Chapter 5

Classical Heuristics for the Capacitated VRP

Gilbert Laporte
Frédéric Semet

5.1 Introduction

Several families of heuristics have been proposed for the VRP. These can be broadly classified into two main classes: *classical heuristics*, developed mostly between 1960 and 1990, and *metaheuristics*, whose growth has occurred in the last decade. Most standard construction and improvement procedures in use today belong to the first class. These methods perform a relatively limited exploration of the search space and typically produce good quality solutions within modest computing times. Moreover, most of them can be easily extended to account for the diversity of constraints encountered in real-life contexts. Therefore, they are still widely used in commercial packages. In metaheuristics, the emphasis is on performing a deep exploration of the most promising regions of the solution space. These methods typically combine sophisticated neighborhood search rules, memory structures, and recombinations of solutions. The quality of solutions produced by these methods is much higher than that obtained by classical heuristics, but the price to pay is increased computing time. Moreover, the procedures usually are context dependent and require finely tuned parameters, which may make their extension to other situations difficult. In a sense, metaheuristics are no more than sophisticated improvement procedures, and they can simply be viewed as natural enhancements of classical heuristics. However, because they make use of several new concepts not present in classical methods, they are covered separately, in Chapter 6.

Classical VRP heuristics can be broadly classified into three categories. *Constructive heuristics* gradually build a feasible solution while keeping an eye on solution cost, but they do not contain an improvement phase per se. In *two-phase heuristics*, the problem is decomposed into its two natural components, clustering of vertices into feasible routes and actual route construction, with possible feedback loops between the two stages. Two-phase heuristics are divided into two classes: *cluster-first, route-second* methods and *route-first, cluster-second* methods. In the first case, vertices are first organized into feasible clusters, and a vehicle route is constructed for each of them. In the second case, a tour is first built on all vertices and is then segmented into feasible vehicle routes. Finally, *improvement methods* attempt to upgrade any feasible solution by performing a sequence of edge or vertex exchanges within or between vehicle routes. These three classes of methods are covered in the next three sections, respectively. The distinction between constructive and improvements methods, however, is often blurred since most constructive algorithms incorporate improvements steps (typically 3-opt (Lin [26])) at various stages. Since the number of available methods and variants is very large, we concentrate on the truly classical heuristics and enhancements, leaving some variants aside. For additional readings on classical heuristics for the VRP, see Christofides, Mingozzi, and Toth [10], Bodin et al. [6], Christofides [9], Golden and Assad [21], and Fisher [16].

Most of the heuristics developed for the VRP apply directly to capacity constrained problems (CVRPs) and normally can be extended to the case where an upper bound is also imposed on the length of any vehicle route (DCVRPs), even if this is not always explicitly mentioned in the algorithm description. Most heuristics work with an unspecified number K of vehicles, but there are some exceptions to this rule. This is clarified for each case. The distance matrix used in the various heuristics described in this chapter can be symmetric or not, but very little computational experience has been reported for the asymmetric case. One important exception is Vigo [44]. A few methods have been designed for planar problems.

5.2 Constructive Methods

Two main techniques are used for constructing VRP solutions: merging existing routes using a *savings* criterion, and gradually assigning vertices to vehicle routes using an insertion cost.

5.2.1 Clarke and Wright Savings Algorithm

The Clarke and Wright [11] algorithm is perhaps the most widely known heuristic for the VRP. It is based on the notion of savings. When two routes $(0, \dots, i, 0)$ and $(0, j, \dots, 0)$ can feasibly be merged into a single route $(0, \dots, i, j, \dots, 0)$, a distance saving $s_{ij} = c_{i0} + c_{0j} - c_{ij}$ is generated. This algorithm naturally applies to problems for which the number of vehicles is a decision variable, and it works equally well for directed or undirected problems, but Vigo [44] reports that the behavior of the method worsens considerably in the directed case, although the number of potential route merges is then halved. A parallel and a sequential version of the algorithm are available. The algorithm works as follows.

Step 1 (savings computation). Compute the savings $s_{ij} = c_{i0} + c_{0j} - c_{ij}$ for $i, j = 1, \dots, n$ and $i \neq j$. Create n vehicle routes $(0, i, 0)$ for $i = 1, \dots, n$. Order the savings in a nonincreasing fashion.

Parallel version

Step 2 (best feasible merge). Starting from the top of the savings list, execute the following. Given a saving s_{ij}, determine whether there exist two routes, one containing arc or edge

$(0, j)$ and the other containing arc or edge $(i, 0)$, that can feasibly be merged. If so, combine these two routes by deleting $(0, j)$ and $(i, 0)$ and introducing (i, j).

Sequential version
Step 2 (route extension). Consider in turn each route $(0, i, \ldots, j, 0)$. Determine the first saving s_{ki} or $s_{j\ell}$ that can feasibly be used to merge the current route with another route containing arc or edge $(k, 0)$ or containing arc or edge $(0, \ell)$. Implement the merge and repeat this operation to the current route. If no feasible merge exists, consider the next route and reapply the same operations. Stop when no route merge is feasible.

There is great variability in the numerical results reported for the savings heuristics, and authors often do not mention whether the parallel or the sequential version is considered. In Table 5.1, we compare these two versions on the 14 symmetric instances of Christofides, Mingozzi, and Toth [10], using real distances. These results indicate that the parallel version of the savings method clearly dominates the sequential one. Computing times on a Sun Ultrasparc 10 workstation (42 Mflops) are typically less than 0.2 second.

5.2.2 Enhancements of the Clarke and Wright Algorithm

One drawback of the original Clarke and Wright algorithm is that it tends to produce good routes at the beginning but less interesting routes toward the end, including some circumferential routes. To remedy this, Gaskell [19] and Yellow [48] proposed generalized savings of the form $s_{ij} = c_{i0} + c_{0j} - \lambda c_{ij}$, where λ is a route shape parameter. The larger the λ,

Table 5.1. *Computational comparison of two implementations of the Clarke and Wright algorithm.*

Problem	Sequential	Parallel	Best known solution value
E051-05e	625.56	584.64	524.61[1]
E076-10e	1005.25	900.26	835.26[1]
E101-08e	982.48	886.83	826.14[1]
E101-10c	939.99	833.51	819.56[1]
E121-07c	1291.33	1071.07	1042.11[1]
E151-12c	1299.39	1133.43	1028.42[1]
E200-17c	1708.00	1395.74	1291.45[1]
D051-06c	670.01	618.40	555.43[1]
D076-11c	989.42	975.46	909.68[1]
D101-09c	1054.70	973.94	865.94[1]
D101-11c	952.53	875.75	866.37[1]
D121-11c	1646.60	1596.72	1541.14[2]
D151-14c	1383.87	1287.64	1162.55[2]
D200-18c	1671.29	1538.66	1395.85[1]

[1]Taillard [41].

