehicle, we must use multiple tours or multiple vehicles. This generalization leads to the Vehicle Routing Problem, which is the core of this chapter. However, we should at least mention the existence of other network routing problems, while referring to [3] for a full account of network routing.

8.1.1 Other network routing problems

In this chapter we only consider very basic symmetric node routing problems, but it is worth noting that node routing problems have lots of applications outside the logistics field. A symmetric TSP can be used to find the optimal path planning for a robot which has to visit a set of points in space to take measurements or to carry out spot welding operations. An asymmetric TSP can be used to model sequence-dependent setup times in a machine scheduling problem; if you produce black paint after a batch of white paint, maybe you do not need to wash the machine too accurately; going the other way around is not that easy, as producing white paint after a batch of black one requires a thorough setup. Similar considerations apply when producing vermouth or, in the textile industry, when we deal with both cheap wool and cashmere. A few concepts we use in solving symmetric problems may also be used to cope with asymmetric problems, but the latter typically require more care, depending on the solution algorithm we use.

It should also be mentioned that sometimes we have to cope with *arc* (or edge) routing problems. Consider a postman in charge of visiting all houses within a portion of a city. Since houses are arranged linearly along streets, it may be much better to represent his problem as the one of visiting all of the arcs at least once, rather than the nodes (which are used in this setting to represent crossroads). Ideally, the postman should visit all of the arcs once, along what we call a *Eulerian* cycle. Actually, a strictly Eulerian cycle may

not exist.[3] The prototypical arc routing problem is the Chinese Postman Problem, i.e., the problem asking for the shortest tour of a graph which visits each arc (actually, edge) at least once.

Finally, we associate arcs with either time, space, or cost information. When dealing with very complex transportation scheduling problems, one may develop a space–time network. On such a network, some arcs represent movement in space and other arcs represent movement in time. This modeling framework is important if we want to manage, e.g., the flow of freight wagons on a railroad network (see [16]). Further complications arise when you also consider the many constraints you may have on the crews to be scheduled on trains or aircrafts.

8.2 SOLUTION METHODS FOR SYMMETRIC TSP

In this section we describe basic heuristic principles for the solution of symmetric TSP. The principles we illustrate are not the most advanced ones, but they are useful to build intuition and pave the way for the development of heuristics aimed at VRP. Conceptually, TSP is a trivial problem: Find the best sequence of stops in a set of cities. Mathematically, we have to find the best solution within a finite set of permutations of "cities." We could simply enumerate all of them and spot the best one. Unfortunately, such a simple-minded approach is not practically feasible but for very small problem instances. If we have 25 cities, there are $24!/2 \approx 3.1 \cdot 10^{23}$ alternative solutions. Assuming that we are able to generate and evaluate one billion solutions per second, it would take something like 9.84 million years to get the optimal tour. If you have to dispatch a fleet of vehicles each and every morning, you need a seriously faster decision approach.

In section B.6.1 we illustrate the branch and bound method as a way to solve optimization problems with a combinatorial component, without resorting to complete enumeration. In principle, we could build a mixed-integer linear programming model with binary variables modeling the sequencing decisions and use a good commercial solver implementing LP-based branch and bound. However, we have also pointed out that the efficiency of these methods relies on the quality of lower bounds; simple TSP model formulations have very weak relaxations, and unless very sophisticated and ad hoc modeling frameworks and solution methods are used, finding the optimal solution is very hard. We will not pursue such approaches, which are hardly available in commercial software, as we prefer to illustrate some *principles* which lend themselves to generalizations when coping with additional constraints that are important for a real-life VRP. Anyway, we should keep in mind that we

[3]Many of us have checked this as children, trying to draw certain geometric figures always keeping the pencil in contact with the paper, without passing twice on the same segment.

could ask someone to come up with an algorithmic black box able to solve a TSP to optimality for not-too-large problems; this can be handy in devising decomposition-based methods.

There is a huge literature on solution methods for TSP, but the methods we consider here can be broadly classified into two categories:

1. **Constructive** methods aim at building a tour by expanding a partial route according to some reasonable criterion; such methods build *one* solution directly. We illustrate two basic constructive approaches in sections 8.2.1 and 8.2.2.

2. **Iterative** methods start from a given solution and try to improve the initial tour by generating a *sequence* of alternative solutions; clearly, iterative methods are more time-consuming and require a constructive method to get a starting point. Nevertheless, the resulting gain in solution quality may be remarkable. We outline iterative methods based on local search in section 8.2.3.

8.2.1 Nearest-neighbor heuristic

The *nearest-neighbor* heuristic is arguably the simplest heuristic that may come to mind to solve TSP. We select a city acting as a starting point, and we grow a partial sequence by appending cities at the end of it. To select the next city to visit, we always choose the closest one to the last city we visited (ruling out those we have already visited). Then, after visiting all of the cities, we close the route by going back to the starting point.

The procedure can be formally stated as follows:

Step 0: initialization. Let $\mathcal{N} = \{1, 2, 3, \ldots, n\}$ be the set of cities we want to visit. Choose a starting point $i^\circ \in \mathcal{N}$; let $\mathcal{V} = \mathcal{N} \backslash i^\circ$ be the set of cities we still have to visit and let $\mathcal{S} = (i^\circ)$ the current partial sequence.[4]

Step 1: choose the next city. Let i^l be the last city in the partial sequence $\mathcal{S}$. Find the closest city j^* in $\mathcal{V}$, i.e., solve $\arg\min_{j \in \mathcal{V}} c_{i^l,j}$. If there are alternative optima, break ties arbitrarily.

Step 2: expand partial sequence. Append city j^* at the end of the partial sequence $(\mathcal{S} \leftarrow (\mathcal{S}, j^*))$ and cancel it from the set of cities yet to visit $(\mathcal{V} \leftarrow \mathcal{V} \backslash j^*)$.

Step 3. If $\mathcal{V} = \emptyset$, i.e., there is no city left to visit, close the route by appending the initial city at the end of the sequence $(\mathcal{S} \leftarrow (\mathcal{S}, i^\circ))$; otherwise, go to step 1.

[4]We recall that the \ operator denotes set difference.

Example 8.1 Let us apply the nearest-neighbor heuristic to the problem of figures 8.1 and 8.2, starting from city 1. The closest city to 1 is city 5, and the partial sequence so far is $(1, 5)$. Among the remaining cities, the closest one to city 5 is 3; the partial sequence is expanded to $(1, 5, 3)$. From city 3 we should go to city 2, and finally we have to terminate the sequence with city 4. The complete tour is $(1, 5, 3, 2, 4, 1)$, with total length 132.

Actually, it is easy to see from the figures that this is not the optimal solution. From the map, the tour $(1, 3, 5, 2, 4, 1)$ looks more sensible; indeed, its length is 124, and it turns out that this is really the optimal tour. In this trivial case, we see quite clearly what is wrong with the nearest-neighbor: We should have gone from node 1 to node 3, but we were too greedy. We may also see that the method might yield different solutions, depending on the starting point. If we start from city 5, we get the tour $(5, 3, 1, 4, 2, 5)$, which is equivalent to $(1, 3, 5, 2, 4, 1)$. We could try all of the possible starting points and keep the best result. However, even this cannot guarantee the optimality of the solution we get. □

The nearest-neighbor heuristic is conceptually simple, easy to implement, and quite fast. The bad news is that it is a *greedy* heuristic, and there is no guarantee on the optimality of the solution we get. Choosing what looks best for the current decision we have to make (select the next city) does not ensure the optimality of the whole tour. A clear danger is disregarding some inconvenient city, leaving it to the last steps of the procedure. The quality of the overall solution can thus deteriorate significantly, as the inconvenient city (which may demand a substantial cost to visit) is going to be inserted at the last step of the procedure; this means that the most critical city is practically inserted in a random position in the tour.

8.2.2 Insertion-based heuristics

The nearest-neighbor approach has many obvious limitations, which we have already mentioned. An additional one is the fact that it allows us to append a city only at the *end* of the current sequence. We could allow insertions in any point in the sequence. Since we must get back to the starting point, it would be even better to expand a closed route, rather than an open sequence that we close at the last step of the procedure. This idea leads to insertion heuristics, which are still very simple. At each step of the algorithm, we have a set $\mathcal{V}$ of residual cities to visit and a partial tour $\mathcal{T}$; what we need is to select an arc (i, j) in $\mathcal{T}$, which should be "opened" to allow insertion of a new city between i and j, leading to a subsequence (i, k, j). Actually, given a partial route, we have to make two decisions:

1. which city $k \in \mathcal{V}$ to insert in $\mathcal{T}$;
2. the insertion point, i.e., between which cities i and j already in $\mathcal{T}$ we should insert k.

