Tour splitting algorithms for vehicle routing problems

C. Prins, N. Labadi & M. Reghioui

International Journal of Production Research,
Vol. 47, No. 2, 15 January 2009, 507–535

Tour splitting algorithms for vehicle routing problems

C. PRINS*, N. LABADI and M. REGHIOUI

Institute Charles Delaunay, University of Technology of Troyes, Troyes, France

(*Revision received September 2008*)

Tour splitting heuristics for capacitated vehicle routing problems build one giant tour visiting all customers and split this tour into capacity-feasible vehicle trips. They are seldom used alone because of a reputation of limited performance. This paper describes how to improve them to obtain better solutions or tackle additional constraints. Numerical tests on the capacitated arc routing problem (CARP) and the capacitated vehicle routing problem (CVRP) show that randomized versions outperform classical constructive heuristics. A greedy randomized adaptive search procedure (GRASP) and an iterated local search (ILS) based on these principles even compete with published metaheuristics, while being faster and simpler.

Keywords: Vehicle routing problem; Capacitated arc routing problem; Route-first cluster-second heuristic; Greedy randomized adaptive search procedure; Iterated local search

1. Introduction

Vehicle routing problems are widespread in distribution and logistics. The most famous is the travelling salesman problem (TSP), which consists of determining a least-cost cycle visiting each node exactly once in a weighted graph. In reality, the limited capacity of vehicles raise more complicated problems with several trips. Consider a road or street network modelled by an undirected and weighted graph $G=(V,E,C)$. V is a set of nodes, containing a depot-node with a fleet of identical vehicles of capacity Q. E is a set of edges connecting the nodes. Each edge $[i,j]$ has a traversal cost $c_{ij}=c_{ji}$ (length, travelling time, etc.).

If the goal is to visit a subset of nodes with demands $q_i \leq Q$ and service costs w_i (e.g. customers) we have a node routing problem, whose best-known representative is the capacitated vehicle routing problem (CVRP). In the CVRP, all customers must be serviced by a set of vehicle trips with minimum total cost. A trip is a circuit performed by one vehicle, beginning and ending at the depot and visiting a subset of customers whose total demand does not exceed Q. Split deliveries are not allowed. As with its particular case the TSP, the CVRP is NP-hard and some instances with 100 customers are not yet solved to optimality.

In arc routing problems, vehicles have to service a subset of edges (e.g. streets) with known demands q_{ij} and service costs w_{ij}. The Chinese postman problem (CPP)

*Corresponding author. Email: christian.prins@utt.fr consists of computing a least-cost cycle visiting each edge at least once: it is polynomial but becomes NP-hard if some edges have no demand (rural postman problem (RPP)). The capacitated case is called the capacitated arc routing problem (CARP). It is NP-hard, like the CVRP, and the smallest standard instances not yet solved to optimality have 34 nodes and 65 edges.

In the sequel, the customers or roads to be serviced are called 'tasks'. The CVRP and the CARP can be met in distribution (delivery of goods to stores for the CVRP, winter gritting for the CARP) or pickup contexts (industrial waste for the CVRP, urban refuse for the CARP). We will say that a task is 'treated' to avoid specifying the context (delivered or collected).

Constructive heuristics for the CVRP and the CARP can be partitioned into four types:

(1) Insertion methods, which build trips one by one by successive insertions of tasks.
(2) Merge methods, which build one initial trip per task and then merge pairs of trips.
(3) Cluster-first route-second heuristics, which define a cluster of tasks for each vehicle and then build one trip in each cluster.
(4) Route-first cluster-second or tour splitting heuristics, which compute a giant tour by relaxing the capacity constraint and then split this tour into capacity-feasible trips.

Table 1 cites the main constructive heuristics for each type and each problem. Some of them have no established name. More details can be found in Hertz and Mittaz (2000) for CARP heuristics, in Laporte *et al.* (2000) for classical constructive heuristics for the CVRP, and in Cordeau *et al.* (2005) for recent metaheuristics for the CVRP.

This paper focuses on route-first cluster-second heuristics which are seldom used alone because of a reputation of poor performance. However, this reputation has never been based on intensive testing. Indeed, Beasley (1983) describes tour splitting methods for the CVRP but provides no numerical result. Ulusoy (1985) presents a similar heuristic for the CARP, but evaluated on one single graph and without comparison with other CARP heuristics.

Our goal is to improve the splitting principle to obtain more efficient algorithms. Section 2 introduces the basic splitting algorithm, called *Split*, and provides an efficient implementation. Section 3 describes possible applications. Section 4 is

Table 1. Examples of constructive heuristics for the CVRP and the CARP.

Type	CVRP	CARP
Insertion	Mole and Jameson (1976)	Path-scanning Golden *et al.* (1983)
Merge	Merge heuristic Clarke and Wright (1964)	Augment-merge Golden and Wong (1981)
Cluster-first route-second	Sweep heuristic Gillett and Miller (1974)	Belenguer *et al.* (1997)
Route-first cluster-second	Beasley (1983)	Ulusoy (1985)

devoted to more powerful splitting heuristics based on randomization and local improvement. The resulting algorithms are evaluated in section 5 for the CARP and in section 6 for the CVRP.

2. Basic splitting algorithm

2.1 *Principle*

A generic description is given here for a CVRP or a CARP with t tasks. The problem is assumed to be reduced to a complete graph. To reduce a CVRP, we only keep the depot (index 0) and the t node-tasks (customers) indexed from 1 to t. A distance matrix D, $(t+1)\times(t+1)$, is then pre-computed to know the cost d_{ij} of a shortest path between any two nodes i and j (tasks or depot). These shortest path can be computed using Dijkstra's algorithm for instance, see Cormen *et al.* (1990) for an efficient implementation.

To reduce a CARP, a dummy loop (index 0) is added on the depot, each edge is replaced by two opposite arcs (possible traversal directions) and the resulting arcs are indexed from 1 to $2t$. The arcs $i=(a, b)$ and $j=(b, a)$ which code the same edge are linked with a pointer *inv* such that $inv(i)=j$ and $inv(j)=i$. D is now a $(2t+1)\times(2t+1)$ distance matrix. The distance between two arcs $i=(a, b)$ and $j=(e,f)$ is the cost of a shortest path from node b to node e.

The basic splitting procedure *Split* requires one giant tour S, computed using any TSP (CVRP case) or RPP algorithm (CARP case). S is a permutation of the t task indices for the CVRP, not exactly for the CARP because each task may appear as one of its two directions. The depot at the beginning and at the end and shortest paths between consecutive tasks are implicit. With these conventions, the cost of S can be computed using equation (1).

$$C(S) = d(0,S_1) + \sum_{i=1}^{t-1} (w(S_i) + d(S_i,S_{i+1})) + w(S_t) + d(S_t,0) \tag{1}$$

Figure 1 provides one example for a CARP with null service costs, a vehicle capacity $Q=9$ and a giant tour S with five edge-tasks (thick segments) and demands in brackets. Thin segments indicate intermediate shortest paths and the dashed ones possible returns to the depot. The number close to one task or path segment is the cost of this task or path.

Split builds an auxiliary graph H with $t+1$ nodes indexed from 0 to t (lower part of figure 1). Each subsequence of tasks $(S_i, \dots, S_j)$ which can be treated as one trip (total demand not greater than Q) is modelled by one arc $(i-1, j)$, weighted by the trip cost. A shortest path from node 0 to node t in H (in **boldface**) corresponds to an optimal splitting of S, subject to the sequence. Here, a CARP solution with three trips and a total cost equal to 140 is obtained. The method is identical for a CVRP, except that S contains customer indices instead of edges.

2.2 *Efficient implementation*

By construction, H is acyclic. Let m be its number of arcs. The shortest path can be computed in $O(m)$ using Bellman's dynamic programming algorithm for directed

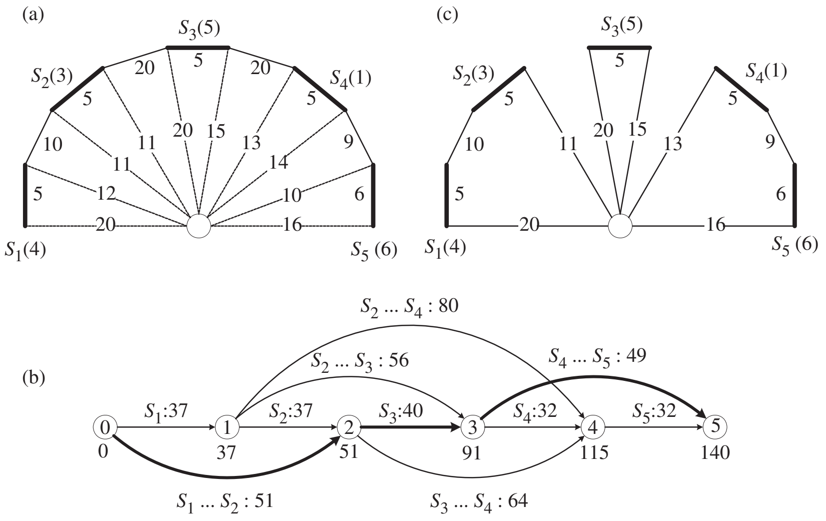

Figure 1. Principle of *Split* on a CARP instance. (a) Giant tour with 5 edge-tasks; (b) auxiliary graph H and shortest path (label under each node); (c) resulting trips.

acyclic graphs, see Cormen *et al.* (1990). In the worst case, $m = t \cdot (t+1)/2 = O(t^2)$ but a tighter expression can be derived from the minimal demand $q_{\min}$. A trip contains at most $b = \lfloor Q/q_{\min} \rfloor$ tasks, there are at most b trips starting with one given customer, each node has at most b outgoing arcs in H, H contains $m = O(bt)$ arcs and Bellman's algorithm runs in $O(bt)$. The second complexity is better when the minimum demand is relatively large compared to vehicle capacity.