[2]Rochat and Taillard [37].

the more emphasis is put on the distance between the vertices to be connected. Golden, Magnanti, and Nguyen [22] report that using $\lambda = 0.4$ or 1.0 yields good solutions, taking into account the number of routes and the total length of the solution.

The Clarke and Wright algorithm can also be time consuming since all savings must be computed, stored, and sorted. Various enhancements have been proposed by a number of authors to speed up computations and to reduce memory requirements. Most of this work took place in the 1970s and at the early 1980s, when researchers worked with computers much less powerful than current workstations. Instances involving 200 to 600 vertices could take from 25 to 300 seconds on an IBM 4341 computer, using a straightforward implementation of the parallel savings method (Nelson et al. [30]). Now, a 200-vertex instance can be solved in 0.3 second on a Sun Ultrasparc 10 workstation with the same kind of implementation. Therefore, these enhancements are useful only for very large instances (more than 1000 vertices). When implementing the savings heuristic, two main issues must be addressed: determination of the maximum saving value and storage requirements.

Computing the maximum saving value is the most time consuming part of the algorithm. Three approaches can be considered. The first uses a full sort (e.g., quicksort) implemented in a straightforward manner. The second approach is an iterative limited sort that can be performed by means of a heap structure (Golden, Magnanti, and Nguyen [22]). A heap is a binary tree where the savings are stored in a such way that the value of the father node is always greater than or equal to that of the son nodes. When two routes are merged, the heap is rebuilt efficiently to eliminate the saving associated with the selected link and all savings corresponding to an interior vertex of a route. The third approach is an iterative computation of the maximum saving value (Paessens [33]). Assuming that distances are positive and that the triangle inequality holds, Paessens shows that $s_{ij} > \overline{s}$ whenever $c_{0i} > 0.5\,\overline{s}$ and $c_{0j} > 0.5\,\overline{s}$, where $\overline{s}$ is the current maximum saving value. This necessary condition is then used to efficiently identify the larger saving values. The three approaches have been implemented by Paessens. Numerical results are reported on four instances with three different vehicle capacities. The iterative determination of the maximum saving value tends to be the most efficient on the average. However, important variations in the computing times can occur, depending on the vehicle capacity, which is not the case when a complete sorting approach is used. To increase the savings method performance in terms of computing time and memory requirements, some authors proposed considering only a subset of all possible savings. Golden, Magnanti, and Nguyen [22] suggested superimposing a grid over the network. The grid is divided into rectangles, and all edges between vertices belonging to nonadjacent rectangles are eliminated with the exception of the edges linking vertices to the depot. Savings are then computed on this subnetwork. Paessens [33] proposed disregarding edges with $c_{ij} > \alpha \max_{k \in \{1,\ldots,n\}} c_{0k}$ for some constant α.

Nelson et al. [30] investigated more complex data structures based on heaps to limit storage requirements and thus obtain more efficient updating operations. They presented four different ways to use adjacency information to eliminate all edges associated with an interior vertex. For noncomplete graphs, the most efficient implementation requires $7m+3n$ storage locations, whereas the storage requirement is only $3m + 3n$ for complete graphs, where m is the number of edges. This is achieved by using hashing functions to identify the vertices associated with a given edge and to determine the location of all edges associated with an interior vertex. The last implementation proposed uses several smaller heaps instead of one large heap. At a given step, the heap contains only savings associated with noninterior vertices which exceed a threshold value. The heap is then processed until it is empty. A new threshold value is finally selected and a new heap is constructed. This is repeated until all edges have been considered. Numerical results show that the last implementation is the best. Instances containing 1000 vertices typically can be solved in 180 seconds on an IBM 4341 computer.

5.2.3 Matching-Based Savings Algorithms

Desrochers and Verhoog [12] and Altinkemer and Gavish [2] described an interesting modification to the standard savings algorithm. The two algorithms are rather similar. At each iteration the saving s_{pq} obtained by merging routes p and q is computed as $s_{pq} = t(S_p) + t(S_q) - t(S_p \cup S_q)$, where S_k is the vertex set of route k and $t(S_k)$ is the length of an optimal Traveling Salesman Problem (TSP) solution on S_k. A max-weight matching problem over the sets S_k is solved using the s_{pq} values as matching weights, and the routes corresponding to optimal matchings are merged, provided feasibility is maintained. Several variants of this basic algorithm are possible, one of which approximates the $t(S_k)$ values instead of computing them exactly.

Another matching based approach is described by Wark and Holt [45]. These authors used a matching algorithm to successively merge clusters, defined as ordered sets of vertices, at their endpoints. Matching weights may be defined as ordinary savings, or these may be modified to favor mergers of clusters whose total weight is far below vehicle capacity or whose length is far below the allowed distance limit on a vehicle route. Starting with n back and forth vehicle routes, the algorithm successively merges clusters. After a merge is performed, only a few lines or columns of the savings matrix need be updated. If all clusters are matched with themselves, then some of them are split with a given probability. The process thus grows a tree of sets of clusters from which a best solution can be selected.

We compare these three matching-based algorithms in Table 5.2 on the 14 instances of Christofides, Mingozzi, and Toth [10], and we also provide a comparison with the parallel version of the Clarke and Wright heuristic. These results must be interpreted with care. First, the rounding rules used for the c_{ij} coefficients are not the same for all heuristics used in the comparison. This rule is not reported for the Desrochers and Verhoog algorithm. Altinkemer and Gavish round distances to the nearest integer. The Wark and Holt and best known solutions are obtained with real distances. Also, the Altinkemer and Gavish results are the best of approximately 40 runs, using several parameters and algorithmic rules. The Wark and Holt results are the best of five runs. Computation times vary between 0.03 and 0.33 second on a Sun Ultrasparc 10 for the Clarke and Wright algorithm, and between 21.40 and 3087.73 seconds on an IBM 3083 for each round of the Altinkemer and Gavish algorithm. Desrochers and Verhoog report average computing times between 38 and 3200 seconds on an unspecified machine. Each run of the Wark and Holt algorithm requires on average between 4 and 107 minutes on a Sun 4/630MP. Despite the above remarks, it can safely be said that the use of a matching-based algorithm yields better results than the standard Clarke and Wright method, but at the expense of much higher computation time. The Wark and Holt heuristic is clearly the best of the three matching-based methods in terms of solution quality. Bold numbers in the table indicate that the algorithm has identified a best known solution.

