Annals of Operations Research 131, 159–185, 2004

Competitive Memetic Algorithms for Arc Routing Problems

PHILIPPE LACOMME lacomme@sp.isima.fr
University Blaise Pascal – Computer Science Laboratory, (CNRS UMR 6158), BP 10125, 63173 Aubière Cedex, France

CHRISTIAN PRINS * and WAHIBA RAMDANE-CHERIF {prins,ramdane}@utt.fr
University of Technology of Troyes – Industrial Systems Optimization Laboratory (LOSI), BP 2060, 10010 Troyes Cedex, France

Abstract. The Capacitated Arc Routing Problem or CARP arises in applications like waste collection or winter gritting. Metaheuristics are tools of choice for solving large instances of this NP-hard problem. The paper presents basic components that can be combined into powerful memetic algorithms (MAs) for solving an extended version of the CARP (ECARP). The best resulting MA outperforms all known heuristics on three sets of benchmark files containing in total 81 instances with up to 140 nodes and 190 edges. In particular, one open instance is broken by reaching a tight lower bound designed by Belenguer and Benavent, 26 best-known solutions are improved, and all other best-known solutions are retrieved.

Keywords: Capacitated Arc Routing Problem, CARP, metaheuristic, memetic algorithm

1. Introduction

Contrary to the well-known *Vehicle Routing Problem* (VRP), in which goods must be delivered to client nodes in a network, the *Capacitated Arc Routing Problem* (CARP) consists of visiting a subset of edges. CARP applications include for instance urban waste collection, winter gritting and inspection of power lines. From now on, to make the paper more concrete without loss of generality, examples are inspired by municipal refuse collection.

The basic CARP of literature tackles undirected networks. Each edge models a two-way street whose both sides are treated in parallel and in any direction (*zigzag collection*), a common practice in residential areas with narrow streets. A fleet of identical vehicles of limited capacity is based at a depot node. Each edge can be traversed any number of times, with a known traversal cost. Some edges are required, i.e., they have a non-zero demand (amount of waste) to be collected by a vehicle. The CARP consists of determining a set of vehicle trips of minimum total cost, such that each trip starts and ends at the depot, each required edge is serviced by one single trip, and the total demand processed by a trip fits vehicle capacity.

* Corresponding author.

The CARP is NP-hard, even in the single-vehicle case called Rural Postman Problem (RPP). Since exact methods are still limited to 20–30 edges (Hirabayashi, Saruwatari, and Nishida, 1992), heuristics are required for solving large instances, e.g., Augment-Merge (Golden and Wong, 1981), Path-Scanning (Golden, DeArmon, and Baker, 1983), Construct-and-Strike (Pearn's improved version, 1989), Augment-Insert (Pearn, 1991) and Ulusoy's tour splitting algorithm (Ulusoy, 1985).

The first metaheuristic for the CARP, a simulated annealing procedure, was designed by Eglese (1994) for solving a winter gritting problem. Several tabu search (TS) algorithms are also available, both for particular cases like the undirected RPP (Hertz, Laporte, and Nanchen-Hugo, 1999) or the mixed RPP (Corberán, Martí, and Romero, 2000) and for the CARP itself (Eglese and Li, 1996; Hertz, Laporte, and Mittaz, 2000). All these metaheuristics and classical heuristics may be evaluated thanks to lower bounds, generally based on linear programming formulations, see (Benavent, Campos, and Corberán, 1992; Belenguer and Benavent, 1998; Amberg and Voß, 2002). On most instances, the best-known lower bound is obtained by a cutting-plane algorithm (Belenguer and Benavent, 2003).

Compared to the VRP, the CARP has been relatively neglected for a long time but it attracts more and more researchers: successful applications are reported (Mourão and Almeida, 2000) and extensions are now investigated, for instance the directed RPP with turn penalties (Benavent and Soler, 1999), the multi-depot CARP (Amberg, Domschke, and Voß, 2000) and the CARP with intermediate facilities (Ghianni, Improta, and Laporte, 2001).