Since we assume that the triangularity property holds, inserting a new city can only increase the total length of the current partial route. Hence, a reasonable criterion is to make decisions in such a way as to minimize the incremental cost of the insertion. The incremental cost of inserting city k between i and j is

$$c_{ik} + c_{kj} - c_{ij}. \tag{8.2}$$

This additional length is typically called **extra mileage**.

The first point we must take care of is how to find the initial partial route. One possibility is selecting the shortest arc (i, j) and let $\mathcal{T} = (i, j, i)$ be the initial partial route. To find the next city to insert in the partial route, we may search $\mathcal{V}$ for the closest city to $\mathcal{T}$, i.e., we may solve

$$\min_{i \in \mathcal{T}, k \in \mathcal{V}} c_{ik}.$$

Then, given the new city (breaking ties arbitrarily), we may look for the best insertion point by minimizing extra mileage. The procedure is repeated until we have the complete tour.

Example 8.2 Let us consider the TSP of figure 8.2 again. The are two cities in the initial route. Choosing the shortest arc in the network, we set the initial route as $\mathcal{T} = (3, 5, 3)$. The closest city to those included in $\mathcal{T}$ is city 1. For now, there is no substantial degree of freedom in choosing the insertion point, and we update the partial route $\mathcal{T} = (3, 5, 1, 3)$. This route is equivalent to $\mathcal{T} = (3, 1, 5, 3)$, since the problem is symmetric and the way we travel the tour is irrelevant.

Now the closest city to those in $\mathcal{T}$ is city 4, since its distance from city 1 is 28, whereas the distance between cities 2 and 5 is 29. Now we must find the optimal insertion point among the three following possibilities:

$$\begin{aligned} c_{34} + c_{45} - c_{35} &= 48 + 36 - 14 = 70, \\ c_{54} + c_{41} - c_{51} &= 36 + 28 - 15 = 49, \\ c_{14} + c_{43} - c_{13} &= 28 + 48 - 21 = 55. \end{aligned}$$

Hence, we set $\mathcal{T} = (3, 5, 4, 1, 3)$. Note that there is no need to reevaluate the whole tour after insertion, as only the incremental cost of the insertion is needed to make the decision. Finally, we have to accommodate city 2:

$$\begin{aligned} c_{32} + c_{25} - c_{35} &= 43 + 29 - 14 = 58, \\ c_{52} + c_{24} - c_{54} &= 29 + 32 - 36 = 25, \\ c_{42} + c_{21} - c_{41} &= 32 + 36 - 28 = 40, \\ c_{12} + c_{23} - c_{13} &= 36 + 43 - 21 = 58. \end{aligned}$$

The final route we get is $(3, 5, 2, 4, 1, 3)$, with total length 124. □

In this case, we get the optimal solution, but this is not guaranteed in general, as the insertion-based heuristic is still a greedy heuristic. We could represent

our basic constructive procedures as a greedy way to explore a search tree, a concept that we introduce in section B.6.1 on branch and bound methods. In a branch and bound method, we prune a branch of the search tree only if we are sure that it cannot lead to an optimal solution. In greedy heuristics, we basically select the most promising branch, forgetting about the others. However, we could reduce the myopic behavior of greedy heuristics by adopting a look-ahead strategy, whereby we explore the consequence of a choice by examining its consequences a few steps further. A further issue concerns breaking ties when we have to make a decision. In insertion-based procedures, we might have two insertion points with the same extra mileage; in the nearest-neighbor heuristic, we may have two or more cities with the same distance from the last one in the partial sequence. In such a case, we could explore the consequences of each alternative a bit deeper in the search tree, rather than breaking ties arbitrarily and take a basically random branch.

In the specific case of the insertion-based approach above, we may also try to improve results, at some additional computing cost, by considering all possible pairs consisting of a new city to insert and its insertion point. In fact, in the procedure above we select a city, and then we explore possible insertion points; we could find the optimal insertion point for each city, and only after evaluation of the result we make a decision. Another variation on the theme is the choice of the initial two-city tour; we could start from the two farthest cities, rather than from the closest pair.

8.2.3 Local search methods

The two approaches we have just considered are constructive, in that they directly build one solution, with a possibly greedy logic. An alternative consists of examining a *sequence* of solutions. The basic idea is trying to improve a given solution using some simple recipe. We can perturb the solution according to a predefined set of rules, which define a *neighborhood* of the current solution: the name stems from the fact that we just apply small changes to the current solution. For instance, since a TSP solution is basically a permutation of cities, we could consider swapping pairs of cities in the tour. Having defined the neighborhood structure, we may look for the best solution within the neighborhood of the current tour. This new candidate solution may be an improvement or not. In the first case, we set the candidate as the new current solution and we repeat the procedure; otherwise we stop.

This very simple approach is called **iterative improvement** and is the simplest example of a large family of methods collectively called **local search** methods. Since we only search locally in the neighborhood of the current solution, we might well get stuck in a locally optimal solution that is far less performing than the globally optimal one. We should note that "locally" means "with respect to the neighborhood structure." In figure 8.4 we illustrate the issue conceptually. If we are minimizing a nonconvex cost function $f(x)$, and we are at point x_L, there is no way to escape from this local optimum and

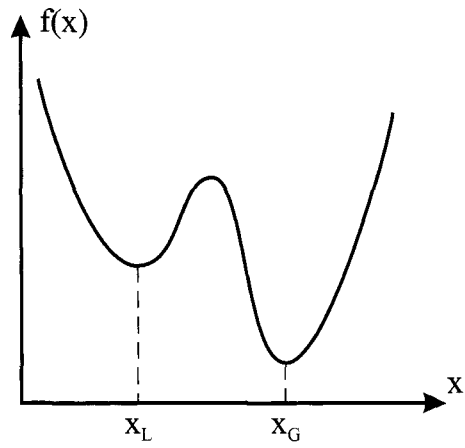


Fig. 8.4 Getting stuck in a local minimum.

get to the global optimum x_G, if we just look to the left and to the right and accept only improving steps. In local search, we cannot really draw a picture like that, because we are moving within a space whose *discrete* points are, e.g., tours on a network; nevertheless, with respect to some "weird" topology, we may have lack of convexity in the cost function, possibly leading a local improvement procedure into bad local optima. Clearly, there is a tradeoff between computational requirements and the richness of the neighborhood structure (in the limit, a somewhat expensive neighborhood could require the complete enumeration of the feasible solutions). On the one hand, defining a small neighborhood is very efficient computationally, but it can leave us in a very bad local optimum. On the other hand, a very rich neighborhood structure opens many more search paths, but it can be too demanding from a computational point of view.

Indeed, the art of local search consists of devising a parsimonious, yet effective neighborhood structure. For instance, in the TSP case we could swap pairs of consecutive cities in the sequence, which is a rather limited neighborhood structure. A richer, and quite effective, neighborhood structure is known as 2−opt. Given a complete tour, we consider all pairs of nonconsecutive arcs. They are canceled and substituted by two alternative arcs in such a way that we obtain another tour. The idea is illustrated in figure 8.5. We see that the two canceled arcs are substituted by arcs "crossing" each other (remember that the network we draw need not be taken as a pictorial representation of the underlying geography). The idea can be generalized by canceling k arcs and replacing them in all possible ways. The k−opt approach, for $k > 2$, tends to get more complex and time-consuming, and significant advantages in terms of quality are not guaranteed.

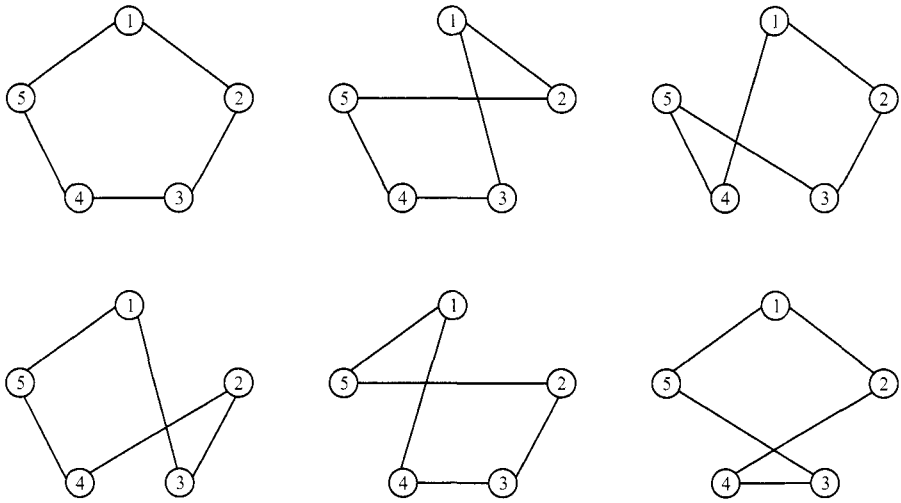


Fig. 8.5 An example of neighborhood generated by the 2−opt rule; the current solution is depicted in the upper-left corner.