Table 5.2. *Computational comparison of four savings-based heuristics.*

Problem	Clarke and Wright[1]	Desrochers and Verhoog[2]	Altinkemer and Gavish[3]	Wark and Holt[4]	Best known solution value
E051-05e	578.64	586	556	**524.6**	524.61[5]
E076-10e	900.26	885	855	835.8	835.26[5]
E101-08e	886.83	889	860	830.7	826.14[5]
E101-10c	833.51	828	834	**819.6**	819.56[5]
E121-07c	1071.07	1058	1047	1043.4	1042.11[5]
E151-12c	1133.43	1133	1085	1038.5	1028.42[5]
E200-17c	1395.74	1424	1351	1321.3	1291.45[5]
D051-06c	618.40	593	577	**555.4**	555.43[5]
D076-11c	975.46	963	939	911.8	909.68[5]
D101-09c	973.94	914	913	878.0	865.94[5]
D101-11c	875.75	882	874	**866.4**	866.37[5]
D121-11c	1596.72	1562	1551	1548.3	1541.14[6]
D151-14c	1287.64	1292	1210	1176.5	1162.55[6]
D200-18c	1538.66	1559	1464	1418.3	1395.85[5]

[1] Parallel savings heuristic implemented by Laporte and Semet (Table 5.1).

[2] Desrochers and Verhoog [12].

[3] Altinkemer and Gavish [2]. Best of approximately 40 versions.

[4] Wark and Holt [45]. Best of five runs.

[5] Taillard [41].

[6] Rochat and Taillard [37].

5.2.4 Sequential Insertion Heuristics

We now describe two representative algorithms based on sequential insertions. Both apply to problems with an unspecified number of vehicles. The first, by Mole and Jameson [29], expands one route at a time. The second, proposed by Christofides, Mingozzi, and Toth [10], applies in turn sequential and parallel route construction procedures. Both methods contain a 3-opt improvement phase.

5.2.4.1 Mole and Jameson Sequential Insertion Heuristic

The Mole and Jameson algorithm uses two parameters λ and μ to expand a route under construction:

$$\alpha(i, k, j) = c_{ik} + c_{kj} - \lambda c_{ij},$$
$$\beta(i, k, j) = \mu c_{0k} - \alpha(i, k, j).$$

The algorithm can be described as follows.

Step 1 (emerging route initialization). If all vertices belong to a route, stop. Otherwise, construct an emerging route $(0, k, 0)$, where k is any unrouted vertex.

Step 2 (next vertex). Compute for each unrouted vertex k the feasible insertion cost $\alpha^*(i_k, k, j_k) = \min\{\alpha(r, k, s)\}$ for all adjacent vertices r and s of the emerging route,

where i_k and j_k are the two vertices yielding α^*. If no insertion is feasible, go to Step 1. Otherwise, the best vertex k^* to insert into the emerging route is the vertex yielding $\beta^*(i_{k^*}, k^*, j_{k^*}) = \max\{(\beta(i_k, k, j_k)\}$ over all unrouted vertices k that can feasibly be inserted. Insert k^* between i_{k^*} and j_{k^*}.
Step 3 (route optimization). Optimize the current route by means of a 3-opt procedure (Lin [26]), and go to Step 2.

Several standard insertion rules are governed by the two parameters λ and μ. For example, if $\lambda = 1$ and $\mu = 0$, the algorithm will insert the vertex yielding the minimum extra distance. If $\lambda = \mu = 0$, the vertex to be inserted will correspond to the smallest sum of distances between two neighbors. If $\mu = \infty$ and $\lambda > 0$, the vertex furthest from the depot will be inserted.

5.2.4.2 Christofides, Mingozzi, and Toth Sequential Insertion Heuristic

Christofides, Mingozzi, and Toth [10] developed a somewhat more sophisticated two-phase insertion heuristic that also uses two user-controlled parameters λ and μ.

Phase 1. Sequential route construction.
Step 1 (first route). Set a first route index k equal to 1.
Step 2 (insertion costs). Select any unrouted vertex i_k to initialize route k. For every unrouted vertex i, compute $\delta_i = c_{0i} + \lambda c_{ii_k}$.
Step 3 (vertex insertion). Let $\delta_{i^*} = \min_{i \in S_k} \{\delta_i\}$, where S_k is the set of unrouted vertices that can be feasibly inserted into route k. Insert vertex i^* into route k. Optimize route k using a 3-opt algorithm. Repeat Step 3 until no more vertices can be assigned to route k.
Step 4 (next route). If all vertices have been inserted into routes, stop. Otherwise, set $k := k + 1$ and go to Step 2.

Phase 2. Parallel route construction
Step 5 (route initializations). Initialize k routes $R_t = (0, i_t, 0)$ $(t = 1, \dots, k)$, where k is the number of routes obtained at the end of Phase 1. Let $J = \{R_1, \dots, R_k\}$.
Step 6 (association costs). For each vertex i not yet associated with a route and for each feasible route $R_t \in J$, compute $\varepsilon_{ti} = c_{0i} + \mu c_{ii_t}$ and $\varepsilon_{t^*i} = \min_t \{\varepsilon_{ti}\}$. Associate vertex i with route R_{t^*} and repeat Step 6 until all vertices have been associated with a route.
Step 7 (insertion costs). Take any route $R_t \in J$ and set $J := J \setminus \{R_t\}$. For every vertex i associated with route R_t, compute $\varepsilon_{t'i} = \min_{R_t \in J} \{\varepsilon_{ti}\}$ and $\tau_i = \varepsilon_{t'i} - \varepsilon_{ti}$.
Step 8 (vertex insertion). Insert into route R_t vertex i^* satisfying $\tau_{i^*} = \max_{i \in S_t} \{\tau_i\}$, where S_t is the set of unrouted vertices associated with route R_t that can feasibly be inserted into route R_t. Optimize route R_t using a 3-opt algorithm. Repeat Step 8 until no more vertices can be inserted into route R_t.
Step 9 (termination check). If $|J| \neq \emptyset$, go to Step 6. Otherwise, if all vertices are routed, stop. If unrouted vertices remain, create new routes starting with Step 1 of Phase 1.

Comparisons between these two constructive algorithms were performed by Christofides, Mingozzi, and Toth [10] on their 14 standard benchmark instances. Results are presented in Table 5.3. This comparison indicates that the sequential insertion heuristic of Christofides, Mingozzi, and Toth [10] (CMT in the table) is superior to the Mole and Jameson algorithm. It yields better solutions in less computing time. It is also better

Table 5.3. *Computational comparison of two sequential insertion heuristics.*

Problem	Mole and Jameson[1] f^*	Mole and Jameson[1] Time[3]	CMT Two-Phase[2] f^*	CMT Two-Phase[2] Time[3]	Best known solution value
E051-05e	575	*5.0*	547	*2.5*	524.61[4]
E076-10e	910	*11.0*	883	*4.2*	835.26[4]
E101-08e	882	*36.0*	851	*9.7*	826.14[4]
E101-10c	879	*37.2*	827	*6.4*	819.56[4]
E121-07c	1100	*68.9*	1066	*11.3*	1042.11[4]
E151-12c	1259	*71.7*	1093	*11.8*	1088.42[4]
E200-17c	1545	*119.6*	1418	*16.7*	1291.45[4]
D051-06c	599	*5.1*	565	*2.6*	555.43[4]
D076-11c	969	*10.1*	969	*4.4*	909.63[4]
D101-09c	999	*28.6*	915	*7.0*	865.94[4]
D101-11c	883	*35.3*	876	*6.3*	866.37[4]
D121-11c	1590	*54.3*	1612	*8.7*	1541.14[5]
D151-14c	1289	*63.6*	1245	*10.1*	1162.55[5]
D200-18c	1770	*110.0*	1508	*15.8*	1395.85[4]

[1] Results were obtained by Christofides, Mingozzi, and Toth [10], except for the first three instances which were solved by Mole and Jameson [29].