Algorithm 1 implements *Split* without generating H explicitly. For each task S_j, it computes two labels: V_j, the cost of a shortest path from node 0 to node j in H and P_j, the predecessor of S_j on this path. At the beginning, the labels in V are null for node 0 and infinite for all other nodes. The main *for* loop scans S using index i. For each i, the *repeat* loop inspects each possible trip $(S_i, \ldots, S_j)$ starting with task S_i and evaluates its cost *cost* and its load *load*. This loop stops when *load* exceeds vehicle capacity or when j reaches the end of S.

A feasible trip $(S_i, \ldots, S_j)$ gives one arc $(i-1, j)$ in H. Instead of storing this arc, the label of node j is directly updated when improved. For each j, note that *cost* and *load* are updated in $O(1)$ from their values for $j-1$. This avoids an additional loop based on equation (1) to compute the cost of the trip. This trick is essential to achieve an $O(m)$ complexity, resulting from the two nested loops which are equivalent to an inspection of all arcs of H. Moreover, the algorithm runs in $O(t)$ space only, instead of $O(m)$ if H were generated explicitly.

The resulting algorithm is simple, compact and fast. For the example of figure 1, it returns $V = (0, 37, 51, 91, 115, 140)$ (these labels are given under each node) and $P = (0, 0, 0, 2, 2, 3)$. The cost of the shortest path is $V_t = 140$. The trips can be extracted easily with Algorithm 2 which uses the predecessors in P to backtrack from node t in the implicit graph H. This very simple algorithm returns the solution as a list of trips L, each trip being a list of tasks.

```
V(0) := 0
P(0) := 0
for i := 1 to t do V(i) := ∞ endfor
for i := 1 to t do
   j     := i
   load  := 0
   repeat
      load := load + q(S(j))
      if i = j then
         cost := D(0,S(i)) + w(S(i)) + D(S(i),0)
      else
         cost := cost - D(S(j-1),0) + D(S(j-1),S(j)) + w(S(j))
         + D(S(j),0)
      endif
      if (load ≤ Q) and (V(i-1) + cost < V(j)) then
         V(j) := V(i-1) + cost
         P(j) := i-1
      endif
      j := j + 1
   until (j > t) or (load > Q)
endfor
```

Algorithm 1. An efficient implementation of *Split* for the CVRP or the CARP.

3. Examples of applications

3.1 *Heuristics for the CVRP and the CARP*

An obvious utilization of *Split* is to provide heuristics for the CVRP and the CARP. The result is optimal, subject to the order defined by the given giant tour S but, obviously, the giant tour which gives the best solution after splitting is unknown. In particular, solving the associated NP-hard TSP or RPP to get an optimal giant tour rarely leads to an optimal CVRP or CARP solution. However, on average, *Split* gets good solutions from high-quality giant tours. Hence, in practice, the giant tour can be computed using any algorithm for the TSP or for the RPP.

As already mentioned, the principle of such heuristics was suggested by Beasley (1983) for the CVRP, but without numerical results. Concerning the CARP, Ulusoy (1985) exploited the fact that all edges are required in most instances

```
initialize one empty list of trips L
j := t
repeat
   insert one empty trip T at the beginning of L
   i := P(j)
   for k := i + 1 to j do
      insert task S(k) at the end of T
   endfor
   j := i
until i = 0
```

Algorithm 2. Extraction of trips after Algorithm 1.

from literature: in that case, the computation of an optimal giant tour becomes a polynomial Chinese postman problem.

Split is also useful to evaluate chromosomes coded as giant tours, in genetic algorithms (GAs) for vehicle routing problems. In that case, the GA searches the set of possible giant tours to find one that gives an optimal solution after splitting. An advantage of this encoding is to recycle existing crossovers for the TSP (LOX, OX, etc.). Very efficient memetic algorithms (MA: GA hybridized with a local search) based on these principles were designed for the CVRP (Prins 2004), the CARP (Lacomme *et al.* 2004) and more general problems on mixed graphs with required nodes, arcs and edges (Prins and Bouchenoua 2004).

3.2 *Fleet mix problems*

The vehicle fleet mix problem or VFMP (Golden *et al.* 1984) arises when vehicles are not yet purchased and must be selected among several models with known capacities and costs. The goal is to determine the trips and their vehicles to minimize the travel costs and the fleet cost. *Split* is easily modified for this problem: add to the cost of each arc in H (possible trip) the cost of the cheapest vehicle, among those with sufficient capacity.

3.3 *Maximum working time L for the drivers*

Like the maximum load Q, any subsequence of the giant tour S that violates a given maximum working time L is not represented by an arc in the auxiliary graph H. More generally, any lower or upper limit concerning a trip attribute (maximum volume for example) can be handled in the same way. The MA of Prins (2004) tackles in this way some distance-constrained CVRP instances from literature.

3.4 *Time windows*

When each task has a time window, the insertion of a customer in a trip can shift arrival times at subsequent customers, with possible window violations. In *Split*, time windows affect only the construction of the auxiliary graph: the shortest path computation does not change. For a given subsequence $(S_i, \ldots, S_j)$, starting from the depot, a waiting time is added each time the vehicle arrives at a customer before the beginning of his window. If the vehicle arrives too late, which is not allowed, the subsequence is not included in H. Labadi *et al.* (2007) applied this version of *Split* to a GRASP for the CARP with time windows (CARPTW).

3.5 *Limited number K of vehicles*

Compared to the basic *Split*, the construction of H is modified in the examples of 3.2, 3.3 and 3.4 but the shortest path algorithm is not affected. The two remaining applications require other algorithms. In general, the number of trips or vehicles used is not limited in the CVRP and the CARP: it is a decision variable. But if the fleet is limited to K vehicles, *Split* must compute a shortest path with at most K arcs. Bellman's algorithm for general graphs (Cormen *et al.* 1990) can be used for this purpose, because its iteration number i computes shortest paths with at most i arcs. Therefore, the algorithm may be stopped at the end of iteration K. This version of *Split* was used in a scatter search for a periodic CARP with limited fleet, raised by the long-term planning of municipal waste collection (Chu *et al.* 2005).

3.6 *Heterogeneous fleet*

In the heterogeneous fleet VRP (HFVRP), nt vehicle types are available and each type k is defined by an availability a_k, a capacity Q_k, a fixed cost f_k and a cost per distance unit v_k. Note that the VFMP corresponds to the case with infinite availabilities. *Split* must be modified to assign a vehicle type to each arc (trip) of H and compute a shortest path respecting the availability of each type. This kind of 'resource-constrained shortest path problem' is no longer polynomial and may be even infeasible. However, it can be solved in pseudo-polynomial time using dynamic programming techniques (Desrochers 1988).

The MA of Prins for the CVRP (2004) has been modified by Labadi *et al.* (2006) to tackle both the HFVRP and its particular case the VFMP. Soutera and Lacomme (2005) studied a memetic algorithm for the CARP with heterogeneous fleet. The chromosomes encoded as giant tours are evaluated by a greedy splitting heuristic instead of the exact but pseudo-polynomial version of *Split*. Yet not optimal, this heuristic accepts infeasible chromosomes by penalizing the edges which are not serviced. The MA progressively eliminates such infeasible solutions during its convergence.

4. Improved splitting algorithms

Three directions are investigated in this section to build improved splitting algorithms. The first one is to randomize the giant tour construction, to generate a series of distinct tours. The second direction is to improve the splitting process by a better exploitation of giant tours. The third direction consists of adding an optional local search procedure.

4.1 *Randomization of giant tour construction*

If randomness is introduced in a greedy heuristic, the resulting algorithm can be called at will to produce a sequence of giant tours. The simplest candidate for randomization is probably the 'nearest neighbour heuristic': starting from the depot, it extends the tour iteratively by joining the closest task v not yet treated. Let u be the last task reached by the tour, Z the set of free tasks (i.e. the ones not yet treated), $d_{\min} = \min\{d_{uz} \mid z \in Z\}$ and $d_{\max} = \max\{d_{uz} \mid z \in Z\}$. We designed three randomized versions called *RT*, *RTF* and *RC*.

RT version (random task). RT selects randomly the next task v in the restricted candidate list $L = \{z \in Z \mid d_{uz} = d_{\min}\}$. This works well for the CARP because $d_{\min}$ is often null, with several candidate edges. For instance, consider a crossroad with four streets not yet treated. If a vehicle treats one of these streets, toward the crossroad, it may choose one of the three adjacent streets to continue.

In most classical CVRP instances, customers are randomly spread in the Euclidean plane and L often contains one single candidate. In that case, a threshold $\theta \leq 1$ can be used to have more choice, via a modified candidate list $L' = \{z \in Z \mid d_{\min} \leq d_{uz} \leq d_{\min} + \theta \cdot (d_{\max} - d_{\min})\}$. For instance, if $\theta = 0.1$, *RT* will select one free task whose distance to u is between $d_{\min}$ and $d_{\min}$ plus 10% of the distance spread. The CARP version corresponds to the case $\theta = 0$.

RTF (RT flower). The candidates are partitioned in two lists L_1 and L_2. L_1 gathers the free tasks which drive the vehicle away from the depot, i.e. $L_1 = \{z \in Z \mid d_{uz} = d_{\min}$ and $d_{z0} \geq d_{u0}\}$. The other tasks are stored in L_2. Let *load* denote the current load of the tour. If one list is empty, *RTF* selects one task at random in the other list. If no list is empty, *RTF* randomly draws the next task v in L_1 if (*load* mod Q) $\leq Q/2$, and in L_2 otherwise. In other words, the giant tour tends to go away from the depot if its load is in one interval $[k.Q, (k+0.5).Q]$ for some integer $k \geq 0$, otherwise it gets closer to the depot. The name *RTF* comes from the flower-shaped resulting tour. A splitting algorithm is more likely to cut the sequence when the vehicle is close to the depot.