Example 8.3 We consider once more the TSP of figure 8.2, and we tackle it by a 2−opt approach starting from the tour $(1, 2, 3, 4, 5, 1)$, whose total length is 178. We must compute the total length of each neighboring tour, as depicted in figure 8.5:[5]

$$(1, 2, 5, 4, 3, 1) \rightarrow 170,$$
$$(1, 2, 3, 5, 4, 1) \rightarrow 157,$$
$$(1, 3, 2, 4, 5, 1) \rightarrow 147,$$
$$(1, 4, 3, 2, 5, 1) \rightarrow 163,$$
$$(1, 2, 4, 3, 5, 1) \rightarrow 145.$$

The best tour in this set is $(1, 2, 4, 3, 5, 1)$, which gets to be the new current tour. Then we evaluate the new neighborhood:

$$(1, 2, 5, 3, 4, 1) \rightarrow 155,$$
$$(1, 2, 4, 5, 3, 1) \rightarrow 139,$$
$$(1, 4, 2, 3, 5, 1) \rightarrow 132,$$
$$(1, 3, 4, 2, 5, 1) \rightarrow 145,$$
$$(1, 2, 3, 4, 5, 1) \rightarrow 178.$$

[5]From an implementation point of view, this task can be made extremely efficient by proper use of data structures, also avoiding the recomputation of total length from scratch and just evaluating an incremental cost; see [18].

Note that the last solution in this neighborhood is just the initial tour. The new current solution is $(1, 4, 2, 3, 5, 1)$, with total length 132. Repeating the procedure one more time, we get

$$(1, 4, 5, 3, 2, 1) \rightarrow 157,$$
$$(1, 4, 2, 5, 3, 1) \rightarrow 124,$$
$$(1, 2, 4, 3, 5, 1) \rightarrow 145,$$
$$(1, 3, 2, 4, 5, 1) \rightarrow 147,$$
$$(1, 4, 3, 2, 5, 1) \rightarrow 163.$$

We leave to the reader the task to verify that no further improvements can be obtained. Since we cannot find any improving tour, the algorithm stops. □

In this lucky example, we actually end up with the optimal solution, but we do not know that (in this small case, we may prove that 124 is the optimal length by complete enumeration). In general, this does not happen, and the solution we stop at may depend on the initial tour. The difficulty is that the search process may get stuck into a local optimum, and there is no way out because we only accept improving perturbations (see figure 8.4). There are a couple of ideas that may come to our mind to overcome this difficulty:

- We could start the search from different initial tours, possibly generated by alternative constructive heuristics or by random generation. The idea of generating multiple starting points randomly leads to GRASP (**Greedy Randomized Adaptive Search Procedure**) methods.

- We may try to overcome the tendency to get stuck in local optima by allowing nonimproving perturbations according to a sensible strategy. In fact, looking back at figure 8.4, we see that in order to travel from x_L to x_G, we must accept a temporary increase in cost.

The last idea has lead to a fairly wide family of local search approaches, which we just outline below, referring the interested reader to references at the end of the chapter.

- In **simulated annealing**, optimization is interpreted as an energy minimization process. In classical mechanics, a physical system evolves in such a way as to minimize its energy: A ball subject to gravity force will roll into a hole, minimizing its potential energy, and will stay there. There is no way a ball can pop up from the hole all by itself. In optimization terms, this means that if the ball rolls into a local minimum, it gets stuck there. In Statistical Mechanics, under the effect of thermal noise, there is some probability that a system will find itself in a higher energy state without external intervention. The probability of this upward jump increases with temperature and decreases with the size of the jump, i.e., the energy difference between the two states. Annealing is a

technological process whereby a material is slowly cooled, allowing it to escape from local minima and to reach a lower energy level. If we cool the material too fast, we get a glass; if the cooling process is slow, we get a good crystal structure when the final temperature is so low that the system cannot change configuration anymore. Simulated annealing exploits this idea for optimization, allowing nonimproving perturbations according to a stochastic mechanism. Given a current solution with cost C_{old}, we randomly sample an alternative solution in its neighborhood, with cost C_{new}. The alternative solution is accepted with probability given by

$$\min\left\{1, \exp\left[\frac{-(C_{\text{new}} - C_{\text{old}})}{T}\right]\right\},$$

where T is a control parameter acting as a temperature, which is decreased according to a cooling schedule. We see that at high temperatures, the search process is free to wander and explore the solution space, whereas at low temperatures it works just like local improvement. When the algorithm freezes, the best solution visited will be reported.

- Another idea for a stochastic search mechanism is mimicking biological evolution, rather than statistical mechanics. In **genetic algorithms**, unlike other local search mechanisms, we work on a *population* of solutions. Only the best members within the current population have a high chance of surviving: The current population evolves by crossover (offspring are created from two parents) and mutation (a random perturbation is applied) mechanisms, whereby probability of selection and survival depends on the quality of each solution. In this case, we need a way to map a solution to a data structure, which works like a chromosome, whose genes are the features of a solution (or the parameters of an algorithm to build a solution). The mechanisms for crossover and mutation define the neighborhood structure for this stochastic search algorithm.

- Maybe the most widely applied local search mechanism, as far as TSP and VRP are concerned, is **tabu search**. This approach, unlike the previous two, need not be stochastic. The rationale is that the best solution in the neighborhood of the current one should be accepted, in order to escape from local minima, while biasing the search process towards good solutions. The trouble with this simple idea is that cycling is most likely to occur: When escaping from a local minimum, we accept a nonimproving alternative, but the best solution in the neighborhood of this new solution may well be the previous local optimum. To avoid cycling, we use a data structure to store some attributes of each solution we visit, or some feature of the perturbations we apply to get them. This data structure works as a tabu list, which forbids revisiting solutions or applying perturbations undoing what we have just accomplished. The

tabu list is a sort of short-term memory, as only the most recent tabu attributes are kept there, in order to avoid restricting the search process too much. Long-term memory mechanisms have been proposed to improve the ability of diversifying search by exploring new regions of the solution space.

Local search algorithms look conceptually simple, but in fact getting them to work properly requires a fair amount of skill and ingenuity, not to mention experience. Defining a good neighborhood structure, as well as setting the parameters governing the algorithm, is not trivial. To get a feeling for the subtle issues we may have to face, consider the application of the 2−opt neighborhood structure to an asymmetric TSP. If we cross arcs, like we did in figure 8.5, the consequence is that we actually invert part of the sequence; in other words, part of the tour is traveled clockwise rather than counterclockwise (and vice versa). This is not relevant in the symmetric case, but when the distance matrix is not symmetric, the new solution may be radically different from the previous one. We face a similar issue when dealing with time-windows in a VRP; even if distances do not change, changing the time instants at which we visit customers may have adverse effects. In practice, some knowledge of network and graph optimization may be needed in order to find a good heuristic for a complex case; common sense is not always enough.

8.3 SOLUTION METHODS FOR BASIC VRP

VRP is a generalization of TSP, accounting for multiple vehicles whose routes are subject to additional constraints. There is a set of n customers; each customer is located on a node in a network. To serve customers, we have a fleet of vehicles located in node 0. We consider a fleet of homogeneous vehicles, each featuring the same capacity, and one deposit; real-life problems may require relaxing such assumptions. A known demand d_i, $i = 1, \ldots, n$, is associated with customer i. Demand need not be necessarily associated to one item type; what is really important is that demand is measured in the same units as vehicle capacity. Just like in TSP, we would like to minimize distance traveled (or time, or cost); but unlike TSP, each vehicle has a finite capacity, in terms of volume and/or weight. This is what creates the need for multiple vehicles and/or multiple routes, because we cannot serve all of the customers with one route. We have to develop a set of routes, starting and terminating at the deposit, which can be carried out sequentially by one vehicle, or in parallel by a set of vehicles.