[2] Christofides, Mingozzi, and Toth [10].

[3] Seconds on a CDC6600.

[4] Taillard [41].

[5] Rochat and Taillard [37].

than Christofides, Mingozzi, and Toth's implementation of the Clarke and Wright algorithm while requiring about twice the computing time. Again, the rounding convention is not specified, but solution values obtained with the CMT heuristic are in general far from the best known.

5.3 Two-Phase Methods

In this section we first describe three families of cluster-first, route-second methods. The last subsection is devoted to route-first, cluster-second methods. There are several types of cluster-first, route-second methods. The simplest ones, referred to as *elementary clustering methods*, perform a single clustering of the vertex set and then determine a vehicle route on each cluster. The second category uses a *truncated branch-and-bound* approach to produce a good set of vehicle routes. A third class of methods, called *petal algorithms*, produces a large family of overlapping clusters (and associated vehicle routes) and selects from them a feasible set of routes.

5.3.1 Elementary Clustering Methods

We now present three elementary clustering methods: the *sweep algorithm* (see Gillett and Miller [20], Wren [46], and Wren and Holliday [47]), the Fisher and Jaikumar [17] *generalized-assignment-based algorithm*, and the Bramel and Simchi-Levi [7] *location-based heuristic*. Only these last two heuristics assume a fixed value of the number of vehicles K.

5.3.1.1 Sweep Algorithm

The sweep algorithm applies to planar instances of the VRP. Feasible clusters are initially formed by rotating a ray centered at the depot. A vehicle route is then obtained for each cluster by solving a TSP. Some implementations include a postoptimization phase in which vertices are exchanged between adjacent clusters, and routes are reoptimized. To our knowledge, the first mentions of this type of method are found in a book by Wren [46] and in a paper by Wren and Holliday [47], but the sweep algorithm is commonly attributed to Gillett and Miller [20], who popularized it. A simple implementation of this method is as follows. Assume each vertex i is represented by its polar coordinates (θ_i, ρ_i), where θ_i is the angle and ρ_i is the ray length. Assign a value $\theta_i^* = 0$ to an arbitrary vertex i^* and compute the remaining angles from $(0, i^*)$. Rank the vertices in increasing order of their θ_i.

Step 1 (route initialization). Choose an unused vehicle k.
Step 2 (route construction). Starting from the unrouted vertex having the smallest angle, assign vertices to vehicle k as long as its capacity or the maximal route length is not exceeded. In tightly constrained DVRPs, 3-opt may be applied after each insertion. If unrouted vertices remain, go to Step 1.
Step 3 (route optimization). Optimize each vehicle route separately by solving the corresponding TSP (exactly or approximately).

5.3.1.2 Fisher and Jaikumar Algorithm

The Fisher and Jaikumar algorithm is also well known. Instead of using a geometric method to form the clusters, it solves a Generalized Assignment Problem (GAP). It can be described as follows.

Step 1 (seed selection). Choose seed vertices j_k in V to initialize each cluster k.
Step 2 (allocation of customers to seeds). Compute the cost d_{ik} of allocating each customer i to each cluster k as $d_{ik} = \min\{c_{0i} + c_{ij_k} + c_{j_k0},\ c_{0j_k} + c_{j_ki} + c_{i0}\} - (c_{0j_k} + c_{j_k0})$.
Step 3 (generalized assignment). Solve a GAP with costs d_{ij}, customer weights q_i, and vehicle capacity Q.
Step 4 (TSP solution). Solve a TSP for each cluster corresponding to the GAP solution.

The number of vehicle routes K is fixed a priori in the Fisher and Jaikumar heuristic. The authors proposed a geometric method based on the partition of the plane into K cones according to the customer weights. The seed vertices are dummy customers located along the rays bisecting the cones. Once the clusters have been determined, the TSPs are solved optimally using a constraint relaxation–based approach (Miliotis [28]). However, the Fisher and Jaikumar [17] article does not specify how to handle distance restrictions, although some are present in the test problems of Table 5.4.

5.3.1.3 Bramel and Simchi-Levi Algorithm

Bramel and Simchi-Levi [7] described a two-phase heuristic in which the seeds are determined by solving a capacitated location problem and the remaining vertices are gradually included into their allotted route in a second stage. The authors suggest first locating K seeds, called concentrators, among the n customer locations to minimize the total distance of customers to their closest seed while ensuring that the total demand assigned to any concentrator does not exceed Q. Vehicle routes are then constructed by inserting at each step the customer assigned to that route seed having the least insertion cost. Consider a partial route k described by the vector $(0 = i_0, i_1, \ldots, i_\ell, i_{\ell+1} = 0)$, let $T_k = \{0, i_1, \ldots, i_\ell\}$, and denote by $t(T_k)$ the length of an optimal TSP solution on T_k. Then the insertion cost d_{ik} of an unrouted customer i into route k is $d_{ik} = t(T_k \cup \{i\}) - t(T_k)$. Since computing d_{ik} exactly may be time consuming, two approximations $\overline{d}_{ik}$ are proposed: direct cost, $\overline{d}_{ik} = \min_{h=1,\ldots,\ell}\{2c_{ii_h}\}$, and nearest insertion cost, $\overline{d}_{ik} = \min_{h=0,\ldots,\ell}\{c_{i_h i} + c_{i i_{h+1}} - c_{i_h i_{h+1}}\}$. The authors showed that the algorithm defined by the first rule is asymptotically optimal.

5.3.2 Truncated Branch-and-Bound

Christofides, Mingozzi, and Toth [10] proposed a truncated branch-and-bound algorithm for problems with variable K, which is essentially a simplification of a previous exact algorithm by Christofides [8]. The search tree in this procedure contains as many levels as there are vehicle routes, and each level contains a set of feasible and nondominated vehicle routes. In the following implementation proposed by the authors, the tree is so simple that it consists of a single branch at each level, since all branches but one are discarded in the route selection step. However, a limited tree could be constructed by keeping a few promising routes at each level. In what follows, F_h is the set of free (unrouted) vertices at level h.