This paper presents powerful memetic algorithms (MA) for an extended CARP. Compared to an earlier genetic algorithm (GA) for the mixed CARP with forbidden turns (Lacomme, Prins, and Ramdane-Chérif, 2001), they handle other objectives, like the makespan or the number of vehicles used, and extensions like parallel arcs, turn penalties, a maximum trip length and a limited fleet. Several possible bricks for each MA step are designed and tested, e.g., a generational approach and a partial replacement procedure. The best resulting MA is twice faster, it improves 26 best-known solutions and tackles larger instances with 140 nodes and 190 edges.

The extended problem (ECARP) is presented in section 2. Three classical constructive heuristics are extended to the ECARP in section 3 to provide good initial solutions. Section 4 describes possible components for each step of memetic algorithms. Section 5 is devoted to computational evaluations: the best MA structure is defined after a preliminary testing and results are reported for three sets of benchmark instances.

2. Extended CARP model (ECARP)

2.1. Extensions considered and street modelling

For the sake of clarity, this subsection presents without mathematical symbols our extended problem and the modelling technique for the streets of a real network. Subsections 2.2–2.4 are respectively devoted to the required notation, to some complications raised by forbidden turns, and to the representation of solutions. The ECARP tackles the following extensions:

(a) *mixed multigraph* with two kinds of *links* (edges and arcs) and parallel links,

(b) *two distinct costs per link* (deadheading and collecting),

(c) *prohibited turns* (e.g., U-turns) and *turn penalties* (e.g., to penalize left turns),

(d) *maximum trip length* (an upper limit on the cost of any trip).

Like in the basic CARP, the depot is unique, the fleet is homogeneous, no demand exceeds vehicle capacity, and no split collection is allowed. The number of vehicles is a decision variable. To get feasible solutions, the maximum trip length allows a vehicle to reach any required link, collect it, and return to the depot. The cost of a trip comprises collecting costs (for each link collected) and deadheading costs (for each link traversed without collection). The goal is to find a set of trips of minimum total cost, covering all required links.

A mixed graph allows to model two kinds of non-required streets and three kinds of required streets. A non-required street is modelled either as one arc (1-way street) or two opposite arcs (2-way streets). A required street can be: (i) a 2-way street with zigzag collection (giving one edge, like in the basic CARP), (ii) a 2-way street with sides collected separately (giving two opposite arcs), and (iii) a 1-way street (modelled as one arc). We use a *mixed multigraph* to handle more complicated cases: for instance, two parallel arcs can model a one-way street too wide for zigzag collection and requiring two traversals, one for each side.

2.2. Graph transformation and notations

To ease algorithmic design, the mixed multigraph Γ is transformed into a fully directed graph in which each edge is replaced by two arcs with opposite directions. Only one of these arcs must be collected in any feasible solution. To ensure this, both arcs are linked by a pointer variable: when an algorithm selects one direction, both arcs can be marked "collected."

More precisely, the mixed multigraph is coded as a *fully directed graph*, $G = (N, A)$. N is a set of n nodes including a depot node s with identical vehicles of capacity W. A is a set of m arcs identified by indexes from 1 to m instead of pairs of nodes, to avoid ambiguities for parallel arcs. Each arc $u \in A$ begins at node $b(u)$, ends at node $e(u)$ and has a deadheading cost $c(u)$.

The τ required links of Γ (also called *tasks*) comprise ε required edges or *edge-tasks* and α required arcs or *arc-tasks*. They correspond in A to a subset R with $\rho = 2\varepsilon + \alpha$ arcs. Each arc $u \in R$ has a demand $q(u)$, a collecting cost $w(u)$ and a pointer $inv(u)$. All costs and demands are non-negative integers. Each arc-task of Γ is coded in R by one arc u with $inv(u) = 0$, while each edge-task gives two opposite arcs u and v, such that $inv(u) = v$, $inv(v) = u$, $q(u) = q(v)$, $c(u) = c(v)$ and $w(u) = w(v)$. To simplify, in the sequel, an arc in R is also called *task*, knowing that u and $inv(u)$ stands

Table 1
Glossary of mathematical symbols.