Given such assumptions, our input data are a symmetric distance (or travel cost) matrix, the demand per customer, and the vehicles' capacity; the number of vehicles may be given or not, depending on the specific assumptions about the way routes are carried out. We want to find a set of routes minimizing total distance traveled, subject to vehicle capacity constraints. In the

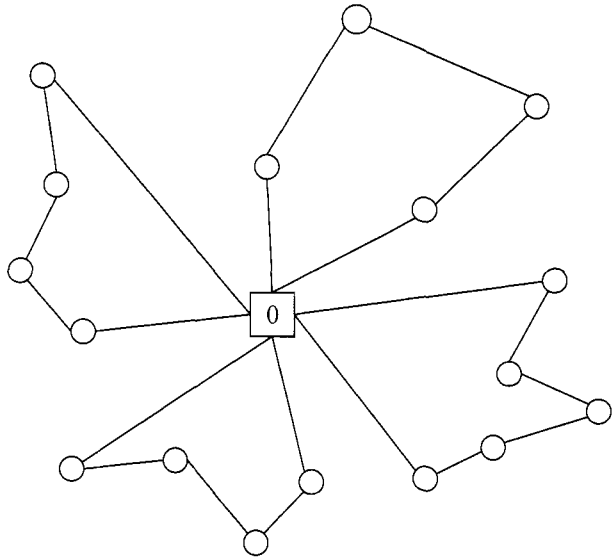


Fig. 8.6 Example of solution of a VRP.

basic VRP, we do not consider additional constraints such as the maximum route duration, which is just another capacity constraint, or time windows for serving customers. Another important simplification is that we sequence customers within each route, but we do not really schedule routes. For instance, suppose that early in the morning we devise five routes. Any route can be executed within the current working day, but we have just four vehicles. Hence, we should decide which routes should be carried out today, and which one will be carried out tomorrow. Clearly, this may depend on priorities associated with customers; alternatively, we could try to devise two routes that can be carried out by the same vehicle within one working day, by returning to the deposit between the two routes. We see that such timing issues might be rather complicated. In the basic VRP, we either assume that the number of vehicles is unlimited, or we try to find a solution serving all of the customers with a given number of vehicles, reporting infeasibility otherwise.

Despite all of these severe limitations, the basic VRP is a tough problem, and tackling it paves the way for solution of more realistic versions. Figure 8.6 illustrates one solution of a VRP. The figure points out the twofold nature of VRP. The solution consists of two elements, since each route consists of a subset of customers and the sequence according to which they are visited by the vehicle. Given the first element, we have one TSP per vehicle. This can be exploited in decomposition strategies; it also suggests that TSP heuristics can provide some basic principles to tackle VRP as well. VRP heuristics, too, can be constructive or iterative. We do not consider local search methods for VRP, because devising neighborhood structures coping with both dimensions

of the problem (i.e., allocation of customers to routes and sequencing within each route) is not trivial, even though the effort in doing so can be quite rewarding.

8.3.1 Constructive methods for VRP

Constructive methods for VRP are based on the idea of growing routes according to various patterns and based on various criteria.[6] To classify constructive methods, we should begin by drawing the line between

- sequential algorithms, in which one route is grown at a time, until all customers have been routed, and
- parallel algorithms, in which several routes are grown together.

Parallel algorithms, in turn, can be classified into two subcategories:

1. We may start from a set of small routes, one per customer, and we proceed by merging routes. The procedure stops when vehicles' capacities prevent us from coalescing routes. One clear disadvantage of this approach is that we have no control over the number of routes we end up with, which may be larger than the number of available vehicles.
2. In order to overcome the aforementioned disadvantage, we may fix the number of routes a priori, say m. The number of routes can be the number of vehicles we plan to use. Typically, we use m well-selected customers to devise an initial set of "seed" routes, each one consisting of one customer. Then we proceed by selecting one customer at a time, which is inserted in one of the m growing routes.

Finally, we have to specify the criteria we use in growing routes. There are many of them, but we illustrate the two fundamental ones by referring to figure 8.7.

- The **savings** criterion. The rationale behind the savings criterion is that if two customers, say i and j, are served by two vehicles along separate routes, the two vehicles have to drive from the deposit to the customer and back. Hence, the total traveled distance amounts to $c_{0i}+c_{i0}+c_{0j}+c_{j0}$. If the two routes are merged and the two customers are served by the same vehicle, the new total length will be $c_{0i}+c_{ij}+c_{j0}$, with a saving $s_{ij} = c_{i0} + c_{0j} - c_{ij}$. Referring to figure 8.7, we cancel the two dashed arcs, replacing them with arc (i, j). Actually, the argument, as it is stated, applies only to routes consisting of one customer visit. In fact, it can be applied more generally, provided that customers i and j are the first or the last on their respective routes (see figure 8.8; remember that

[6] This section relies heavily on material from [7].

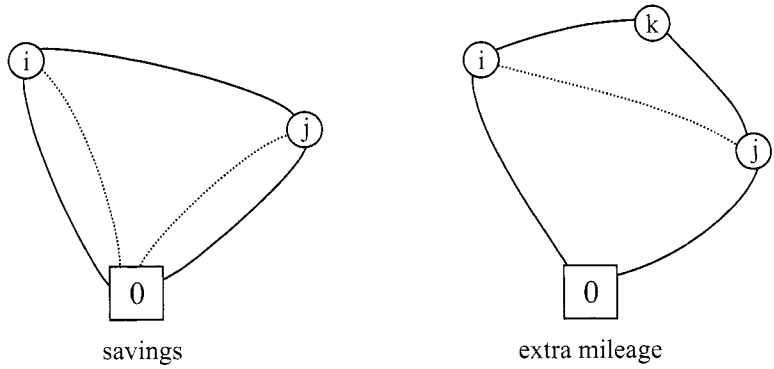


Fig. 8.7 Illustrating savings and extra-mileage criteria.

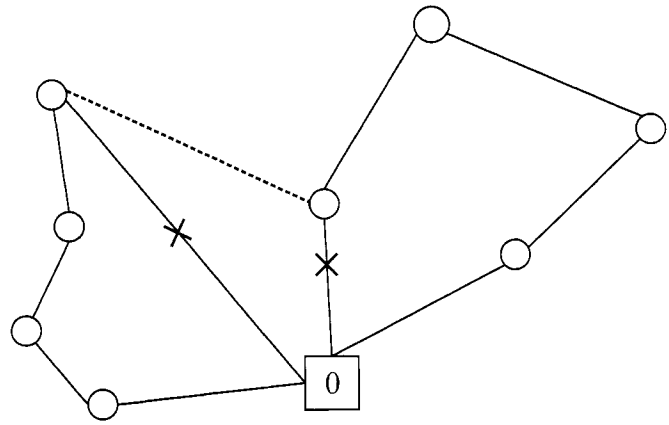


Fig. 8.8 Merging two partial routes by the end points.

we are just considering symmetric problems; hence, being last or first in the route is actually the same thing). This idea can be used to merge partial routes together, provided that capacity constraints are satisfied; according to this metric, we should give priority to the merger with the largest saving. Another relevant point is that by merging routes, we decrease the number of required vehicles.

- The **extra-mileage** criterion. We have already met the extra-mileage criterion when discussing insertion-based heuristics for TSP (see section 8.2.2). Here the idea is inserting customer k on the path from customer i to j, incurring an increase of the route length given by $e_{ikj} = c_{ik} + c_{kj} - c_{ij}$. Referring again to figure 8.7 (right side), we get rid of the dashed arc and insert two arcs, in such a way that extra mileage e_{ikj} is minimal.

Table 8.1 Distance matrix and customer demand for example 8.4

c_{ij}	0	1	2	3	4	5	6	7
0	–	4	2	4	3	3	5	6
1		–	2	7	4	6	5	3
2			–	5	4	4	3	4
3				–	3	1	6	9
4					–	3	7	7
5						–	5	8
6							–	5
7								–

i	1	2	3	4	5	6	7
d_i	9	6	14	8	9	6	5

Table 8.2 Savings matrix for example 8.4

r_{ij}	1	2	3	4	5	6	7
1	–	4	1	3	1	4	7
2		–	1	1	1	4	4
3			–	4	6	2	1
4				–	3	1	2
5					–	3	1
6						–	6
7							–

To illustrate the concepts above in a concrete setting, we may describe an early algorithm for VRP, known under the names of Clarke and Wright.[7] The method is based on the savings criterion, and it is a parallel algorithm of the first type, i.e., it is based on the coalescence of smaller routes.

Example 8.4 Clarke–Wright's algorithm is best illustrated by a small example, whose input data are displayed in table 8.1. The distance matrix is symmetric and we assume that vehicles' capacity is 20. To begin with, we may compute a savings matrix, with an entry for each pair of customers; the result is reported in table 8.2. Since we always join customers when they are placed at an endpoint of a route, this savings matrix can be computed once for all. Actually, not all of its entries are relevant: For instance, customers 1 and 3 cannot be served by the same vehicle, because their total demand is $9 + 14 = 23$, which exceeds vehicle capacity. The starting set of routes is

$$(0,1,0); \quad (0,2,0); \quad (0,3,0); \quad (0,4,0); \quad (0,5,0); \quad (0,6,0); \quad (0,7,0).$$

[7]See the original reference [8].