Step 1 (initialization). Set $h := 1$ and $F_h := V \setminus \{0\}$.
Step 2 (route generation). If $F_h = \emptyset$, stop. Otherwise, select an unrouted customer $i \in F_h$ and generate a set R_i of routes containing i and customers in F_h. These routes are gradually generated using a linear combination of two criteria: savings and insertion costs.
Step 3 (route evaluation). Evaluate each route $r \in R_i$ using the function $f(r) = t(S_r \cup \{0\}) + u(F_h \setminus S_r)$, where S_r is the vertex set of route r, $t(S_r \cup \{0\})$ is the length of a good TSP solution on $S_r \cup \{0\}$, and $u(F_h \setminus S_r)$ is the length of a shortest spanning tree over the yet unrouted customers.
Step 4 (route selection). Determine the route r^* yielding $\min_{r \in R_i} \{f(r)\}$. Set $h := h + 1$ and $F_h := F_{h-1} \setminus S_{r^*}$. Go to Step 2.

We provide in Table 5.4 comparative computational results for the four algorithms described in sections 5.3.1 and 5.3.2. Again, the comparison is made on the 14 Christofides, Mingozzi, and Toth [10] benchmark instances. Bramel and Simchi-Levi [7] used real distances. For the remaining algorithms, the rounding convention is not specified.

In terms of solution quality, these methods seem to perform better than the constructive algorithms presented in section 5.2. Also, for less computational effort, the truncated branch-and-bound algorithm tends to produce better solutions than the sweep algorithm. The Fisher and Jaikumar method seems to work well on most instances, but a number of reported solution values have been questioned by some authors (see Wark and Holt [45, p. 1163]).

Table 5.4. *Computational comparison of four constructive heuristics.*

Problem	Sweep[1]		Generalized assignment[2]		Location-based heuristic[3]		Truncated branch-and-bound[4]		Best known solution
	f^{*4}	Time[4]	f^{*2}	Time[2]	f^{*3}	Time[3]	f^{*4}	Time[4]	value
E051-05e	532	*12.2*	524	*9.3*	**524.6**	*68*	534	*7.1*	524.61[5]
E076-10e	874	*24.3*	857	*12.0*	848.2	*406*	871	*15.6*	835.26[5]
E101-08e	851	*65.1*	833	*17.7*	832.9	*890*	851	*38.2*	826.14[5]
E101-10c	937	*50.8*	824	*6.4*	826.1	*400*	816	*39.3*	819.56[5]
E121-07c	1266	*104.3*	—	—	1051.5	*1303*	1092	*51.1*	1042.11[5]
E151-12c	1079	*142.0*	1014	*33.6*	1088.6	*2552*	1064	*81.1*	1028.42[5]
E200-17c	1389	*252.2*	1420	*40.1*	1461.2	*4142*	1386	*138.4*	1291.45[5]
D051-06c	560	*11.4*	560	*15.2*	—	—	560	*5.3*	555.43[5]
D076-11c	933	*23.8*	916	*20.6*	—	—	924	*13.6*	909.63[5]
D101-09c	888	*58.5*	885	*52.2*	—	—	885	*33.4*	865.94[5]
D101-11c	949	*53.6*	876	*6.3*	—	—	878	*45.2*	866.37[5]
D121-11c	1776	*85.5*	—	—	—	—	1608	*61.8*	1541.14[6]
D151-14c	1230	*134.7*	1230	*121.3*	—	—	1217	*74.0*	1162.55[6]
D200-18c	1518	*238.5*	1518	*136.6*	—	—	1509	*135.6*	1395.85[5]

[1]Gillett and Miller [20], implemented by Christofides, Mingozzi and Toth [10].

[2]Fisher and Jaikumar [17]. Computing times are seconds on a DEC-10, considered by Fisher and Jaikumar to be seven times slower than CDC6600.

[3]Bramel and Simchi-Levi [7]. Computing times are seconds on an RS6000, Model 550. Nearest insertion costs were used in this implementation.

[4]Christofides, Mingozzi and Toth [10]. Computing times are seconds on a CDC6600.

[5]Taillard [41].

[6]Rochat and Taillard [37].

The location-based heuristic of Bramel and Simchi-Levi seems often to improve on the Fisher and Jaikumar method.

5.3.3 Petal Algorithms

A natural extension of the sweep algorithm is to generate several routes, called *petals*, and make a final selection by solving a set partitioning problem of the form

$$\min \sum_{k \in S} d_k x_k$$

subject to

$$\sum_{k \in S} a_{ik} x_k = 1 \qquad \forall\, i = 1, \ldots, n,$$
$$x_k \in \{0, 1\} \qquad \forall\, k \in S,$$

where S is the set of routes, $x_k = 1$ if and only if route k belongs to the solution, a_{ik} is the binary parameter equal to 1 only if vertex i belongs to route k, and d_k is the cost of petal k. If the routes correspond to contiguous sectors of vertices, then this problem possesses the column circular property and can be solved in polynomial time (Ryan, Hjorring, and Glover [38]).

This formulation was first proposed by Balinski and Quandt [3], but it becomes impractical when $|S|$ is large. Agarwal, Mathur, and Salkin [1] used column generation to solve small instances of the VRP optimally ($10 \le n \le 25$). Heuristic rules for producing a promising subset S' of simple vehicle routes, called 1-petals, have been put forward by Foster and Ryan [18] and by Ryan, Hjorring, and Glover [38]. Renaud, Boctor, and Laporte [36] go one step further by including in S' not only single vehicle routes but also configurations, called 2-petals, consisting of two embedded or intersecting routes. The generation of 2-petals is quite involved and is not be described here.

Renaud, Boctor, and Laporte [36] compared their results with their own implementation of the sweep algorithm (Gillett and Miller [20]) and of the petal algorithm of Foster and Ryan [18]. The 14 standard benchmark problems were solved with real distances. Results presented in Table 5.5 indicate that the 2-petal algorithm produces solutions whose value is on the average 2.38% above that of the best known (compared with 7.09% for sweep and 5.85% for 1-petal). Average computing times are 1.76 seconds for sweep, 0.26 second for 1-petal, and 3.48 seconds for 2-petal. The larger times taken by sweep and 2-petal are due to the postoptimization phase, which is absent from 1-petal. Sweep uses 3-opt, whereas 2-petal uses 4-opt* (Renaud, Boctor, and Laporte [35]).