RC (random criterion). This third version is inspired by the *path-scanning* (*PS*) heuristic for the CARP (Golden *et al.* 1983). *PS* is a deterministic heuristic which builds the trips one by one using the nearest neighbour principle. The incumbent trip is iteratively extended by adding the nearest free task that fits the residual vehicle capacity. The particularity of *PS* is to use five rules to break ties:

- R_1: select the task which minimizes the cost to return to the depot.
- R_2: select the task which maximizes this return cost.
- R_3: select the task with maximum productivity (demand/service time).
- R_4: select the task with minimum productivity.
- R_5: apply rule R_2 if $load \leq Q/2$ and rule R_1 otherwise.

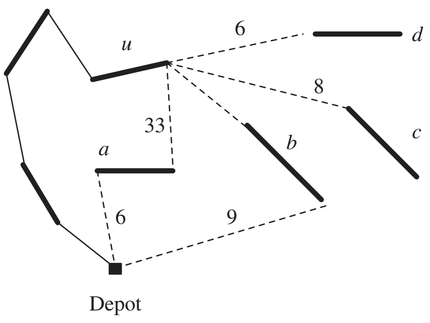

Figure 2. One iteration of Path-Scanning using rule R_1.

PS computes one solution for each rule and returns the best one. Figure 2 depicts one step with R_1. The incumbent trip ends at edge u. The two nearest tasks are a and b with $d_{\min} = 3$. *PS* selects a because the resulting trip gets closer to the depot (distance 6 instead of 9).

The heuristic *RC* uses the same rules as *PS*, but to build one giant tour. Like in *RT*, each iteration of *RC* determines first the restricted candidate list $L = \{z \in Z \mid d_{uz} = d_{\min}\}$. However, while *RT* draws the next task v at random in L, *RC* determines it by randomly selecting one of the five rules. For instance, if rule R_1 is drawn, *RT* selects in L the task minimizing the cost to go back to the depot. The goal of R_5 in the CARP is to start getting closer to the depot when the load reaches 50% of vehicle capacity. To obtain the same effect on the giant tour, *RC* works with a fictitious vehicle capacity equal to the sum of demands.

4.2 *Better exploitation of the giant tour*

Split-S or Split with shifts. In the basic version of *Split*, each sub-sequence $(S_i, S_{i+1}, \dots, S_j)$ of the giant tour S corresponds to a trip visiting customers in the same order. However, a rotation (circular left shift) can give a trip with smaller cost. For instance, a shift by one position corresponds to the order $(S_{i+1}, S_{i+2}, \dots, S_j, S_i)$. It is then possible to compute the best rotation, defined by the index r of the new first task ($r = i + 1$ in the example above), and to weight the associated arc $(i - 1, j)$ of the auxiliary graph H by the cost of the rotated trip. Index r must also be memorized with the arc, to perform the ad hoc rotations when the trips are extracted from the shortest path.

Figure 3 depicts rotations for the subsequence (S_2, S_3, S_4) of figure 1, assuming that $d(S_4, S_2) = 10$. The trip costs for $r = 2, 3, 4$ are respectively 80, 76 and 73. So, the cost of arc (1, 4) for this subsequence in the auxiliary graph is 73. If this arc belongs to the shortest path after splitting, the rotation for $r = 4$ will be performed to get the associated trip (S_4, S_2, S_3).

Property 1: Like *Split*, *Split-S* can be implemented in $O(m)$ for an auxiliary graph H of m arcs and it is not necessary to generate H explicitly.

Proof: Algorithm 1 for *Split* uses two nested loops to browse the subsequences $(S_i, \dots, S_j)$ which correspond to the m arcs of the auxiliary graph. Its complexity is

$O(m)$ because the load and cost for $Z' = (S_i, \ldots, S_j, S_{j+1})$ are deduced in $O(1)$ from those for $Z = (S_i, \ldots, S_j)$. The algorithm of *Split-S* is similar: it evaluates all feasible subsequences to avoid generating H but computes for each one the best entry point for a rotation. Let C_e (respectively C'_e) the cost of a trip with entry e in Z (Z') and r (r') the best entry in $Z(Z')$.

For each i, j is initialized to i and there is no choice for the best entry point: $r = i$. If task S_{j+1} is added, we have for any integer e in $]i, j]$: $C'_e = C_e - d(S_j, S_i) + d(S_j, S_{j+1}) + w(S_{j+1}) + d(S_{j+1}, S_i)$. Hence, the cost variations of all rotations with $e \in\ ]i, j]$ are identical, which means that the best entry point f among those in $]i, j]$ is the same for Z and Z'. Therefore, the best entry r' for Z' can be computed in $O(1)$ by comparing three entry points only: f, i and $j+1$. □

Split-F or Split with flips. The cost of a trip is sometimes improved by inverting (flipping) the direction of some edges. This version *Split-F* is explained in figure 4 for the CARP and the subsequence (S_2, S_3, S_4) of figure 1. The CVRP is not concerned because nodes have no traversal directions.

As mentioned in section 2.1, each required edge is coded as two arcs linked by a pointer *inv*. The upper layer contains the arcs of the giant tour. The lower layer contains the opposite arcs. Thin lines and unframed numbers represent shortest

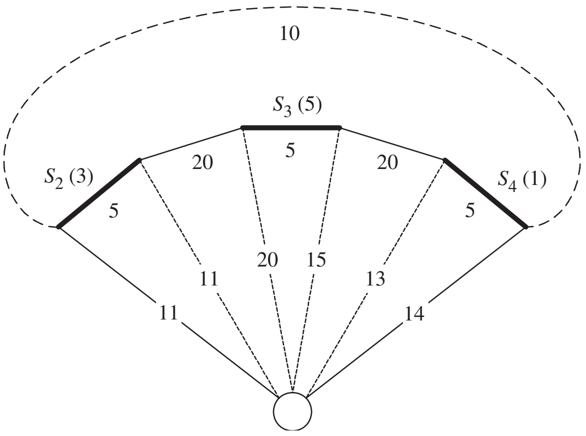

Figure 3. Rotations for a trip of three tasks.

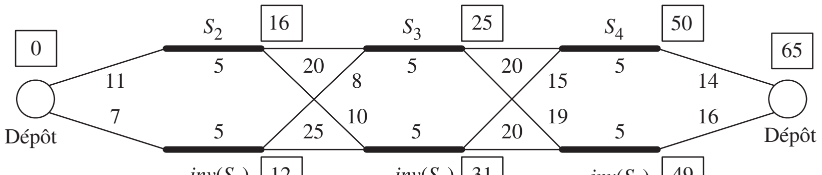

Figure 4. Optimal inversion of edge directions for a given sub-sequence.

paths in the road network and their costs. The costs of the upper layer come from figure 1, the others are imaginary. The best traversal directions for sub-sequence (S_2, S_3, S_4) are given by a shortest path in the acyclic graph of the figure. The labels calculated using Bellman's algorithm correspond to the framed numbers. The result is the trip ($inv(S_2)$, S_3, $inv(S_4)$) with cost 65 instead of 80 for the original directions.

Like in *Split-S*, this technique can be used to assign the best possible cost to the arc $(i-1, j)$ that models a subsequence $(S_i, \ldots, S_j)$ in the auxiliary graph H. The splitting process itself (computation of a shortest path in H) does not change.

Property 2: Like *Split* and *Split-S*, *Split-F* can be implemented in $O(m)$ for an auxiliary graph H of m arcs and it is not necessary to generate H explicitly.

Proof: Like in Algorithm 1, all feasible sub-sequences of S can be evaluated to avoid generating H explicitly. The claimed complexity can be achieved if the best edge directions for a subsequence $Z' = (S_i, \ldots, S_j, S_{j+1})$ can be derived in $O(1)$ from the best directions for $Z = (S_i, \ldots, S_j)$. The trick is to continue Bellman's algorithm with the labels already computed for S_j. For instance, consider figure 4 and $Z = (S_2, S_3)$: if S_4 is added, the shortest path computation can continue from labels 25 and 31. □

Split-SF or Split with shifts and flips. For the CARP, it is also possible to simultaneously determine the best rotation and the best edge directions for each sub-sequence, giving what we call the *Split-SF* version of *Split*. However, we have not found an implementation in $O(m)$ like *Split*, *Split-S* and *Split-F*. If b denotes the maximum number of tasks in a capacity-feasible subsequence, it was shown in section 2.2 that *Split* runs in $O(bt)$. Our best implementation for *Split-SF* runs in $O(b^2t)$.

Iterative versions. Consider one giant tour S^0 and a CVRP or CARP solution L^0 computed by *Split-S*, the version with shifts. Some sub-sequences (trips) have undergone rotations. The concatenation of the lists of tasks of these trips yields a new giant tour S^1. For instance, let $S^0 = (1, 2, 3, 4, 5, 6, 7, 8)$ and assume that *Split* finds three trips $(1, 2, 3), (5, 6, 4)$ and $(7, 8)$. The second one has been obtained by rotating sub-sequence $(4, 5, 6)$. Then $S^1 = (1, 2, 3, 5, 6, 4, 7, 8)$.

After splitting, S^1 gives a solution L^1 at least as good as L^0. This process can be repeated as long as it improves the incumbent solution. This iterative version of *Split-S* is called *Split-SI*. For the CARP, it is also possible to design iterative versions *Split-FI* and *Split-SFI* for *Split-F* and *Split-SF*. In all these iterative versions, *Split-S* behaves like a move in a local search.