Mixed multigraph Γ	Data for each arc u		Miscellaneous	
$\tau = \varepsilon + \alpha$ no. of tasks in Γ	$b(u)$	begin node	s	depot node
ε, α no. of edge-tasks, no of arc-tasks	$e(u)$	end node	K	fleet size (variable)
$G = (N, A)$ directed encoding of Γ	$q(u)$	demand	W	vehicle capacity
n no. of nodes in N	$c(u)$	deadheading cost	L	maximum trip length
m no. of arcs in A	$w(u)$	service cost	$pen(u, v)$	penalty for turn (u, v)
$R \subseteq A$ subset of arcs coding the tasks	$inv(u)$	pointer to opposite arc	D	$m \times m$ distance matrix
$\rho = 2 \cdot \varepsilon + \alpha$ no. of arcs in R	$suc(u)$	set of successor-arcs	P	$m \times m$ predecessor matrix

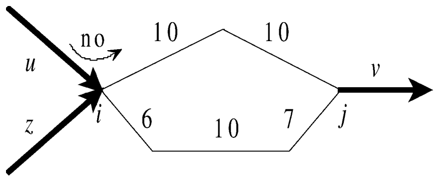


Figure 1. Notion of feasible deadheading path (see text).

for the same task (a required edge) if $inv(u) \neq 0$. Table 1 provides a quick reference for the remainder of the paper. From now on, having built the directed encoding of the mixed multigraph, all related algorithms work in that directed equivalence.

2.3. *Forbidden turns, turn penalties and distance matrix*

This subsection shows how to make forbidden turns transparent. In figure 1, the shortest path from node i to node j is the upper one (cost 20). However, the best path for a vehicle coming from arc u is the lower one (cost 23), because it is forbidden to turn left after u.

Such ambiguities can be avoided by considering paths between two arcs. Let $suc(u)$ be the set of allowed successor-arcs for arc u, i.e. $v \in suc(u)$ if $e(u) = b(v)$ and the turn (u, v) is allowed. Given two arcs u and v, we define a *feasible deadheading path* from u to v as a sequence of arcs $\mu = (u = u_1, u_2, \ldots, u_k = v)$, such that $u_{i+1} \in suc(u_i)$ for $i = 1, \ldots, k-1$. Its deadheading cost $c(\mu)$ is defined by equation (1). Note that the costs of u and v are not included.

$$c(\mu) = pen(u, u_2) + \sum_{i=2}^{k-1} \big(c(u_i) + pen(u_i, u_{i+1})\big). \tag{1}$$

Dijkstra's algorithm (e.g., see Cormen, Leiserson, and Rivest, 1990) can be adapted to pre-compute a shortest feasible path between all pairs of arcs, in two $m \times m$ matrices D and P. $D(u, v)$ is the cost of the shortest path found from arc u to arc v and $P(u, v)$ is the predecessor of v on this path. For instance, in figure 1, $D(u, v) = 23$ and $D(z, v) = 20$.

```
for v := 1 to m do D(u,v) := ∞; fix(v) := false endfor
for each v in suc(u) do D(u,v) := pen(u,v); P(u,v) := u endfor
for count := 1 to m do
  v        := argmin{D(u,z):fix(z)=false}
  fix(v)   := true
  for each z in suc(v) with D(u,v) + c(v) + pen(v,z) < D(u,z) do
    D(u,z) := D(u,v) + c(v) + pen(v,z)
    P(u,z) := v
  endfor
endfor
```

Algorithm 1. Algorithm for shortest paths from one given arc u to all other arcs.

Paths from/to the depot s are handled by putting in A one fictitious loop σ with $b(\sigma) = e(\sigma) = s$.

Algorithm 1 computes row u of D and P. It must be called m times with $u = 1, 2, \ldots, m$ to fill D and P. An arc v is said *fixed* when a shortest path from u to v is obtained. Initially, no arc is fixed and all paths from u have an infinite cost. Each iteration of the third *for* loop determines the destination arc v with the smallest path cost, among the arcs not yet fixed. This arc is fixed and each successor-arc z is checked to see if the provisional path from u to z can be improved.

The algorithm runs in $O(m^2)$. A heap data structure (Cormen, Leiserson, and Rivest, 1990) allows an $O(h \log m)$ version, with h the total number of allowed turns in G. So, D and P can be computed in $O(mh \log m)$ by calling the algorithm m times. For real street networks with $m \approx 4n$ and $h \approx 4m \approx 16n$, D and P are computed very quickly, in $O(n^2 \log n)$.