5.3.4 Route-First, Cluster-Second Methods

Route-first, cluster-second methods construct in a first phase a giant TSP tour, disregarding side constraints, and decompose this tour into feasible vehicle routes in a second phase. This idea applies to problems with a free number of vehicles. It was first put forward by Beasley [4], who observed that the second-phase problem is a standard shortest-path problem on an acyclic graph and can thus be solved in $O(n^2)$ time using, for example,

Table 5.5. *Computational comparison of three petal heuristics.*

Problem	Sweep[1] f^{*3}	Sweep[1] Time[3]	1-petal algorithm[2] f^{*3}	1-petal algorithm[2] Time[3]	2-petal algorithm[3] f^{*3}	2-petal algorithm[3] Time[3]	Best known solution value
E051-05e	531.90	*0.12*	531.90	*0.10*	**524.61**	*0.76*	524.61[4]
E076-10e	884.20	*0.17*	885.02	*0.07*	854.09	*0.52*	835.26[4]
E101-08e	846.34	*1.18*	836.34	*0.32*	830.40	*3.84*	826.14[4]
E101-10c	919.51	*0.64*	824.77	*0.21*	824.77	*2.11*	819.56[4]
E121-07c	1265.65	*3.52*	1252.84	*0.61*	1109.14	*11.70*	1042.11[4]
E151-12c	1075.38	*2.53*	1070.50	*0.41*	1054.62	*5.93*	1028.42[4]
E200-17c	1396.05	*3.60*	1406.84	*0.41*	1354.23	*6.21*	1291.45[4]
D051-06c	560.08	*0.16*	560.08	*0.09*	560.08	*0.56*	555.43[4]
D076-11c	965.51	*0.19*	968.89	*0.07*	922.75	*0.43*	909.63[4]
D101-09c	883.56	*1.47*	877.80	*0.25*	877.29	*2.91*	865.94[4]
D101-11c	911.81	*0.85*	894.77	*0.17*	885.87	*1.69*	866.37[4]
D121-11c	1785.30	*2.24*	1773.69	*0.26*	1585.20	*3.31*	1541.14[5]
D151-14c	1220.71	*3.00*	1220.20	*0.26*	1194.51	*3.58*	1162.55[5]
D200-18c	1526.64	*4.91*	1515.95	*0.35*	1470.31	*5.19*	1395.85[4]

[1]Gillett and Miller [20], implemented by Renaud, Boctor, and Laporte [36].

[2]Foster and Ryan [18], implemented by Renaud, Boctor, and Laporte [36].

[3]Renaud, Boctor, and Laporte [36]. All computing times are seconds on a Sun Sparcstation 2 (210.5Mips, 4.2 Mflops), with 32 megabytes RAM.

[4]Taillard [41].

[5]Rochat and Taillard [37].

Dijkstra's [13] algorithm. In the shortest-path algorithm, the cost d_{ij} of traveling between nodes i and j is equal to $c_{0i}+c_{0j}+\ell_{ij}$, where ℓ_{ij} is the cost of traveling from i to j on the TSP tour. Haimovich and Rinnooy Kan [23] showed that if all customers have unit demand, this algorithm is asymptotically optimal. However, this is not so for general demands, except in some trivial cases (Bertsimas and Simchi-Levi [5]). We are not aware of any computational experience showing that route-first, cluster-second heuristics are competitive with other approaches.

5.4 Improvement Heuristics

Improvement heuristics for the VRP operate on each vehicle route taken separately or on several routes at a time. In a first case, any improvement heuristic for the TSP can be applied. In the second case, procedures that exploit the multiroute structure of the VRP can be developed.

5.4.1 Single-Route Improvements

Most improvement procedures for the TSP can be described in terms of Lin's [26] λ-opt mechanism. Here, λ edges are removed from the tour, and the λ remaining segments are reconnected in all possible ways. If any profitable reconnection (the first or the best) is identified, it is implemented. The procedure stops at a local minimum when no further improvements can be obtained. Checking the λ-optimality of a solution can be achieved in $O(n^{\lambda})$ time. Several modifications to this basic scheme have been developed. Lin and Kernighan [27] modified λ dynamically throughout the search. Or [31] proposed the Or-opt method, which consists of displacing strings of 3, 2, or 1 consecutive vertices to another location. This amounts to performing a restricted form of 3-opt interchanges. Checking Or-optimality requires $O(n^2)$ time. In the same spirit, Renaud, Boctor, and Laporte [35] developed a restricted version of the 4-opt algorithm, called 4-opt*, which attempts a subset of promising reconnections between a chain of at most w edges and another chain of two edges. Checking whether a solution is 4-opt* requires $O(wn^2)$ operations. Johnson and McGeoch [24] performed a thorough empirical analysis of these and other improvement procedures for the TSP and concluded that a careful implementation of the Lin–Kernighan scheme yields the best results on average. Since the description of this technique is rather extensive, readers are referred to the Johnson and McGeoch article for further details.

As mentioned, several heuristics described in this chapter already incorporate some form of reoptimization at intermediate steps. The Clarke and Wright algorithm is different in this respect in that it is typically implemented as a pure constructive heuristic, without reoptimization. To investigate the effect of postoptimization on the Clarke and Wright algorithm, we implemented two versions of 3-opt. In the first one, FI, the first improving move is performed, whereas in the second one, BI, the whole neighborhood is explored to identify the best improvement. Comparative results on the 14 Christofides, Mingozzi, and Toth [10] instances are presented in Table 5.6. Again, all running times are below 0.2 second on a Sun Ultrasparc 1 workstation (42 Mflops). The effect of applying 3-opt after the Clarke and Wright constructive heuristic is sometimes negligible, but it can reach 2% in some instances. The use of 3-opt, when applied after the sequential heuristic, is never sufficient to correct the relative inefficiency of the constructive step. The best solutions are consistently obtained by the parallel savings algorithm combined with 3-opt and BI. This algorithm is very fast to run (it requires an average of 0.13 second on the 14 benchmark instances) and produces solutions whose value is on average 6.71% above that of the best known. This compares with 7.08% for parallel savings without 3-opt, 18.75% for sequential savings without 3-opt, and 7.09% for the Renaud, Boctor, and Laporte [36] implementation of the sweep algorithm.

5.4.2 Multiroute Improvements

Thompson and Psaraftis [42], Van Breedam [43], and Kindervater and Savelsbergh [25] provide descriptions of multiroute edge exchanges for the VRP. These encompass a large number of edge exchange schemes used by several authors (see, e.g., Stewart and Golden [40], Dror and Levy [14], Salhi and Rand [39], Fahrion and Wrede [15], Potvin et al. [34], Osman [32], and Taillard [41]). The Thompson and Psaraftis paper describes a general b-cyclic, k-transfer scheme in which a circular permutation of b routes is considered and k customers from each route are shifted to the next route of the cyclic permutation. The authors show that applying specific sequences of b-cyclic, k-transfer exchanges (with $b = 2$ or b variable, and $k = 1$ or 2) yields interesting results. Van Breedam classified the improvement operations as string cross, string exchange, string relocation, and string mix, which all can be viewed as spe-