4.3 *Local search*

Our local search works on complete solutions, not on giant tours. The moves tested are well known for the CVRP (Toth and Vigo 2002). They include the relocation of one task, the relocation of a chain of two tasks, the exchange of two tasks and 2-opt moves. All these moves are implemented to involve one or two trips. Similar moves can be defined for the CARP. The main difference is that the two directions of an

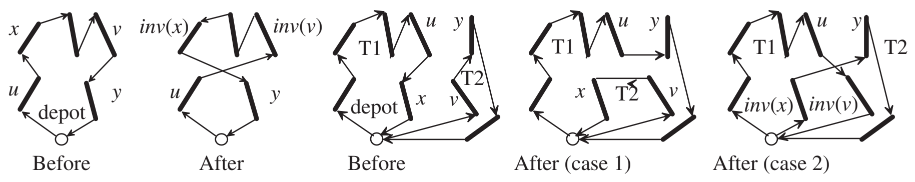

Figure 5. Examples of 2-opt moves for the CARP.

edge must be tested in reinsertions. For instance, relocating one edge *u* consists of removing it from its current trip and to reinsert either *u* or *inv*(*u*) in the same trip or in another trip.

What we call a 2-opt move for the CARP is to replace two intermediate shortest paths by two others. In figure 5, the paths between edges *u* and *x* and *v* and *y* are removed. On the left, the four edges are in the same trip and there is no choice to reconnect the trip. On the right, two trips are involved and they can be repaired in two ways, provided they satisfy capacity constraints. Graphically, such moves resemble the 2-opt moves for the TSP and CVRP, but the traversal directions of some edge-tasks can be inverted in case 2.

Each iteration of the local search evaluates the feasible moves. When one improving move is detected, it is executed and the remaining moves are not inspected (*first improvement local search*). The process is repeated until the solution can no longer be improved. Another option is to evaluate all moves to apply the best one (*best improvement local search*). Our tests have shown that this option is more time-consuming and does not yield better results.

4.4 *Generic GRASP based on tour splitting*

A generic greedy randomized adaptive search procedure (GRASP) for the CARP and the CVRP can be obtained by combining previous components. Its general structure is given by Algorithm 3. The input data include a known lower bound *LB*, the number of iterations *niter*, the kind of giant tour *tourkind* (*RT*, *RTF*, *RC*), the kind of splitting algorithm *splitkind* (*basic*, *S*, *F*, *SF*, *SI*, *FI*, *SFI*), and a Boolean *LS* equal to true if local improvement is required. The result is the best solution found, *bestsoln*.

The algorithm initializes the random generator to get reproducible results and then performs *niter* iterations. Each iteration creates one giant tour *tour* by calling *RandomGiantTour*. Depending on *tourkind*, this procedure calls one of the heuristics *RT*, *RTF* or *RC* described in section 4.1. Then, *tour* is split by the *SplitTour* procedure. According to *splitkind*, this procedure calls one of the splitting methods of section 4.2: the basic version (*Split*), the version with shifts (*Split-S*), with flips (*Split-F*), with both (*Split-SF*) or their iterative forms *Split-SI*, *Split-FI* and *Split-SFI*. If $LS = true$, the resulting solution *soln* is improved using the local search of 4.3. The iteration ends by updating the best solution when it is improved. When a lower bound *LB* is given, the main loop may be stopped when this bound is achieved.

```
procedure GRASP (LB,niter,tourkind,splitkind,LS,bestsoln)
    initialize random generator
    bestcost := ∞
    for iter := 1to niter do
        RandomGiantTour (tourkind,tour)
        SplitTour (tour,splitkind,soln)
        if LS = true then
            LocalSearch (soln)
        endif
        if cost(soln) < bestcost then
            bestsoln := soln
            bestcost := cost(bestsoln)
        endif
        if bestcost = LB then return
    endfor
endprocedure
```

Algorithm 3. Generic GRASP.

4.5 *Iterated local search based on tour splitting*

A tutorial about iterated local search (ILS) can be found in Lourenço *et al.* (2002). Starting from one heuristic solution improved by local search each iteration of this fast metaheuristic applies to the incumbent solution S one perturbation or mutation procedure, followed by local search. The resulting solution S' replaces S only in case of improvement.

A more evolved ILS combining the splitting method *SplitTour*, the local search *LocalSearch*, a mutation procedure *Mutate* and a concatenation procedure *SolnToTour* is sketched in Algorithm 4. The first two components were already described for the GRASP. ILS starts from a solution *bestsoln* computed using a merge heuristic, i.e. the merge heuristic of Clarke and Wright for the CVRP, or augment-merge for the CARP. This solution is improved by local search, then *SolnToTour* concatenates its trips to give the giant tour *besttour*.

Each iteration *iter* takes a copy *tour* of *besttour* and mutates it using *swaps* random exchanges of tasks. *Swaps* is set to a minimum value *minswaps* at the beginning. The new tour is split and the resulting solution *soln* is improved by local search. If *soln* is not better than the incumbent solution *bestsoln*, *swaps* is incremented but without exceeding a maximum value *maxswaps*. Otherwise, *soln* is assigned to *bestsoln*, *nswaps* is reset to *minswaps*, and the new incumbent giant tour is deduced from *bestsoln* using *SolnToTour*.

The algorithm stops after *niter* iterations, after *niterwi* iterations without improvement or when the lower bound *LB* which can be supplied on input is reached. Compared to a basic ILS, mutation intensity is dynamically modified and there is an alternation between complete solutions (with detailed trips) and

```
      procedure ILS
   (LB,niter,niterwi,splitkind,minswaps,maxswaps,bestsoln)
         initialize random generator
         initialize bestsoln using a merge heuristic
         LocalSearch (bestsoln)
         SolnToTour  (bestsoln,besttour)
         iterwi := 0
         swaps  := minswaps
        for iter := 1 to niter do
            tour := besttour
            Mutate (tour,swaps)
            SplitTour (tour,splitkind,soln)
            LocalSearch (soln)
            if cost(soln) < cost(bestsoln) then
               bestsoln := soln
               SolnToTour (bestsoln,besttour)
               iterwi := 0
               swaps  := minswaps
           else
               iterwi := iterwi + 1
               swaps  := min (swaps+1,maxswaps)
           endif
            if (cost(bestsoln) = LB) or (iterwi = niterwi) then return
        endfor
      endprocedure
```

Algorithm 4. Iterated local search.

giant tours. The splitting procedure complements the local search: it behaves like a general move, able to simultaneously modify all trip limits.

5. Results for the CARP

5.1 *Implementation*

Algorithm 3 and 4 with all their procedures were implemented for the CARP, in the Pascal-like language Delphi, and evaluated on a laptop PC with a 1.8 GHz Pentium 4 Mobile, 1024 MB RAM, and Windows 2000.

5.2 *Instances*

Three sets of classical CARP instances can be found at http://www.uv.es/~belengue/carp.html. The 23 *gdb* files have seven to 27 nodes and 11 to 55 edges, all required.

The 34 *val* files have 24 to 50 nodes and 34 to 97 edges, all required. The last set (*egl* files) comprises 24 larger instances with 77 to 140 nodes, 98 to 190 edges, and 51 to 190 required edges. All edge costs and demands are integer. Belenguer and Benavent (2003) computed good lower bounds for all these instances, using a cutting plane algorithm based on an integer linear model.

The current best metaheuristics are two tabu search methods (Hertz *et al.* 2000, Brandão and Eglese 2008), a guided local search (Beullens *et al.* 2003, Muyldermans 2003) and a memetic algorithm (Lacomme *et al.* 2004).

Longo *et al.* (2006) and Baldacci and Maniezzo (2006) proposed a technique to convert a CARP instance into an equivalent CVRP. The resulting instance is larger, but the idea is interesting because the CVRP has been more studied than the CARP and very sophisticated branch-and-cut methods are available. Using this technique, these authors improved several lower bounds in the three sets of CARP instances, showing that the best-known solutions found by metaheuristics are in fact optimal for all *gdb*, 28 *val* and 5 *egl* instances.

The current best lower bounds and best-known solutions used in this paper have been gathered by Brandão and Eglese (2008). The deviations to lower bounds and the number of optima given in our tables of results have been computed using these values.

5.3 *Results without local search*

Algorithm 3 with $niter = 20$ iterations but without local search ($LS = false$) was compared with two classical constructive heuristics for the CARP: path-scanning or PS (Golden *et al.* 1983) and augment-merge or AM (Golden and Wong 1981). The following performance factors were evaluated for each heuristic and each set of instances: the average and worst deviations to the best-known lower bound *LB* in percent, the number of proven optima (when *LB* is reached) and the average and worst running times.

The results are indicated with a common format in tables 2 to 4, for *gdb*, *val* and *egl* files. Each table shows solution values computed by the heuristics *RC*, *RT* and *RTF* described in section 4.1 The first column lists the performance factors, the next two recall the results of PS and AM for comparison, and the other columns give the results for the different versions of *Split* described in section 4.2: the basic form of algorithm 1 (*Split*), the version with shifts (*Split-S*), with flips (*Split-F*), with shifts and flips (*Split-SF*) and their iterative forms.

The tables show that all splitting algorithms with 20 random giant tours easily outperform the constructive heuristics PS and AM. Moreover, for one given giant tour heuristic (*RC*, *RT* or *RTF*), the basic version of *Split* is improved only a bit if edges may be flipped (*Split-F*), more if shifts are allowed (*Split-S*) and even more when both shifts and flips are enabled (*Split-SF*).

Iterative versions *SI*, *FI* and *SFI* bring further improvements but are more time-consuming. In particular, the combination *RTF* + *Split-SFI* solves to optimality eight out of the 23 *gdb* instances, which is remarkable. However, all running times remain negligible, except for *SF* and *SFI* which have the worst algorithmic complexities (see section 4.2).

When comparing the three heuristics for the giant tour, *RTF* and its flower-shaped tours give the best results, except on *egl* instances: in that case *RC* is a bit better with iterative versions of *Split* while *RT* is best for non-iterative versions. Moreover, in spite of their larger size, *egl* instances require less time than the *val*.

This behaviour can be explained: compared with vehicle capacity, the average demand per edge is larger in *egl* files. A first consequence is that many free edges are

Table 2. Results of Algorithm 3 for the 23 *gdb* instances ($LS = false$, $niter = 20$).