We see from table 8.2 that 7 is the largest saving, and it is obtained by joining customers 1 and 7, leading to the new set of routes:

$$(0,1,7,0); \quad (0,2,0); \quad (0,3,0); \quad (0,4,0); \quad (0,5,0); \quad (0,6,0).$$

Then the table shows that two savings amount to 6, but the one associated with customers 3 and 5 is not compatible with vehicle capacity. We should check if joining customers 6 and 7 is feasible, since the latter customer is already on the same route as customer 1; the total demand for these three customers equals the vehicle capacity; hence, we may get rid of customers 1, 6, and 7. Now, current routes are

$$(0,1,7,6,0); \quad (0,2,0); \quad (0,3,0); \quad (0,4,0); \quad (0,5,0),$$

The best feasible option is merging customers 4 and 5, with a saving of 3. Joining them, we get

$$(0,1,7,6,0); \quad (0,2,0); \quad (0,3,0); \quad (0,4,5,0).$$

Now, neither customer 2 nor customer 3 fits the route (0,4,5,0); all we can do is merging customers 2 and 3, which yields the final set of routes:

$$(0,1,7,6,0); \quad (0,2,3,0); \quad (0,4,5,0).$$

□

Clarke and Wright's algorithm is conceptually quite easy, and it played a prominent historical role, but it suffers from a few limitations. To begin with, when we merge routes, we do so only by joining a pairs of customers at the endpoints of their respective route (see figure 8.8). Maybe, inserting new customers in arbitrary points of a route could be advantageous. Furthermore, there is no control over the number of routes we end up with; in the example above, we could not use less than three vehicles anyway, but in general, if we have a given number of vehicles and we have to serve all of the customers in parallel, we would like to make sure that the number of routes is kept under control.

We can also exploit the ideas behind the insertion-based heuristic for TSP (see section 8.2.2) to come up with a sequential algorithm based on extra mileage. The idea is growing one route by inserting one customer at a time; the customer and its insertion point are determined by minimizing extra mileage, provided that the vehicle capacity constraint is satisfied. The current route may be closed when there is no way to insert any other customer: then we start again with a new route. A potential weakness of such an idea is that, in order to saturate the current route, we could be forced to add a very distant customer. This may happen if there is a small residual capacity on the truck and the only customer with a small demand, fitting the residual capacity, is really far from the cluster of customers in the current route. This may be a good reason to prefer a parallel approach, in which we select which customer

to insert, on which route, and at which point. If we wish to use m vehicles, it is natural to start from m seed routes, each consisting of one customer. We can grow the routes using the extra-mileage criterion; this way, we can control the number of vehicles we use (assuming we can serve all of the customers with that number of vehicles). If there is a fixed cost associated with each vehicle, we may change the number of seed routes, trading off the number of vehicles against total distance traveled.

A common issue with parallel approaches of the second kind is the selection of seed customers. A sensible rule is that they should be distant from the deposit and distant from each other. The rationale behind the first requirement is that distant customers are an inconveniency, but they must be served anyway; it is better to include them in a route immediately, in order to avoid late insertions that may generate a large increase of the route length. Furthermore, it is natural to think that if customers are far away from each other, they are best served by separate routes. Hence, let us denote by σ_j, $j = 1, \ldots, m$, the seeds to initialize the desired m routes. The first seed is selected by maximizing its distance from the deposit. Then, after having selected the first k seeds, the next seed σ_{k+1} is found by solving

$$\max_i \min \{c_{i0},\ c_{i,\sigma_1},\ c_{i,\sigma_2}, \ldots,\ c_{i,\sigma_k}\}. \tag{8.3}$$

In plain terms, the new seed maximizes the minimum distance between itself and the deposit and the other seeds. The idea is illustrated in figure 8.9, under the assumption of Euclidean distances. Customer 1 is the first seed we would select, since it is the farthest one from the deposit. Customers 3, 4, and 5 do not make good seeds because they are close to the deposit. The next farthest node in the network is associated with customer 2. However, this node is close to customer 1; it is reasonable to assume that they will be served by the same route. The second seed we should select is customer 6, which is far from both node 0 and node 1. Of course, this is just a sensible heuristic, which only considers distance. One could also consider demand size with respect to vehicle capacity: It may be not advisable to leave customers with large demand to late insertions, as they may be hard to fit to residual capacity.

We see that there is room for a large variety of combinations of heuristics principles. Since we may grow routes using extra-mileage or savings criteria, a natural question is whether one of them performs best. As expected, there is no easy answer, and the result may depend on the problem instance. To get an intuitive feeling for the underlying issues, we may have a look at picture 8.10, which illustrates a rather artificial but instructive example (see [7]). We have four customers, located on an equilateral triangle; here we consider Euclidean distance, i.e., we assume that the distances we see in the drawing correspond to the real ones. Each customer demand is 1, and vehicle capacity is 2; so, each route should serve two customers. If we start with four separate one-customer routes, and we merge them in parallel using a savings criterion, we end up with

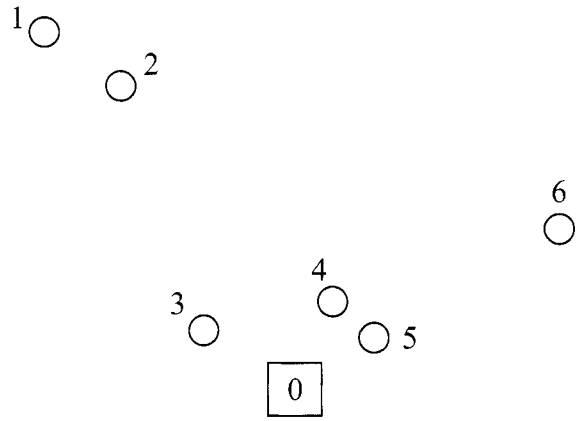
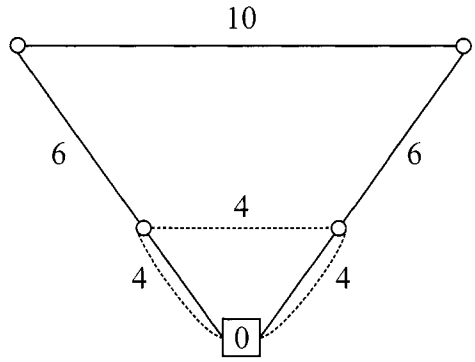
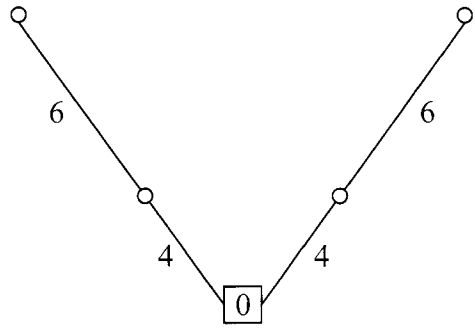


Fig. 8.9 Selecting seeds for parallel constructive heuristics.



Saving criterion: total length 42



Extra-mileage criterion: total length 40

Fig. 8.10 Alternative criteria to merge routes (case 1).

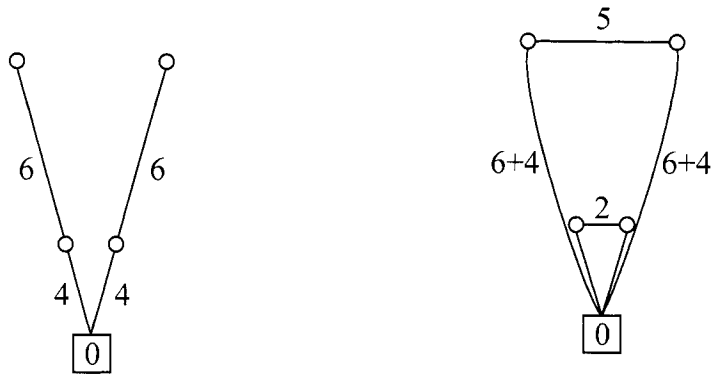


Fig. 8.11 Alternative criteria to merge routes (case 2).

the solution illustrated in the upper part of figure 8.10, with total length 42. Note that the largest saving ($10+10-10=10$) is obtained by joining the two farthest customers; this leads to a long an "circumferential" route. If we use extra mileage, e.g., starting from two seed routes associated to the farthest customers, we get the solution in the lower part of the figure, with total length 40; this happens because we can serve the two closer customers with no extra mileage. In this problem instance, the extra-mileage criterion performs better, but if we shrink the bottom angle and reduce horizontal distances in the triangle, as illustrated in figure 8.11, we get a different conclusion. In this second case, we still get total length 40 when using extra mileage, whereas the saving criterion yields a solution with total length 35. By the way, this second solution has lower total length, but it could be unsatisfactory in terms of workload balance: One driver gets a much easier task than the other one, an issue that we do not consider here, but may play a very important role. This example is clearly artificial, but it helps in building intuition about the qualitative properties of routes developed using the two criteria. We see from figures 8.10 and 8.11 that saving yields "circumferential" routes. This happens because the savings criterion may consider joining far customers *attractive*. In fact, Clarke and Wright's algorithm was included in an early software tool for VRP, developed by IBM in the 1970s. This package, called VPSX, was sometimes criticized by practitioners just because of the circumferential nature of proposed routes. On the contrary, when looking for small extra-mileage, it is natural to get more "radial" patterns.