Table 5.6. *The effect of 3-opt on the Clarke and Wright algorithm.*

	Sequential				Parallel				
Problem	No 3-opt[1]	+ 3-opt FI[2]	+ 3-opt BI[3]	K[4]	No 3-opt[5]	+ 3-opt FI[6]	+ 3-opt BI[7]	K[8]	Best known solution value
E051-05e	625.56	624.20	624.20	5	584.64	578.56	578.56	6	524.61[9]
E076-10e	1005.25	991.94	991.94	10	900.26	888.04	888.04	10	835.26[9]
E101-08e	982.48	980.93	980.93	8	886.83	878.70	878.70	8	826.14[9]
E101-10c	939.99	930.78	928.64	10	833.51	824.42	824.42	10	819.56[9]
E121-07c	1291.33	1232.90	1237.26	7	1071.07	1049.43	1048.53	7	1042.11[9]
E151-12c	1299.39	1270.34	1270.34	12	1133.43	1128.24	1128.24	12	1028.42[9]
E200-17c	1708.00	1667.65	1669.74	16	1395.74	1386.84	1386.84	17	1291.45[9]
D051-06c	670.01	663.59	663.59	6	618.40	616.66	616.66	6	555.43[9]
D076-11c	989.42	988.74	988.74	12	975.46	974.79	974.79	12	909.68[9]
D101-09c	1054.70	1046.69	1046.69	10	973.94	968.73	968.73	9	865.9 [9]
D101-11c	952.53	943.79	943.79	11	875.75	868.50	868.50	11	866.37[9]
D121-11c	1646.60	1638.39	1637.07	11	1596.72	1587.93	1587.93	11	1541.14[10]
D151-14c	1383.87	1374.15	1374.15	15	1287.64	1284.63	1284.63	15	1162.55[10]
D200-18c	1671.29	1652.58	1652.58	20	1538.66	1523.24	1521.94	19	1395.85[9]

[1] Sequential savings.

[2] Sequential savings + 3-opt and first improvement.

[3] Sequential savings + 3-opt and best improvement.

[4] Sequential savings: number of vehicles in solution.

[5] Parallel savings.

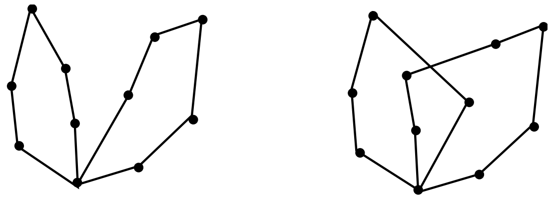
[6] Parallel savings + 3-opt and first improvement.

[7] Parallel savings + 3-opt and best improvement.

[8] Parallel savings: number of vehicles in solution.

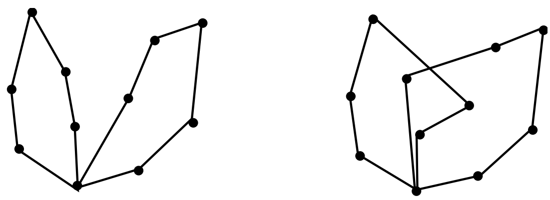
[9] Taillard [41].

[10] Rochat and Taillard [37].



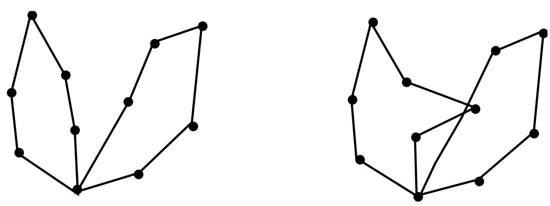
(a) Before **(b) After**

Figure 5.1. *String cross.*



(a) Before **(b) After**

Figure 5.2. *String exchange.*



(a) Before **(b) After**

Figure 5.3. *String relocation.*

cial cases of 2-cyclic exchanges, and provides a computational analysis on test problems. Kindervater and Savelsbergh define similar operations and perform experiments mostly in the context of the VRP with time windows.

We now summarize Van Breedam's analysis. The four operations considered are

- String cross (SC). Two strings (or chains) of vertices are exchanged by crossing two edges of two different routes; see Figure 5.1.
- String exchange (SE). Two strings of at most k vertices are exchanged between two routes; see Figure 5.2.
- String relocation (SR). A string of at most k vertices is moved from one route to another, typically with $k = 1$ or 2; see Figure 5.3.
- String mix (SM). The best move between SE and SR is selected.

To evaluate these moves, Van Breedam considers the two local improvement strategies, FI and BI. Van Breedam then defines a set of parameters that can influence the behavior of the local improvement procedure. These parameters are the initial solution (poor, good), the string length (k) for moves of type SE, SR, SM ($k = 1$ or 2), the selection strategy (FI, BI), and the evaluation procedure for a string length $k > 1$ (evaluate all possible string lengths between a pair of routes, increase k when a whole evaluation cycle has been completed without identifying an improvement move). To compare the various improvement heuristics, Van Breedam selects 15 tests problems among 420 instances. However, nine of these include either pickup and deliveries constraints or time-windows constraints and are therefore not relevant within the context of this chapter. The remaining six instances contain capacity constraints where all customers have the same demand, so that the capacity constraint is exactly satisfied and only SC or SE moves can be performed. Therefore, the following conclusions should be interpreted with caution. The first observation made by Van Breedam is that it is better to initiate the search from a good solution than from a poor one, in terms of both final solution quality and computing time. Also, the best solutions are obtained when SE moves are performed with a string length $k = 2$. However, using $k = 2$ is about twice as slow as using $k = 1$. Overall, SE moves appear to be the best. This is confirmed in a further comparison of local improvement, simulated annealing, and tabu search heuristics using various types of moves. The local improvement heuristic with SE moves yields solution values that are 2.2% above the best known, compared with 4.7% for SC moves, but computing times are more than four times larger with SC moves.

5.5 Conclusions

More than 35 years have passed since the publication of the savings heuristic for the VRP, and during this period a wide variety of solution techniques have been proposed. Comparisons between these heuristics are not always easy to make, especially since several implementation features can affect the performance of an algorithm. Also, the number and size of test problems used in the comparisons is rather limited, and researchers have not systematically applied the same rounding conventions, although this has been corrected in the last few years. It is now clear that in terms of solution quality, classical heuristics based on simple construction and local descent improvement techniques do not compete with the best tabu search implementations described in Chapter 6. However, several methods presented in this chapter can be easily adapted to other variants of the VRP and are easy to implement. This explains to a large extent their widespread use in commercial software. Thus, the Clarke and Wright algorithm remains probably the most popular method in practice. When followed by the BI version of 3-opt, it produces in almost no time solution values that fall within about 7% of the best known results. Much better performances are observed with some other algorithms (for example with the 2-petal algorithm), but the price to pay is often coding complexity.

Because metaheuristics for the CVRP outperform classical methods in terms of solution quality (and sometimes now in terms of computing time), we believe there is little room left for significant improvement in the area of classical heuristics. The time has come to turn the page.

Bibliography

[1] Y. Agarwal, K. Mathur, and H.M. Salkin. A set-partitioning-based exact algorithm for the vehicle routing problem. *Networks*, 19:731–749, 1989.

[2] K. Altinkemer and B. Gavish. Parallel savings based heuristic for the delivery problem. *Operations Research*, 39:456–469, 1991.

[3] M. Balinski and R. Quandt. On an integer program for a delivery problem. *Operations Research*, 12:300–304, 1964.