			Version of *Split*						
	PS	AM	Basic	*S*	*F*	*SF*	*SI*	*FI*	*SFI*
Using *RC*									
Avg. dev. *LB* %	10.790	7.190	5.520	4.670	5.440	4.300	4.330	5.070	3.030
Worst dev. %	33.050	24.240	17.160	15.180	17.160	14.540	15.180	16.500	13.530
Optima found	2	2	3	3	3	3	3	3	5
Avg. time (s)	0.001	0.001	0.001	0.001	0.001	0.007	0.002	0.003	0.016
Worst time (s)	0.002	0.001	0.003	0.003	0.004	0.031	0.005	0.006	0.073
Using *RT*									
Avg. dev. *LB* %	10.790	7.190	4.390	3.820	4.140	3.450	3.650	3.580	2.870
Worst dev. %	33.050	24.240	17.160	15.510	17.160	15.180	13.320	14.520	13.200
Optima found	2	2	3	3	3	4	3	5	7
Avg. time (s)	0.001	0.001	0.001	0.001	0.001	0.006	0.002	0.002	0.013
Worst time (s)	0.002	0.001	0.002	0.002	0.002	0.032	0.004	0.005	0.070
Using *RTF*									
Avg. dev. *LB* %	10.790	7.190	3.490	2.730	3.260	2.420	2.670	3.160	2.290
Worst dev. %	33.050	24.240	14.190	13.860	13.200	11.220	13.860	13.200	11.220
Optima found	2	2	4	5	5	7	5	5	8
Avg. time (s)	0.001	0.001	0.001	0.001	0.001	0.006	0.002	0.002	0.014
Worst time (s)	0.002	0.001	0.002	0.002	0.003	0.032	0.006	0.006	0.072

Table 3. Results of Algorithm 3 for the 34 *val* instances ($LS = false$, $niter = 20$).

			Version of *Split*						
	PS	AM	Basic	*S*	*F*	*SF*	*SI*	*FI*	*SFI*
Using *RC*									
Avg. dev. *LB* %	15.750	10.330	11.450	9.640	11.090	9.280	9.530	10.430	8.420
Worst dev. %	27.250	17.030	19.110	16.990	18.730	15.830	16.990	18.730	15.830
Optima found	0	0	0	0	0	0	0	0	0
Avg. time (s)	0.002	0.001	0.005	0.006	0.008	0.257	0.011	0.011	0.670
Worst time (s)	0.005	0.002	0.010	0.010	0.020	1.693	0.020	0.030	3.916
Using *RT*									
Avg. dev. *LB* %	15.750	10.330	10.990	9.200	10.360	8.660	8.990	9.970	8.150
Worst dev. %	27.250	17.030	18.060	17.950	17.680	16.600	17.950	17.570	16.410
Optima found	0	0	0	0	0	1	0	0	1
Avg. time (s)	0.002	0.001	0.003	0.004	0.004	0.253	0.008	0.010	0.664
Worst time (s)	0.005	0.002	0.008	0.008	0.010	1.647	0.016	0.022	4.522
Using *RTF*									
Avg. dev. *LB* %	15.750	10.330	8.710	7.240	8.120	6.660	7.070	7.830	6.270
Worst dev. %	27.250	17.030	17.450	14.800	17.100	14.540	14.800	16.360	13.350
Optima found	0	0	0	0	0	0	0	0	0
Avg. time (s)	0.002	0.001	0.004	0.005	0.004	0.259	0.009	0.011	0.623
Worst time (s)	0.005	0.002	0.010	0.011	0.010	1.743	0.020	0.030	4.196

discarded from candidate lists because of capacity violations. *RTF* has even less choice because it partitions the candidates in two lists (see section 4.1) and benefits from randomization to a lesser extent. A second consequence is a reduced average number of tasks per trip: the auxiliary graph is less dense, which accelerate splitting algorithms.

Finally, if the three sets of instances are considered in increasing order of size (*gdb*, *val*, *egl*), the number of problems solved to optimality decreases quickly (up to eight for *gdb* files, only one for the *val* and none for the *egl*), while deviations to the lower bounds increase.

Table 5 illustrates the impact of a growing number of iterations, using the *gdb* instances, the *RTF* heuristic to build giant tours and *Split-S* to evaluate them. The deviation to *LB* decreases more and more slowly and a majority of instances (16 out of 23) are solved to optimality. Similar decreases can be observed on the harder *val* and *egl* instances, but at most three optima are found for the *val* and none for the *egl*.

Table 4. Results of Algorithm 3 for the 24 *egl* instances ($LS = false$, $niter = 20$).

			Version of *Split*						
	PS	AM	Basic	*S*	*F*	*SF*	*SI*	*FI*	*SFI*
Using *RC*									
Avg. dev. *LB* %	24.690	21.810	14.020	12.180	13.470	11.870	11.830	12.710	10.910
Worst dev. %	30.920	30.410	17.430	16.400	16.880	15.970	16.270	16.350	14.930
Optima found	0	0	0	0	0	0	0	0	0
Avg. time (s)	0.010	0.002	0.015	0.016	0.016	0.079	0.033	0.043	0.242
Worst time (s)	0.021	0.010	0.030	0.030	0.040	0.220	0.070	0.100	0.771
Using *RT*									
Avg. dev. *LB* %	24.680	21.800	13.450	12.160	13.100	11.930	11.930	12.800	11.300
Worst dev. %	30.920	30.410	18.510	16.300	17.970	15.700	14.820	17.590	15.700
Optima found	0	0	0	0	0	0	0	0	0
Avg. time (s)	0.010	0.002	0.010	0.009	0.012	0.073	0.030	0.035	0.241
Worst time (s)	0.021	0.010	0.030	0.020	0.020	0.201	0.060	0.081	0.841
Using *RTF*									
Avg. dev. *LB* %	24.680	21.800	14.190	12.210	13.900	11.940	11.990	13.370	11.430
Worst dev. %	30.920	30.410	17.850	15.960	17.380	15.810	15.590	16.300	15.370
Optima found	0	0	0	0	0	0	0	0	0
Avg. time (s)	0.010	0.002	0.012	0.010	0.013	0.075	0.030	0.038	0.253
Worst time (s)	0.021	0.010	0.021	0.020	0.030	0.210	0.070	0.080	0.851

Table 5. Impact of iterations of Algorithm 3 on *gdb* files. Example using *RTF* and *Split-S*.

Iterations *niter*	20	50	100	500	1000
Avg. dev. *LB* %	2.730	2.100	1.980	1.350	1.200
Worst dev. %	13.860	13.860	12.870	8.580	8.580
Optima found	5	10	10	13	16
Avg. time (s)	0.001	0.002	0.004	0.017	0.027
Worst time (s)	0.002	0.006	0.012	0.070	0.110

Summarizing, and this was not known before, randomized tour splitting heuristics can easily outperform classical CARP heuristics in terms of deviations to lower bounds. They are also simpler to implement. Even if they are often slower than PS and AM, their running times are still negligible. Such fast splitting heuristics are therefore useful to provide metaheuristics with good initial solutions or tackle very large instances.

5.4 *Results with local search*

The GRASP (algorithm 3 with $LS = true$) and the ILS (Algorithm 4) were compared with three state-of-the-art metaheuristics: the tabu search of Hertz *et al.* (2000), the tabu search TSA version 2 of Brandão and Eglese (2008) and the memetic algorithm of Lacomme *et al.* (2004). These metaheuristics are respectively called TSH, TSA and MA in our tables. Their running times were scaled for our 1.8 GHz PC, using Dongarra's performance factors (2006).

The setting of parameters in GRASP and ILS is the same for all instances in a given set (*gdb*, *val* and *egl*) but slightly better results can be obtained by using one specific setting for each set of instances. These specific settings are listed in table 6. Section 5.3 has shown that random giant tours give on average better CARP solutions if the basic *Split* is replaced by enhanced versions, especially the iterative ones. This is no longer true in GRASP and ILS: the most powerful splitting algorithms seem to reduce the margin for improvement left to local search. This kind of mutual neutralization between components is for instance well known when designing a GRASP: the greedy randomized heuristic must not be 'too good', to ensure enough diversity. Table 6 shows that our best results were obtained without using iterative versions of *Split*. Moreover, the best version of ILS uses the basic *Split* for *val* and *egl* files.

The results are reported in tables 7 to 9, corresponding to the three sets of instances. Detailed results per instance are listed in tables A1 to A3, in the Appendix. Successive columns correspond to the performance criteria, the three published metaheuristics TSH, TSA and MA, the GRASP with 500, 1000 and 5000 iterations (G-500, G-1000 and G-5000), and the ILS.

For *gdb* instances (table 7), the best average results for the GRASP were obtained using the *RC* heuristic for the giant tours and *Split-S* (version with shifts) to

Table 6. Parameters used by GRASP and ILS for each set of instances.

Method	Parameter	*Gdb* files	*Val* files	*Egl* files
GRASP	*tourkind*	*RC*	*RTF*	*RT*
	splitkind	*Split-S*	*Split-SI*	*Split-SF*
	niter	500, 1000, 5000	500, 1000, 5000	500, 1000, 5000
ILS	*splitkind*	*Split-S*	Basic *Split*	Basic *Split*
	niter	500	1000	1000
	niterwi	200	1000	500
	minswaps	1	2	1
	maxswaps	5	4	3

split them. It was impossible to outperform TSA and MA but, compared to TSH, a GRASP with 500 iterations already improves the worst deviation, while being 25 times faster. With 1000 iterations, it improves both the average and worst deviations and is 16 times faster. With 5000 iterations, it finds one more optimum and is still five times faster than TSH and 2.8 times faster than MA. ILS is only a bit better than the GRASP with 5000 iterations, but it is six times faster.