The bottom line of the discussion so far is that it may be difficult to devise a *robust* method based on a single heuristic principle. Occasionally, any heuristic may yield a very poor solution. One way out of this difficulty is combining heuristic principles, possibly introducing one or more parameters which may

be adjusted as needed. One such idea is introducing a modified saving

$$\tilde{s}_{ij} = s_{ij} - \theta c_{ij},$$

where the parameter θ tends to penalize the inclusion of long arcs in the route, even if they yield large savings. This may prevent the creation of too circumferential routes. This is a simple example of parameterized criterion, and quite complex criteria have been proposed in the literature. Choosing the right value of one or more parameters is a tough task, but since constructive heuristics are quite fast, probably the best idea is simply using brute force and running the heuristic for several values of the parameter, keeping the best solution. Then, the solution can be further refined by local search.

8.3.2 Decomposition methods for VRP: cluster first, route second

VRP is a twofold problem with a *clustering* component (i.e., assigning a group of customers to each vehicle) and a routing component (i.e., finding the best route for each vehicle). Since the second problem dimension boils down to a set of TSP problems, which we may deal with rather effectively, the idea of decomposing the overall problem into two subproblems is quite natural. Various decomposition methods have been proposed and can be classified into two broad categories:

1. In **route-first, cluster-second** methods we first find one tour covering all of the customers, e.g., using some TSP solution method; then, we partition the resulting tour into routes compatible with vehicles' capacity.

2. In **cluster-first, route-second** methods we first assign customers to vehicles, subject to capacity constraints, and then we solve one TSP per vehicle.

Here we outline a couple of possible implementations of the second principle.

An early and intuitive sequential decomposition method, due to Gillett and Miller,[8] is called the **sweep** method. The approach has a strong geometric motivation, which is illustrated in figure 8.12. We draw a ray from the deposit, and we rotate the ray clockwise or counterclockwise; in doing so, we "sweep" customers in an order depending on their location. Whenever the ray passes over a customer, this is included in the current cluster and the process continues until we find a customer which cannot be fitted to the residual vehicle capacity. Then we form a cluster by grouping the "swept" customers; we route this subset of customers by solving the corresponding TSP, and we proceed by forming and routing the next cluster. We see that this approach is sequential in nature, which means that we have no control over the number

[8]See the original reference [11].

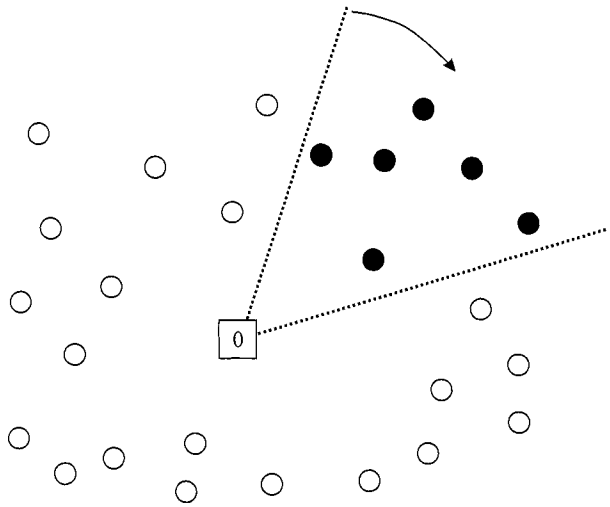


Fig. 8.12 Applying the *sweep* method to a Euclidean VRP.

of routes we will end up with; maybe by swapping and reassigning customers between clusters, we could reduce the number of routes. However, even if there exists a set of feasible routes involving a given number m vehicles, there is no guarantee that we will be able to find it. A further limitation of the approach is that it relies on a geometric argument; figure 8.12 should be really interpreted as a map of customer locations. If distances between nodes are strongly related to the Euclidean distances between points on the map, the result may be satisfactory. However, if the nature of terrain and roads is such that Euclidean distances are not closely related to actual distances, the quality of results could be low.

More recent and sophisticated decomposition-based methods have been proposed to overcome the limitations above, by exploiting partial mathematical modeling of VRP. Modeling the VRP by integer programming is possible, but not effective, unless nontrivial modeling and solution approaches are adopted. However, we may build a partial model, e.g., in order to assign customers to vehicles, leaving the routing task to a TSP solver. We illustrate here an idea due to Fisher and Jaikumar [10], which exploits a prototypical combinatorial optimization model known as *generalized assignment*. In this problem we are given a set of n jobs which must be carried out on a set of m machines (typically, $m < n$). Machines need not be identical: For each pair consisting of job $i = 1, \ldots, n$ and machine $k = 1, \ldots, m$, we have a processing time p_{ik} and a cost c_{ik}. We would like to carry out the whole set of jobs at minimal cost, but we might not be able to assign each job to the cheapest machine, because of capacity constraints: Each machine k is available for R_k time units. We can build an integer programming model by introducing a set

of binary decision variables y_{ik}, set to 1 if job i is assigned to machine k, 0 otherwise:

$$\min \quad \sum_{i=1}^{n}\sum_{k=1}^{m} c_{ik}y_{ik} \tag{8.4}$$

$$\text{s.t.} \quad \sum_{k=1}^{m} y_{ik} = 1, \qquad i = 1,\ldots,n, \tag{8.5}$$

$$\sum_{i=1}^{n} p_{ik}y_{ik} \leq R_k, \qquad k = 1,\ldots,m, \tag{8.6}$$

$$y_{ik} \in \{0,1\}.$$

The objective function (8.4) is total cost; constraint (8.5) makes sure that each job is assigned to exactly one machine, and (8.6) is the capacity constraint for each machine. This literal description of generalized assignment leaves room to many interpretations. In the VRP case, we may interpret jobs as customers to be served and machines as vehicles. So far, we have assumed that vehicles are identical, and we stick to this case denoting the vehicle capacity by R; however, we see an immediate advantage of this approach, which helps in getting rid of many limitations. We may also include restrictions on the type of vehicle that can be used to serve a customer (e.g., because large trucks cannot be used in old town centers). Denoting the demand from customer i by d_i, *in principle* we can write the following model:

$$\min \quad \sum_{k=1}^{m} f(\mathbf{y}_k)$$

$$\text{s.t.} \quad \sum_{k=1}^{m} y_{ik} = 1, \qquad i = 1,\ldots,n,$$

$$\sum_{i=1}^{n} d_i y_{ik} \leq R, \qquad k = 1,\ldots,m,$$

$$y_{ik} \in \{0,1\}.$$

This is just a generalized assignment problem with a weird objective function. Here vector $\mathbf{y}_k$ consists of all of the decision variables y_{ik} associated with vehicle k. From a conceptual point of view, we may imagine a function $f(\mathbf{y}_k)$ which yields the optimal tour length obtained by solving to optimality a TSP restricted to the customers assigned to vehicle k. If we were really able to write such a function analytically, the model above would be a working model for VRP. Of course, we are not that lucky, but we can try to approximate function $f(\cdot)$ in a way which is suitable to solution by linear integer programming:

$$f(\mathbf{y}_k) \approx \sum_{i=1}^{n} g_{ik}y_{ik}.$$

The parameter g_{ik} should be an approximation of the cost of assigning customer i to vehicle k. Clearly, such a linear function cannot really capture the interactions among various assignments, which influence each other when solving the TSP. However, we can try to find a suitable value if we assume that a set of m customer seeds is given. Such seeds play the same role as in constructive parallel heuristics of type 2 (i.e., those based on growing a given number of routes), and they could be selected by the logic behind expression (8.3) on page 418; we select m seeds by finding a subset of m customers which are far from the deposit and far from each other. Given the seeds, which are associated with the m routes, we may estimate the cost of inserting customer i in any route by computing the extra mileage with respect to the deposit 0 and the seed σ_k $(k = 1, \ldots, m)$ of that route:

$$g_{ik} = c_{0i} + c_{i,\sigma_k} - c_{0,\sigma_k}.$$

Now that we have a linear approximation of the TSP cost, we may solve the generalized assignment problem by branch-and-bound, or by ad hoc methods if problem size precludes using a commercial integer programming package; then we solve one TSP for each vehicle. A noteworthy feature of the approach is that if there exists a feasible solution using m vehicles, we will find one; constructive heuristics do not offer such a guarantee. Another important remark is that the generalized assignment formulation, as we have already pointed out, can be extended to cope with heterogeneous vehicles and to model some additional constraints on the vehicles that can be used to serve a customer (some goods need freezer trucks, or separate sections because of mutual incompatibility; for instance, think of food and chemicals).