[4] J.E. Beasley. Route-first cluster-second methods for vehicle routing. *Omega*, 11:403–408, 1983.

[5] D.J. Bertsimas and D. Simchi-Levi. A new generation of vehicle routing research: Robust algorithms addressing uncertainty. *Operations Research*, 44:286–304, 1996.

[6] L.D. Bodin, B.L. Golden, A.A. Assad, and M. Ball. Routing and scheduling of vehicles and crews, the state of the art. *Computers and Operations Research*, 10:63–212, 1983.

[7] J. Bramel and D. Simchi-Levi. A location based heuristic for general routing problems. *Operations Research*, 43:649–660, 1995.

[8] N. Christofides. The vehicle routing problem. *RAIRO*, 10:55–70, 1976.

[9] N. Christofides. Vehicle routing. In E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys, editors, *The Traveling Salesman Problem*, Wiley, Chichester, UK, 1985, pp. 431–448.

[10] 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, UK, 1979, pp 315–338.

[11] G. Clarke and J.V. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12:568–581, 1964.

[12] M. Desrochers and T.W. Verhoog. A matching based savings algorithm for the vehicle routing problem. Technical Report Cahiers du GERAD G–89–04, École des Hautes Études Commerciales de Montréal, Canada, 1989.

[13] E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[14] M. Dror and L. Levy. Vehicle routing improvement algorithms: Comparison of a greedy and a matching implementation for inventory routing. *Computers and Operations Research*, 13:33–45, 1986.

[15] R. Fahrion and W. Wrede. On a principle of chain-exchange for vehicle-routing problems (1-vrp). *Journal of Operational Research Society*, 41:821–827, 1990.

[16] M.L. Fisher. Vehicle routing. In M.O. Ball, T.L. Magnanti, C.L. Monma, and G.L. Nemhauser, editors, *Network Routing*, *Handbooks in Operations Research and Management Science* 8, North-Holland, Amsterdam, 1995, pp. 1–33.

[17] M.L. Fisher and R. Jaikumar. A generalized assignment heuristic for the vehicle routing problem. *Networks*, 11:109–124, 1981.

[18] B.A. Foster and D.M. Ryan. An integer programming approach to the vehicle scheduling problem. *Operations Research*, 27:367–384, 1976.

[19] T.J. Gaskell. Bases for vehicle fleet scheduling. *Operational Research Quarterly*, 18:281–295, 1967.

[20] B.E. Gillett and L.R. Miller. A heuristic algorithm for the vehicle dispatch problem. *Operations Research*, 22:340–349, 1974.

[21] B.L. Golden and A.A. Assad. *Vehicle Routing: Methods and Studies*. North-Holland, Amsterdam, 1988.

[22] B.L. Golden, T.L. Magnanti, and H.Q. Nguyen. Implementing vehicle routing algorithms. *Networks*, 7:113–148, 1977.

[23] M. Haimovich and A.H.G. Rinnooy Kan. Bounds and heuristics for capacitated routing problems. *Mathematics of Operations Research*, 10:527–542, 1985.

[24] D.S. Johnson and L.A. McGeoch. The traveling salesman problem: A case study. In E.H.L. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, Wiley, Chichester, UK, 1997, pp. 215–310.

[25] G.A.P. Kindervater and M.W.P. Savelsbergh. Vehicle routing: Handling edge exchanges. In E.H.L. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, Wiley, Chichester, UK, 1997, pp. 337–360.

[26] S. Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44:2245–2269, 1965.

[27] S. Lin and B.W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.

[28] P. Miliotis. Integer programming approaches to the travelling salesman problem. *Mathematical Programming*, 10:367–378, 1976.

[29] R.H. Mole and S.R. Jameson. A sequential route-building algorithm employing a generalized savings criterion. *Operational Research Quarterly*, 27:503–511, 1976.

[30] M.D. Nelson, K.E. Nygard, J.H. Griffin, and W.E. Shreve. Implementation techniques for the vehicle routing problem. *Computers and Operations Research*, 12:273–283, 1985.

[31] I. Or. Traveling salesman-type combinatorial optimization problems and their relation to the logistics of regional blood banking. Ph.D. dissertation, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL, 1976.

[32] I.H. Osman. Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annals of Operations Research*, 41:421–451, 1993.

[33] H. Paessens. The savings algorithm for the vehicle routing problem. *European Journal of Operational Research*, 34:336–344, 1988.

[34] J.-Y. Potvin, T. Kervahut, B. Garcia, and J.-M. Rousseau. The vehicle routing problem with time windows—Part I: Tabu search. *INFORMS Journal on Computing*, 8:158–164, 1996.

[35] J. Renaud, F.F. Boctor, and G. Laporte. A fast composite heuristic for the symmetric traveling salesman problem. *INFORMS Journal on Computing*, 8:134–143, 1996.

[36] J. Renaud, F.F. Boctor, and G. Laporte. An improved petal heuristic for the vehicle routing problem. *Journal of Operational Research Society*, 47:329–336, 1996.

[37] Y. Rochat and E.D. Taillard. Probabilistic diversification and intensification in local search for vehicle routing. *Journal of Heuristics*, 1:147–167, 1995.

[38] D.M. Ryan, C. Hjorring, and F. Glover. Extensions of the petal method for vehicle routing. *Journal of Operational Research Society*, 44:289–296, 1993.

[39] S. Salhi and G.K. Rand. Improvements to vehicle routing heuristics. *Journal of Operational Research Society*, 38:293–295, 1987.

[40] W.R. Stewart Jr. and B.L. Golden. A Lagrangean relaxation heuristic for vehicle routing. *European Journal of Operational Research*, 15:84–88, 1984.

[41] E.D. Taillard. Parallel iterative search methods for vehicle routing problems. *Networks*, 23:661–673, 1993.

[42] P.M. Thompson and H.N. Psaraftis. Cyclic transfer algorithms for multi-vehicle routing and scheduling problems. *Operations Research*, 41:935–946, 1993.

[43] A. Van Breedam. An analysis of the behavior of heuristics for the vehicle routing problem for a selection of problems with vehicle-related, customer-related, and time-related constraints. Ph.D. dissertation, University of Antwerp, 1994.

[44] D. Vigo. A heuristic algorithm for the asymmetric capacitated vehicle routing problem. *European Journal of Operational Research*, 89:108–126, 1996.

[45] P. Wark and J. Holt. A repeated matching heuristic for the vehicle routing problem. *Journal of Operational Research Society*, 45:1156–1167, 1994.

[46] A. Wren. *Computers in Transport Planning and Operation*. Ian Allan, London, 1971.

[47] A. Wren and A. Holliday. Computer scheduling of vehicles from one or more depots to a number of delivery points. *Operational Research Quarterly*, 23:333–344, 1972.

[48] P. Yellow. A computational modification to the savings method of vehicle scheduling. *Operational Research Quarterly*, 21:281–283, 1970.