Combining *RTF* with *Split-SI* provides best results on *val* instances (table 8). Again, it was not possible to do better than TSA and MA. But 500 iterations are enough for the GRASP to outperform the deviations of TSH to *LB*, while being 19 times faster. As from 1000 iterations, the GRASP takes place between TSH and MA. For 5000 giant tours, it finds 24 optima out of 34 instances and is still 3.5 times faster than TSH and twice faster than MA. Like in table 6, the ILS outperforms the GRASP in terms of solution gaps and it is here three times faster. The iterated local search is 11 times faster than the slowest algorithm, TSH.

The *egl* instances are much harder (table 9), as indicated by larger deviations and running times achieved by all algorithms. The best GRASP combines the *RT* giant tour heuristic and *Split* with shifts and flips. The running times of TSH are not available for these instances but the GRASP with 500 iterations already provides

Table 7. Results of GRASP and ILS on the 23 *gdb* instances.

Method	TSH	TSA	MA	G-500	G-1000	G-5000	ILS
Avg. dev. *LB* %	0.353	0.070	0.025	0.582	0.290	0.246	0.242
Worst dev. %	4.620	0.858	0.575	3.300	3.300	3.300	2.640
Optima found	19	21	22	14	18	20	19
Avg. time (s)	6.390	1.910	3.740	0.260	0.400	1.330	0.200
Worst time (s)	43.190	20.300	36.240	1.680	3.100	15.830	1.880

Table 8. Results of GRASP and ILS on the 34 *val* instances.

Method	TSH	TSA	MA	G-500	G-1000	G-5000	ILS
Avg. dev. *LB* %	1.595	0.216	0.256	1.280	0.975	0.683	0.561
Worst dev. %	8.096	2.124	2.852	5.985	5.985	5.598	3.668
Optima found	17	28	28	14	20	24	24
Avg. time (s)	45.210	15.690	27.150	2.370	4.260	12.770	4.180
Worst time (s)	250.120	169.630	152.250	7.160	14.190	65.770	20.240

Table 9. Results of GRASP and ILS on the 24 *egl* instances.

Method	TSH	TSA	MA	G-500	G-1000	G-5000	ILS
Avg. dev. *LB* %	3.633	1.307	1.389	3.581	3.215	2.873	1.822
Worst dev. %	7.354	3.037	3.209	5.913	5.683	5.427	3.486
Optima found	0	2	5	0	1	1	3
Avg. time (s)	Unknown	226.61	373.11	25.22	48.90	264.18	36.56
Worst time (s)	Unknown	742.47	1088.61	77.88	157.41	866.91	112.76

better solutions. The ILS clearly outperforms the best GRASP while being seven times faster. Compared with the best published metaheuristic (TSA), its average deviation to the lower bound is not as good but the worst deviation is quite close, one more optimum is obtained, and ILS is 6.2 times quicker.

Summarizing, GRASP and ILS take place between the tabu search TSH and the two other published metaheuristics, the memetic algorithm and the very recent tabu search TSA. However, our algorithms are faster than the other methods and have a simpler structure. The ILS is even remarkably fast and its small loss in terms of solution quality compared to the best metaheuristic (TSA) tends to reduce when instance size grows.

The fact that GRASP and ILS do not outperform MA and TSA is not really surprising. The MA is also based on giant tours and it works in parallel on a population of individuals, while GRASP and ILS have no recombination operator. TSA has a more involved structure. It performs five runs from five initial solutions computed by good constructive heuristics. Another run with different parameters is then applied to the best resulting solution.

An interesting property of ILS is that it gives better results and runs faster than a GRASP with the same number of calls to local search: compare ILS to the GRASP with 500 iterations for *gdb* files and 1000 iterations for *val* and *egl* files. Indeed, GRASP performs a kind of random sampling of local optima while ILS builds a coherent trajectory of local optima. Each ILS local optimum and each solution derived by mutation are in general relatively close in solution space, provided the perturbation is not too strong. This proximity allows the local search to quickly re-optimize the new solution, using a small number of improving moves.

A key-point in the efficiency of GRASP and ILS is to work with giant tours and a splitting algorithm, with an alternation between giant tours and complete CARP solutions in ILS. We evaluated a GRASP and an ILS operating directly on CARP solutions, i.e. the greedy randomized heuristic in GRASP and the mutation in ILS yield CARP solutions and the splitting algorithm is no longer used. The average deviation to the lower bound is approximately multiplied by two for GRASP and three for ILS.

A natural question is to wonder if the MA could be improved if the basic *Split* were replaced by a more effective version like *Split-S*. We tried such modifications and obtained degraded results. As already mentioned for the GRASP, the reinforcement of the splitting method weakens the local search applied afterward. Another way of reducing the gap between the GRASP and the MA could be the addition of a path re-linking process in the GRASP or after it, but the claimed simplicity would be lost.

6. Results for CVRP instances

6.1 *Implementation and instances*

The conversion of our CARP algorithms into CVRP versions was relatively straightforward, using the coding conventions stated in section 2.1. On one hand, some components are no longer necessary, like the splitting algorithms able to flip edges *Split-F*, *Split-SF* and their iterative versions. On the other hand, the CVRP version requires the implementation of the Clarke and Wright merge heuristic, to provide the ILS with an initial solution. Like the CARP, the CVRP software was written in Delphi and tested on the same laptop PC with a 1.8 GHz Pentium 4 Mobile, 1024 MB RAM, and Windows 2000.

The algorithms were evaluated with a set of 14 classical instances proposed by Christofides *et al.* (1979), downloadable from the OR Library: http://people.brunel.ac.uk/~mastjjb/jeb/orlib. These Euclidean files contain $t = 50$ to 199 tasks (customers). Files 6 to 10, 13 and 14 have a route-length restriction and non-zero service times. They are easily tackled using the modification of *Split* described in section 3.2. The other instances are pure CVRPs.

6.2 *Results with local search*

The tour splitting methods were compared with the Clarke and Wright merge heuristic (1964) or CW, CW improved by a call to a 3-opt local search after each merger of two trips (ICW), and the MA of Prins for the CVRP (2004). The results mentioned for CW and ICW come from our own implementations. The CW and ICW algorithms are already so good that splitting methods without local search do not do better. Hence, the results reported here concern only the two algorithms with local improvement, the GRASP and the ILS.

Compared with a CARP with the same number of tasks, the algorithms are slower and this can be explained by several factors. Firstly, CARP instances contain integer costs while CVRP instances require double precision real numbers for the Euclidean distance. This tradition in the CVRP metaheuristic community must be respected to allow reliable comparisons with published results, although it leads to slower computations.

Because of real numbers, the average number of successive neighbourhood explorations per call to the local search is much larger than for the CARP, since many moves have cost variations smaller than one unit. The local search is also less productive for instances 6 to 10, 13 and 14, because a lot of moves violate the route-length restriction. Finally, no tight lower bound is available to interrupt the search, contrary to CARP instances.

For these reasons, GRASP and ILS were evaluated with a reduced number of iterations. The best GRASP is based on the greedy randomized heuristic *RT* with a tolerance $\theta = 1.1$ (section 4.1) and $splitkind = S$. It was executed with $niter = 50$ and 500 iterations. The following parameters were used for ILS: $splitkind = S$, $niter = 500$, $niterwi = 200$, $minswaps = 2$ and $maxswaps = 3$. Note that both algorithms call *Split* with shifts: versions using the basic *Split* or the iterative version *Split-SI* give inferior results.

Some key-figures are given in table 10, while detailed results instance per instance are listed in table A4, in the Appendix. Due to the lack of lower bounds, the deviation to *LB* and the number of optima are replaced by the deviation to the best-known solutions (BKS), listed in Cordeau *et al.* (2005), and by the number of best-known solutions retrieved by the algorithms.

CW and ICW are interesting for their negligible running times, but the metaheuristics provide much smaller deviations. The winner but slowest is the MA. The GRASP with a comparable running time corresponding to 500 iterations gives inferior results. Like for the CARP, ILS offers an excellent compromise between

Table 10. Results of GRASP and ILS on the 14 CVRP instances.

Method	CW	ICW	MA	G-50	G-500	ILS
Avg. dev. BKS %	7.536	6.620	0.235	1.740	1.207	0.295
Worst dev. %	12.472	11.708	1.745	4.947	3.961	1.450
BKS found	0	0	8	2	5	6
Avg. time (s)	0.043	0.054	271.146	20.208	188.311	45.558
Worst time (s)	0.150	0.191	1381.080	98.983	1016.111	285.661

speed and solution quality. Here, ILS is quite close to MA for the average deviation to best-known solutions and the number of best-known solutions retrieved. Its worst deviation is even slightly better and it is SIX times faster.

7. Conclusion

Although tour splitting methods are seldom used in vehicle routing due to a reputation of limited performance, they can be strongly improved using randomization and local search. Tests on 81 CARP instances from literature show that even basic versions outperform more elaborated classical constructive heuristics. When local improvement is added to give a GRASP and an ILS, the resulting methods compete with sophisticated metaheuristics, while displaying shorter running times. Similar conclusions can be drawn from the tests with 14 CVRP instances, although local improvement must be added to outperform the classical Clarke and Wright heuristic combined with 3-opt.

Two obvious advantages of these methods are their speed and their extreme simplicity, at least when compact and efficient implementations like algorithm 1 are used. Moreover, the splitting principle is very flexible and can support different objective functions and additional constraints, provided the underlying shortest path problem remains polynomial.

Acknowledgement

This research has been funded by the Champagne-Ardenne Regional Council (France) and the European Social Fund.

Appendix

Tables A1 to A3 detail the synthetic tables 7 to 9 of section 5.3. The columns *n*, *m* and *t* denote the number of nodes, edges and required edges. For each instance, the other columns provide the best-known lower bound *LB* and the cost and running time (scaled in seconds for a 1.8 GHz Pentium 4M PC) of the compared algorithms, see text for explanations. The *average* and *worst* lines at the end contain either the average and worst deviations to the lower bound *LB* in % (for the cost columns) or the average and worst running times (in *time* columns). Asterisks denote proven optima, when the lower bound is reached.