A later extension of the generalized assignment approach was proposed by Bramel and Simchi-Levi [6], in order to avoid the *a priori* selection of seeds. In order to integrate seed selection with customer clustering, they proposed a *concentrator location* formulation. Let us introduce the following decision variables:

$$z_j = \begin{cases} 1 & \text{if customer } j \text{ is selected as a seed,} \\ 0 & \text{otherwise;} \end{cases}$$

$$y_{ij} = \begin{cases} 1 & \text{if customer } i \text{ is assinged to a route, whose seed is customer } j, \\ 0 & \text{otherwise.} \end{cases}$$

We also need the cost coefficients:

$$g_{ij} = c_{0i} + c_{ij} - c_{0j}, \qquad v_j = 2c_{0j}.$$

The first cost is a familiar extra mileage, whereas the second one is associated to the selection of customer j as a seed; the cost is approximated by the distance of a round trip from deposit to customer j and back. The resulting optimization model is

$$\min \quad \sum_{i=1}^{n} \sum_{j=1}^{n} g_{ij} y_{ij} + \sum_{j=1}^{n} v_j z_j \tag{8.7}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} z_j = m, \tag{8.8}$$

$$\sum_{i=1}^{n} d_i y_{ij} \leq R, \qquad j = 1, \ldots, n, \tag{8.9}$$

$$\sum_{j=1}^{n} y_{ij} = 1, \qquad i = 1, \ldots, n, \tag{8.10}$$

$$y_{ij} \leq z_j, \qquad i, j = 1, \ldots, n, \tag{8.11}$$

$$y_{ij},\ z_j \in \{0, 1\}.$$

The objective function (8.7) is total cost; constraint (8.8) enforces the selection of a given number m of seeds; constraints (8.9) and (8.10) are essentially the same as the generalized assignment formulation; finally, constraint (8.11) states that if customer j is not selected as a seed, we cannot assign any customer i to it (i.e., to the route associated with seed j).

8.4 ADDITIONAL FEATURES OF REAL-LIFE VRP

In the previous section we have considered solution approaches for the basic VRP. Typical routing problems have several additional features that make their solution a bit tougher, even though the heuristic principles we have just outlined can be adapted. In the following list, we illustrate a few of these complications.

- In the basic VRP, given a set of customers along with their demand, we have to build a brand new set of routes; if the demand pattern changes, the set of routes may change as well. From an organizational point of view, this may be an inconveniency. Hence, at a more tactical level, one may try to come up with a set of fixed routes which are traveled several times. Such **fixed routing** problems may also be formulated and solved in the case of uncertain demand.

- We have taken the number of vehicles as given (or irrelevant). In **fleet planning** problems, the aim is sizing a fleet of vehicles. This type of problem is also relevant in point-to-point transportation.

- We have considered a static, deterministic, and single-period problem. Depending on the practical context, uncertainty may affect travel times and/or demand. One simple approach to tackle the first source of uncertainty would be introducing slack time by judiciously overestimating travel times. Demand uncertainty can be tackled by similar means. However, the real issue is arguably the real-time management for such a problem; this calls for efficient real-time data collection and an effective organization in order to adapt routes and delivery on-the-fly. Dealing

with uncertainty may result in a tough, dynamic and stochastic problem. Even if we rule out uncertainty, we may have to cope with a multiperiod problem, whereby we have to develop routes for a few consecutive periods; as we have already noted in the book, multiperiod problems need not be dynamic in the sense of adapting to uncertain events.

- We have considered a symmetric distance matrix, whereas sometimes the underlying TSP structure is asymmetric. Apart from adjustments in solution algorithms, a difficult issue is filling the matrix with reliable data. In the past, one possibility was computing plain Euclidean distances and then inflating them by coefficients modeling the difficulty of the terrain. Given technological advancement, geographic information systems are now typically exploited to this aim. This can also be done at the single customer address level, e.g., analyzing the ZIP code, by a process called *geocoding*.

- The maximum tour length in terms of time and space may be a constraining factor, and not only capacity. Capacity is actually multidimensional, involving both weight and volume, potentially for separate parts of the truck, such as refrigerated and nonrefrigerated. Exploiting the available volume is an optimization problem in itself, for which software packages have been developed. We should emphasize however, that such an optimization is desirable for point-to-point service, but it may get into the way of unloading stuff in multiple delivery problems such as VRP: Having fully loaded the truck is of little use, if the parcel of the first customer in the route lies at the unreachable bottom of the truckload.

- The objective function of a real VRP may involve multiple costs, and not only distance. Some desirable features of a route may hardly be expressed in monetary terms. If not all of the customers can be accommodated for delivery today, we must decide which ones will be served tomorrow. Moreover, overtime driver cost may by quantified, but workload balance issues in human resource management may be hardly turned into a cost. The flexibility of local search in dealing with complex objectives may be used to advantage.

- We may have multiple deposits and heterogeneous vehicles (possibly with separate sections). Cluster-first, route-second methods may be adapted in some cases.

- In **inventory routing** problems, vehicle routing is coupled with inventory management. In basic VRP, we consider customer demand as given; but if inventories are taken into account, a brand new dimension is open, offering both degrees of freedom and additional complexity. If inventory is available at customer nodes, we may better manage vehicle

capacity, by delivering flexible quantities depending on current inventory state. We may also better cope with demand uncertainty. The price we pay for such opportunities is the difficulty of the integrated problem.

- Last but not least, delivery may be subject to **time windows**, linked to traffic conditions, customer's requirement, or to the availability of an unloading bay. This places additional constraints on solution methods.

Given all of these complexity factors, the staggering amount of scientific literature on VRP is no surprise. One way to cope with real-life VRPs is to extend and adapt the heuristic principles we have briefly illustrated. Local search approaches are certainly an interesting way to tackle generalized versions of the basic VRP. However, sometimes constraints are so tight that even finding a *feasible* solution is difficult. In this case, sophisticated mathematical modeling and solution approaches can offer some advantage. Since this level of sophistication requires advanced optimization concepts, in the next section we just offer some clues on how constructive heuristics can be extended to cope with time windows. Whatever solution approach we take, we should bear in mind that real-life VRPs may be subject to significant uncertainty and ill-defined objectives linked to human factors; hence, solution approaches must be cast within a well-designed decision support system.

8.4.1 Constructive methods for the VRP with time windows

In the VRP with time windows,[9] each customer $i = 1, \ldots, n$ is associated with an interval $[e_i, l_i]$, whose endpoints are the *earliest time* and the *latest time* for the start of service (in our case, unloading the vehicle). In practice, multiple time windows may be associated with a single customer. Let s_i be the duration of service and t_{ij} be the time to travel from customer i to customer j. If the vehicle arrives early with respect to the time window, then it must wait. Hence, if b_i denotes the start of service for customer i, and the vehicle visits customer j after customer i a j, we have

$$b_j = \max\{e_j,\ b_i + s_i + t_{ij}\}.$$

In the basic VRP we were deliberately ambiguous in using a "cost" c_{ij} which could be related to space, or time, or a mixture of both. For the VRP with time windows, we must take both time and space into account; we will denote the distance between customers i and j by q_{ij}.

Based on our knowledge of constructive heuristics, one of the first ideas that may come to our mind is to build routes sequentially, using an extension of the nearest-neighbor TSP approach. In this case, "nearest" mixes both

[9] This section is based on [17], to which we refer for a full treatment.

space and time considerations. We recall that in this algorithm customers are always appended at the end of the growing route, which may be a limitation. Let i be the last visited customer; we must define a hybrid measure c_{ij} of "closeness" between i and j. One possibility is the following:

$$c_{ij} = \lambda_1 q_{ij} + \lambda_2 T_{ij} + \lambda_3 v_{ij},$$

where:

- the weights λ_i are non-negative and sum up to one (actually, we must just give two weights);
- q_{ij} is the distance between customers i and j;
- the quantity
$$T_{ij} = b_j - (b_i + s_i)$$
takes into account the time difference between the end of service at i and the beginning of service at j (if j follows i on the route);
- the quantity
$$v_{ij} = l_j - (b_i + s_i + t_{ij})$$
measures the time slack we still have for service at customer j, i.e., how much time remains to the end of its time window (the smaller v_{ij}, the more urgent it is serving j after i; hence, this factor works in the same way as the previous two in making service of j after i desirable).

This generalized metric is used as a simple priority rule to append customers to the current route. When no more customers can be appended, because the vehicle is full or no time window is compatible, we close the current route and start a new one. Clearly, we have no direct control on the number of routes we build, and the algorithm looks quite greedy. In any metric depending on weights, parameter fine-tuning is an issue. However, in this case we just have to select a combination of two parameters ranging between 0 and 1; since a greedy procedure is very fast, we may simply carry out a grid search, trying several weight combinations and plucking the best solution found.