Table A1. Results for TSH, TSA, MA, GRASP with 5000 iterations and ILS on the 23 *gdb* files.

Instance	*n*	*m*	*t*	*LB*	TSH	Time	TSA	Time	MA	Time	GRASP	Time	ILS	Time
gdb1	12	22	22	316	*316	2.23	*316	0.0	*316	0.00	*316	0.00	*316	0.01
gdb2	12	26	26	339	*339	3.66	*339	0.1	*339	0.31	*339	0.73	*339	0.00
gdb3	12	22	22	275	*275	0.05	*275	0.0	*275	0.04	*275	0.02	*275	0.01
gdb4	11	19	19	287	*287	0.06	*287	0.0	*287	0.00	*287	0.00	*287	0.00
gdb5	13	26	26	377	*377	3.96	*377	0.1	*377	0.08	*377	0.29	*377	0.09
gdb6	12	22	22	298	*298	0.60	*298	0.0	*298	0.12	*298	0.01	*298	0.01
gdb7	12	22	22	325	*325	0.00	*325	0.0	*325	0.03	*325	0.01	*325	0.00
gdb10	27	46	46	348	352	43.19	*348	1.2	350	28.19	*348	1.18	350	1.88
gdb11	27	51	51	303	317	38.17	*303	20.3	*303	5.02	313	15.82	311	0.97
gdb12	12	25	25	275	*275	1.10	*275	0.0	*275	0.04	*275	0.03	*275	0.01
gdb13	22	45	45	395	*395	1.62	*395	0.1	*395	0.89	*395	0.09	*395	0.05
gdb14	13	23	23	458	*458	14.61	*458	0.6	*458	6.92	*458	1.14	*458	0.02
gdb15	10	28	28	536	544	1.71	540	3.7	*536	5.25	544	2.63	544	0.24
gdb16	7	21	21	100	*100	0.34	*100	0.1	*100	0.03	*100	0.01	*100	0.00
gdb17	7	21	21	58	*58	0.00	*58	0.0	*58	0.00	*58	0.00	*58	0.00
gdb18	8	28	28	127	*127	1.20	*127	0.1	*127	0.04	*127	0.18	*127	0.00
gdb19	8	28	28	91	*91	0.00	*91	0.0	*91	0.03	*91	0.00	*91	0.00
gdb20	9	36	36	164	*164	0.20	*164	0.0	*164	0.08	*164	0.02	*164	0.01
gdb21	8	11	11	55	*55	0.14	*55	0.0	*55	0.00	*55	0.00	*55	0.00
gdb22	11	22	22	121	*121	6.73	*121	0.2	*121	0.23	*121	0.04	*121	0.05
gdb23	11	33	33	156	*156	0.80	*156	0.0	*156	0.12	*156	0.42	*156	0.03
gdb24	11	44	44	200	*200	2.39	*200	0.1	*200	2.37	*200	1.03	*200	0.43
gdb25	11	55	55	233	235	24.33	235	17.3	*233	36.24	235	6.88	235	0.84
Average					0.353%	6.39s	0.070%	1.91s	0.025%	3.74s	0.246%	1.33s	0.242%	0.20s
Worst					4.620%	43.19s	0.858%	20.30s	0.575%	36.24s	3.300%	15.83s	2.640%	1.88s
Optima/23					19		21		22		20		19	

Table A2. Results for TSH, TSA, MA, GRASP with 5000 iterations and IS on the 34 *val* files.

Instance	*n*	*m*	*t*	*LB*	TSH	Time	TSA	Time	MA	Time	GRASP	Time	ILS	Time
val1a	24	39	39	173	*173	0.01	*173	0.0	*173	0.00	*173	0.01	*173	0.03
val1b	24	39	39	173	*173	6.56	*173	0.7	*173	5.68	*173	1.19	*173	2.33
val1c	24	39	39	245	*245	65.99	*245	9.4	*245	20.30	*245	0.56	*245	1.15
val2a	24	34	34	227	*227	0.12	*227	0.0	*227	0.03	*227	0.00	*227	0.00
val2b	24	34	34	259	260	9.22	*259	0.2	*259	0.16	*259	0.90	*259	0.06
val2c	24	34	34	457	494	22.41	*457	6.1	*457	15.41	468	5.24	471	1.48
val3a	24	35	35	81	*81	0.54	*81	0.0	*81	0.03	*81	0.01	*81	0.00
val3b	24	35	35	87	*87	1.98	*87	0.0	*87	0.00	*87	0.09	*87	0.08
val3c	24	35	35	138	*138	29.49	*138	1.0	*138	19.99	*138	0.80	*138	0.13
val4a	41	69	69	400	*400	20.05	*400	0.3	*400	0.51	*400	0.78	*400	2.90
val4b	41	69	69	412	416	53.57	*412	4.3	*412	0.86	*412	1.76	*412	0.11
val4c	41	69	69	428	453	49.60	*428	29.6	*428	13.53	430	29.19	434	7.88
val4d	41	69	69	526	556	165.36	530	85.6	541	73.11	540	33.08	538	7.94
val5a	34	65	65	423	*423	2.69	*423	0.2	*423	1.32	*423	0.70	*423	3.77
val5b	34	65	65	446	448	29.31	*446	0.1	*446	0.74	*446	0.07	*446	0.00
val5c	34	65	65	473	476	37.71	474	8.2	474	71.51	474	21.29	474	6.91
val5d	34	65	65	573	607	158.67	583	57.0	581	64.24	596	24.42	589	6.67
val6a	31	50	50	223	*223	2.75	*223	1.2	*223	0.12	*223	0.04	*223	0.00
val6b	31	50	50	233	241	19.07	*233	9.9	*233	47.68	*233	1.61	*233	0.53
val6c	31	50	50	317	329	60.31	*317	17.8	*317	36.98	319	13.70	*317	1.16

val7a	40	66	66	279	*279	4.67	*279	0.8	*279	1.39	*279	1.38	*279	0.00
val7b	40	66	66	283	*283	0.01	*283	0.4	*283	0.31	*283	0.19	*283	0.23
val7c	40	66	66	334	343	85.98	*334	28.8	*334	71.63	338	33.26	335	10.15
val8a	30	63	63	386	*386	2.72	*386	0.2	*386	0.47	*386	0.07	*386	0.31
val8b	30	63	63	395	401	57.67	*395	1.4	*395	7.04	*395	0.91	*395	0.03
val8c	30	63	63	518	533	104.36	529	43.3	527	50.59	547	23.76	537	5.86
val9a	50	92	92	323	*323	20.18	*323	0.0	*323	12.95	*323	12.46	*323	2.16
val9b	50	92	92	326	329	42.40	*326	0.4	*326	20.81	*326	6.69	*326	2.78
val9c	50	92	92	332	*332	39.96	*332	0.3	*332	50.40	*332	14.79	*332	6.08
val9d	50	92	92	385	409	250.12	391	47.0	391	149.48	398	65.77	395	17.93
val10a	50	97	97	428	*428	3.91	*428	2.5	*428	18.04	*428	33.87	*428	11.63
val10b	50	97	97	436	*436	13.05	*436	1.4	*436	3.31	*436	28.84	438	20.24
val10c	50	97	97	446	451	66.18	*446	5.8	*446	12.25	*446	11.46	*446	2.77
val10d	50	97	97	525	544	110.67	530	169.6	530	152.25	539	65.14	537	18.97
Average					1.595%	45.21s	0.216%	15.69s	0.256%	27.15s	0.683%	12.77s	0.561%	4.18s
Worst					8.096%	250.12s	2.124%	169.63s	2.852%	152.25s	5.598%	65.77s	3.668%	20.24s
Optima/34					17		28		28		24		24	

Table A3. Results for TSH, TSA, MA, GRASP with 5000 iterations and ILS on the 24 *egl* files.

Instance	*n*	*m*	*t*	*LB*	TSH	TSA	Time	MA	Time	GRASP	Time	ILS	Time
egl-e1-A	77	98	51	3548	3625	3548	17.19	*3548	52.58	*3548	3.29	*3548	0.06
egl-e1-B	77	98	51	4498	4532	4533	21.78	*4498	49.19	4514	22.27	*4498	0.54
egl-e1-C	77	98	51	5566	5663	5595	18.74	5595	50.39	5660	20.05	5613	5.97
egl-e2-A	77	98	72	5018	5233	5018	49.31	*5018	108.03	5031	57.60	*5018	0.00
egl-e2-B	77	98	72	6305	6422	6343	51.88	6340	108.61	6397	47.21	6361	10.56
egl-e2-C	77	98	72	8234	8603	8347	61.21	8415	91.78	8531	46.73	8459	10.78
egl-e3-A	77	98	87	5898	5907	5902	60.12	*5898	171.34	5922	99.26	5922	12.50
egl-e3-B	77	98	87	7704	7921	7816	88.20	7822	180.79	7875	84.88	7907	18.69
egl-e3-C	77	98	87	10163	10805	10309	104.46	10433	146.10	10517	83.63	10383	16.90
egl-e4-A	77	98	98	6408	6489	6473	105.39	6461	206.64	6535	122.55	6502	29.83
egl-e4-B	77	98	98	8884	9216	9063	130.36	9021	221.50	9270	110.02	9068	20.84
egl-e4-C	77	98	98	11427	11824	11627	146.69	11779	178.68	11885	117.66	11806	22.03
egl-s1-A	140	190	75	5018	5149	5072	51.80	*5018	147.70	5113	66.24	5019	19.24
egl-s1-B	140	190	75	6384	6641	6388	62.84	6435	147.81	6516	56.59	6435	9.50
egl-s1-C	140	190	75	8493	8687	8535	61.60	8518	117.21	8568	66.90	8592	6.35
egl-s2-A	140	190	147	9824	10373	10038	307.30	9995	619.05	10273	441.61	10082	77.87
egl-s2-B	140	190	147	12968	13495	13178	348.68	13174	538.43	13498	407.31	13420	60.16
egl-s2-C	140	190	147	16353	17121	16505	401.18	16795	528.83	16919	501.50	16879	38.11
egl-s3-A	140	190	159	10143	10541	10451	431.04	10296	757.91	10531	537.33	10356	82.83
egl-s3-B	140	190	159	13616	14291	13981	443.80	14053	753.32	14170	517.52	13911	77.71
egl-s3-C	140	190	159	17100	17789	17346	463.87	17297	619.00	17895	625.89	17589	65.49
egl-s4-A	140	190	190	12143	13036	12462	541.96	12442	1088.61	12802	708.29	12497	112.76
egl-s4-B	140	190	190	16093	16924	16490	742.47	16531	1012.62	16917	729.15	16595	92.21
egl-s4-C	140	190	190	20375	21486	20733	726.83	20832	1058.47	21413	866.91	20931	86.64
Average					3.633%	1.307%	226.61s	1.389%	373.11s	2.873%	264.18s	1.822%	36.56s
Worst					7.354%	3.037%	742.47s	3.209%	1088.61s	5.427%	866.91s	3.486%	112.76s
Optima/24					0	2		5		1		3	

Notes: There is no *time* column for TSH in this table (running times not published). Using other parameters, the ILS improved the best-known solution for egl-e3-c, with 10292 instead of 10305 reported by Brandão and Eglese (2008).

Table A4. Results for ICW, MA, GRASP (50 and 500 itérations) and ILS on CVRP instances (see section 6.2).

File	*t*	BKS	ICW	Time	MA	Time	G-50	Time	G-500	Time	ILS	Time
1	50	*524.61	572.30	0.010	**524.61**	0.50	524.93	0.521	**524.61**	1.302	**524.61**	0.261
2	75	835.26	888.04	0.010	**835.26**	48.42	860.97	1.572	848.75	15.723	835.77	2.623
3	100	826.14	880.62	0.020	**826.14**	30.76	836.01	6.039	835.55	59.035	829.72	11.988
4	150	1028.42	1128.24	0.081	1031.63	307.43	1052.46	20.509	1051.36	210.312	1031.63	20.590
5	199	1291.29	1383.42	0.181	1300.23	886.66	1355.17	48.560	1342.44	473.861	1310.01	117.899
6	50	555.43	616.66	0.000	**555.43**	0.64	**555.43**	0.711	**555.43**	0.711	556.68	1.763
7	75	909.68	974.79	0.010	912.30	83.01	938.54	3.345	923.30	33.167	**909.68**	2.514
8	100	865.94	967.33	0.030	**865.94**	22.08	874.81	9.594	873.82	99.894	**865.94**	13.560
9	150	1162.55	1280.90	0.080	1164.25	404.89	1192.21	44.293	1170.63	441.034	1163.65	108.305
10	199	1395.85	1531.98	0.180	1420.20	1381.08	1461.75	98.983	1448.88	1016.111	1412.48	285.661
11	120	1042.11	1046.93	0.050	**1042.11**	17.38	1042.80	11.747	**1042.11**	13.369	**1042.11**	16.553
12	100	*819.56	824.42	0.030	**819.56**	2.78	**819.56**	2.333	**819.56**	2.323	**819.56**	4.657
13	120	1541.14	1583.01	0.040	1542.97	605.09	1558.37	26.117	1557.37	259.393	1546.71	48.530
14	100	866.37	868.50	0.031	**866.365**	5.33	867.13	8.582	**866.37**	10.124	**866.37**	2.904
Average			6.620%	0.054s	0.235%	271.15s	1.740%	20.208s	1.207%	188.31s	0.295%	45.558s
Worst			11.708%	0.181s	1.745%	1381.08s	4.947%	98.983s	3.961%	1016.11s	1.450%	285.661s
BKS found			0		8		2		5		6	

Values in boldface indicate best-known solutions retrieved by the algorithms.

References

Baldacci, R. and Maniezzo, V., Exact methods based on node-routing formulations for undirected arc routing problems. *Networks*, 2006, **47**, 52–60.

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

Belenguer, J.M., Benavent, E. and Cognata, F., Un metaheuristico para el problema de rutas por arcos con capacidades. *23th SEIO meeting*, 1997 (Valencia: Spain).

Belenguer, J.M. and Benavent, E., A cutting plane algorithm for the capacitated arc routing problem. *Comput. Oper. Res.*, 2003, **30**, 705–728.

Beullens, P., Muyldermans, M., Cattrysse, D. and Van Oudeheusden, D., A guided local search heuristic for the CARP. *Euro. J. Oper. Res.*, 2003, **147**, 629–643.

Brandão, J. and Eglese, R., A deterministic tabu search algorithm for the capacitated arc routing problem. *Comput. Oper. Res.*, 2008, **35**, 1112–1126.

Cordeau, J.F., Gendreau, M., Hertz, A., Laporte, G. and Sormany, J.S., New heuristics for the vehicle routing problem. In *Logistic Systems: Design and Optimization*, edited by A. Langevin and D. Riopel, pp. 279–298, 2005 (Wiley: New York).

Christofides, N., Mingozzi, A., Toth, P. and Sandi, C., The vehicle routing problem. In *Combinatorial optimization*, edited by the same authors, pp. 315–333, 1979 (Wiley: New York).

Chu, F., Labadi, N. and Prins, C., A scatter search for the periodic capacitated arc routing problem. *Eur. J. Oper. Res.*, 2006, **169**, 586–605.

Clarke, G. and Wright, J.W., Scheduling of vehicles from a central depot to a number of delivery points. *Oper. Res.*, 1964, **12**, 568–581.

Cormen, T.H., Leiserson, C.E. and Rivest, R.L., *Introduction to Algorithms*, 1990 (The MIT Press: Cambridge).

Desrochers, M., An algorithm for the shortest path problem with resource constraints. Research report G-88-27, GERAD, Montréal, Canada, 1988.

Dongarra, J.J., Performance of various computers using standard linear equations software. Report CS–89–85, Computer Science Department, University of Tennessee, Knoxville, 2006.

Gillett, B.E. and Miller, L.R., A heuristic algorithm for the vehicle dispatch problem. *Oper. Res.*, 1974, **22**, 340–349.

Golden, B.L., Assad, A., Levy, L. and Gheysens, F., The fleet size and mix vehicle routing problem. *Comput. Oper. Res.*, 1984, **11**, 49–66.

Golden, B.L., DeArmon, J.S. and Baker, E.K., Computational experiments with algorithms for a class of routing problems. *Comput. Oper. Res.*, 1983, **10**, 47–59.

Golden, B.L. and Wong, R.T., Capacitated arc routing problems. *Networks*, 1981, **11**, 305–315.

Hertz, A., Laporte, G. and Mittaz, M., A tabu search heuristic for the capacitated arc routing problem. *Oper. Res.*, 2000, **48**, 129–135.

Hertz, A. and Mittaz, M., Heuristic algorithms. In *Arc Routing: Theory, Solutions and Applications*, edited by M. Dror, 2000 (Kluwer: Boston).

Labadi, N., Lacomme, P. and Prins, C., A memetic algorithm for the heterogeneous fleet VRP. In *Proceedings of Odysseus 2006* (*3rd International Workshop on Freight Transportation and Logistics*), edited by E. Benavent *et al.*, pp. 209–213, 2006 (University of Valencia: Spain).

Labadi, N., Prins, C. and Reghioui, M., GRASP with path relinking for the capacitated arc routing problem with time windows. In *Applications of Evolutionary Computing*, edited by M. Giacobini *et al.*, Lecture Notes in Computer Science 4448, pp. 722–731, 2007 (Springer: Berlin).

Laporte, G., Gendreau, M., Potvin, J.Y. and Semet, F., Classical and modern heuristics for the vehicle routing problem. *Int. Trans. Oper. Res.*, 2000, **7**, 285–300.

Lacomme, P., Prins, C. and Ramdane-Chérif, W., Competitive memetic algorithms for arc routing problems. *Ann. Oper. Res.*, 2004, **131**, 159–185.

Longo, H., Poggi de Aragão, M. and Uchoa, E., Solving capacitated arc routing problems using a transformation to the CVRP. *Comput. Oper. Res.*, 2006, **33**, 1823–1837.

Lourenço, H.R., Martin, O. and Stützle, T., Iterated local search. In *Handbook of Metaheuristics*, edited by F. Glover and G. Kochenberger, pp. 321–353, 2002 (Kluwer: Boston).

Mole, R.H. and Jameson, S.R., A sequential route-building algorithm employing a generalized savings criterion. *Oper. Res. Quart.*, 1976, **27**, 503–511.

Muyldermans, L., Routing, districting and location for arc traversal problems. PhD thesis, Catholic University of Leuven, Belgium, 2003.

Prins, C., A simple and effective evolutionary algorithm for the Vehicle Routing Problem. *Comput. Oper. Res.*, 2004, **31**, 1985–2002.

Prins, C. and Bouchenoua, S., A memetic algorithm solving the VRP, the CARP, and more general routing problems with nodes, edges and arcs. In *Recent Advances in Memetic Algorithms*, edited by W. Hart, *et al.*, pp. 65–85, 2004 (Springer: Berlin).

Soutera, E. and Lacomme, P., Computerized decision-making system based on a hybrid genetic algorithm for heterogeneous VRP resolution, in *International Conference on Industrial Engineering and Systems Management* (*IESM*'05), Marrakech, Morocco, 2005, 10 pages on CD.

Toth, P. and Vigo, D., *The Vehicle Routing Problem*, 2002 (SIAM: Philadelphia).

Ulusoy, G., The fleet size and mix problem for capacitated arc routing. *Euro. J. Oper. Res.*, 1985, **22**, 329–337.