Given the already familiar limitations of nearest-neighbor, we may consider adapting insertion-based heuristics. Consider a partial route $0, i_1, i_2, \ldots, i_m, 0$, starting and terminating at deposit 0. We may work at two levels:

1. For each unrouted customer u, we compute the best insertion point in the partial route (provided vehicle capacity is not exceeded), according to some metric.
2. We select the best customer to insert, applying some metric, which need not be the same as in the previous point.

We should note that inserting a customer my imply a time shift for all of the following customers along the route.

To evaluate the opportunity of inserting customer u between i and j, we can adapt the following metric (which should be minimized to find the best insertion point):

$$c_1(i,u,j) = \alpha(q_{iu} + q_{uj} - \mu q_{ij}) + (1-\alpha)(b_{j_u} - b_j),$$

where α and μ are parameters to be chosen. The parameter α must be selected in the range $[0,1]$ and controls the relative weight we assign to space vs. time considerations. In fact, the first term in the sum is linked to extra mileage; indeed, it is be the familiar extra mileage if $\mu = 1$. The additional parameter μ allows for extra fine-tuning of the heuristics; actually, such coefficients are common in variations of insertion heuristics for TSP. The second term includes the difference between the new start time of service at customer j, if we insert u, denoted by b_{j_u}, and the current start time b_j. This term tries to capture the time shift effect due to insertion, provided that the insertion is feasible with respect to time windows.

To select the customer u to insert, given the best insertion point above, we may consider the following metric (to be maximized):

$$c_2(i,u,j) = \lambda q_{0u} - c_1(i,u,j),$$

where λ is a parameter to be chosen. If we select parameters $\mu = \alpha = 1$ and $\lambda = 2$, this metric measures the saving in terms of traveled distance if we serve customer u between i and j rather than serving u directly from the deposit. An alternative choice is

$$c_2(i,u,j) = \beta R_d(u) + (1-\beta)R_t(u),$$

where $R_d(u)$ and $R_t(u)$ are total distance and total time of the current route if we insert u. respectively; the parameter β must be selected in the range $[0,1]$ and has essential the same meaning as the parameter α above. This metric, which should be minimized, tries to capture more fully the effect of the insertion.

These very simple rules, and related variants, have a definite advantage in terms of CPU time and conceptual simplicity. They might not be very effective in tightly constrained problems. If we do not want to resort to complex mathematics, we could also consider local search algorithms, but there are additional complications when we try to apply something like a 2−opt neighborhood structure to a VRP with time windows. When we delete and cross a pair of arcs, we invert the direction of some part of the route; this is irrelevant in terms of distance, provided that the problem is symmetric, but it is definitely relevant in terms of time windows. Nevertheless, many clever approaches have been proposed over the years, which are described in the references listed at the end of the chapter.

8.5 FINAL REMARKS

Sub Bab ini dst tidak perlu diterjemahkan

In this chapter we have considered basic VRP as a straightforward extension of classical TSP. The interest of these network routing problems has spurred a significant amount of work, which is documented by a vast literature where a wide array of methods is presented. It is hard to tell if there is one best approach. On the one hand, very sophisticated mathematical approaches have been developed, and despite technical intricacy, their potential for economic impact must be carefully considered. On the other hand, the variety of constraints and complicating features has led software developers to privilege simpler and possibly more adaptable approaches. What we tried to accomplish in this chapter is just to get the reader acquainted with the conceptual foundations of these approaches.

We should also raise a couple of general points, whose practical importance cannot be overemphasized. The first one is that we have considered VRP as an off-line scheduling problem. In practice, disruptions and uncertainty are a way of life; hence, it is essential to develop suitable user interfaces to manage such situations. Such decision support systems must also rely on proper data collection from the field; new satellite-based technologies are being exploited for this task. Last, but not least, we have just considered cost minimization. Environmental issues should remind us that proper transportation management and organization has a significant impact, which goes beyond the bottom line of a single firm.

8.6 FOR FURTHER READING

- We did not consider mathematically sophisticated approaches to solve the TSP, which include branch and bound methods based on Lagrangian relaxation (i.e., the relaxation of complicating constraints by Lagrangian multipliers; see section B.4) and branch and cut methods (i.e., branch and bound methods in which constraints, i.e., cuts, are added to strengthen the lower bound we get from the continuous relaxation; see section B.6.1). A not-so-recent, but still relevant reference book is [14].
- An overview of local search methods for combinatorial optimization can be found in [1]; a recent survey is [5]. A specific reference on tabu search is [12]; see [15] for an application of GRASP to the TSP.
- An excellent survey on early approaches to VRP can be found in [7], from which we have taken part of section 8.3 (in particular, the examples and the discussion associated with figures 8.10 and 8.11).
- A more recent survey, which also includes approaches based on mathematical programming, can be found in [9]. See also [19].
- For VRP under uncertain demand see, e.g., [4].

- For a complete description of simple heuristics for VRP with time windows, see [17]; we have just hinted at a few basic concepts in section 8.4.1.
- Some commercially available software packages for VRP are just based on principles we have outlined, but the complexity of real-life VRP can only be appreciated by having a look at the data requirements to define a problem. Many complicating constraints must be addressed, and the data we have taken for granted, such as the distance matrix, may require a link to a geographic information system. For instance, you can have a look at the web site `http://www.bestroutes.com/`
- Many commercial tools of VRP are "closed" products. There is also the possibility of using software components to tailor a specific application. This is the approach taken by the ILOG Dispatcher library (see `http://www.ilog.com/`)
- For routing applications in transportation by aircraft or railway, see [2] and [13], respectively.

REFERENCES

1. E. Aarts and J.K. Lenstra. *Local Search in Combinatorial Optimization.* Wiley, NewYork, NY, 1997.
2. A.P. Armacost, C. Barnhart, K.A. Ware, and A.M. Wilson. UPS Optimizes Its Air Network. *Interfaces*, 34:15–25, 2004.
3. M.O. Ball, T.L. Magnanti, C.L. Monma, and G.L. Nemhauser, editors. *Network Routing (Handbooks in Operations Research and Management Science, Vol. 8).* Elsevier Science, Amsterdam, 1995.
4. D. Bertsimas and D. Simchi-Levi. The New Generation of Vehicle Routing Research: Robust Algorithms Addressing Uncertainty. *Operations Research*, 44:286–304, 1996.
5. C. Blum and A. Roli. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Computing Surveys*, 35:268–308, 2003.
6. J. Bramel and D. Simchi-Levi. A Location-Based Heuristic for General Routing Problems. Working Paper, Department of Industrial Engineering and Operations Research, Columbia University, 1992.
7. N. Christofides, A. Mingozzi, and P. Toth. The Vehicle Routing Problem. In N. Christofides, A. Mingozzi, P. Toth, and C. Sandi, editors, *Combinatorial Optimization.* Wiley, Chichester, 1979.

8. G. Clarke and J.W. Wright. Scheduling of Vehicles from a Central Depot to a Number of Delivery Points. *Operations Research*, 12:568–581, 1964.

9. M.L. Fisher. Vehicle Routing Problem. In M.O. Ball, T.L. Magnanti, C.L. Monma, and G.L. Nemhauser, editors, *Network Routing (Handbooks in Operations Research and Management Science, Vol. 8)*. Elsevier Science, Amsterdam, 1995.

10. M.L. Fisher and R. Jaikumar. A Generalized Assignment Heuristic for Vehicle Routing. *Networks*, 11:109–124, 1981.

11. B.E. Gillett and L.R. Miller. A Heuristic Algorithm for the Vehicle-Dispatch Problem. *Operations Research*, 22:340–349, 1974.

12. F.W. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Dordrecht, 1997.

13. P. Ireland, R. Case, J. Fallis, Carl Van Dyke, J. Kuehn, and M. Meketon. The Canadian Pacific Railway Transforms Operations by Using Models to Develop Its Operating Plans. *Interfaces*, 34:5–14, 2004.

14. E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley, NewYork, NY, 1985.

15. Y. Marinakis, A. Migdalas, and P.M. Pardalos. Expanding Neighborhood GRASP for the Traveling Salesman Problem. *Computational Optimization and Applications*, 32:231–257, 2005.

16. W.B. Powell. Dynamic Models of Transportation Operations. In A.G. de Kok and S.C. Graves, editors, *Supply Chain Management: Design, Coordination, and Operation*. Elsevier, Amsterdam, 2003.

17. M.M. Solomon. Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints. *Operations Research*, 35:254–265, 1987.

18. R.E. Tarjan. *Data Structures and Network Algorithms*. SIAM, Philadelphia, 1983.

19. P. Toth and D. Vigo. *The Vehicle Routing Problem*. SIAM, Philadelphia, 2001.