2.4. Implementation of trips and solutions

A solution T is a list $(T_1, \ldots, T_K)$ of K vehicle trips (K is a decision variable). Each trip T_i ($i = 1, 2, \ldots, K$) is a list of tasks $(T_{i1}, T_{i2}, \ldots, T_{i,|Ti|})$, with a total demand $load(T_i) \leqslant W$ and a total cost $cost(T_i) \leqslant L$ defined by equations (2)–(3). Implicitly, T_i starts and ends at the depot and shortest feasible paths are assumed between two tasks and between one task and the depot loop σ (cf. 2.3). The total cost of a solution T is the sum of its trip costs. Each arc-task appears once in T and each edge-task occurs as one of its two opposite arcs. So, T requires a space proportional to the number of tasks τ.

$$load(T_i) = \sum_{j=1}^{|T_i|} q(T_{ij}), \tag{2}$$

$$cost(T_i) = D(\sigma, T_{i1}) + \sum_{j=1}^{|T_i|-1} \big(w(T_{ij}) + D(T_{ij}, T_{i,j+1})\big) + D(T_{i,|T_i|}, \sigma). \tag{3}$$

3. Constructive heuristics for the ECARP

This section extends three classical CARP heuristics to the ECARP: Path-Scanning (Golden, DeArmon, and Baker, 1983), Augment-Merge (Golden and Wong, 1981) and Ulusoy's heuristic (Ulusoy, 1985). The extended versions are used in 4.5 to initialize the population of our MA. The main difference with classical versions is to use D, the arc-to-arc distance matrix of 2.3, to handle forbidden turns.

3.1. Extended Path-Scanning (EPS)

This heuristic builds one trip at a time. In constructing each trip, the sequence of tasks is extended by joining the task looking most promising, until capacity W or maximum trip length L are exhausted. For a sequence ending at a task u, the extension step determines the set M of tasks closest to u, not yet collected, and feasible for W and L. Five rules are used to select the next task v in M: (1) maximize the distance $D(v, \sigma)$ to the depot loop σ (cf. 2.3), (2) minimize $D(v, \sigma)$, (3) maximize the *yield* $q(v)/w(v)$, (4) minimize this yield, (5) use rule (1) if the vehicle is less than half-full, else use rule (2).

Once selected, v must be flagged as "collected," to avoid reselection in subsequent iterations. If v belongs to an edge-task, $inv(v)$ must be flagged too. EPS builds one solution per criterion and returns the best one. It can be implemented in $O(\tau^2)$, i.e., $O(n^2)$ for a real street network with $\tau \leqslant \rho \leqslant m \approx 4n$. In spite of its great simplicity, EPS gives good results in practice, thanks to compensation effects among criteria: the five solutions are never simultaneously bad.

3.2. Extended Augment-Merge (EAM)

The original version is illustrated in figure 2. τ trips are built (one per task) and sorted in decreasing cost order. For each trip T_i ($i = 1, 2, \ldots, \tau - 1$), the *augment phase* scans each smaller trip T_j ($j = i + 1, i + 2, \ldots, \tau$). If the unique task u of T_j is on a deadheading path of T_i and if $load(T_i) + q(u) \leqslant W$, T_j is absorbed. The cost of T_i does not vary because deadheading and service costs are equal in the basic CARP. However, the total cost decreases by $cost(T_j)$. Then, the *merge phase* evaluates the concatenation

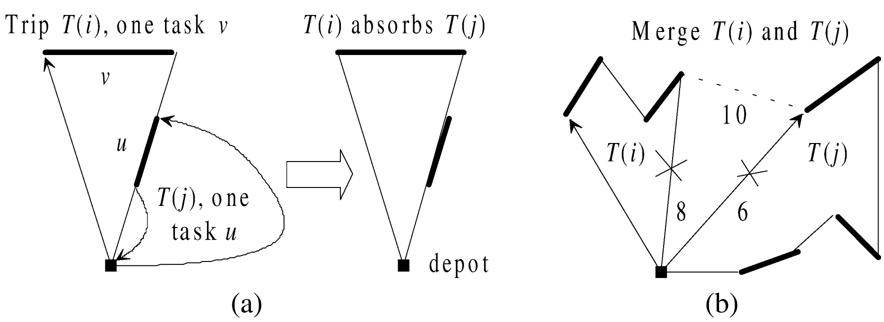


Figure 2. Principle of *augment* (a) and *merge* (b). Thick lines correspond to edge-tasks, thin lines to shortest deadheading paths.

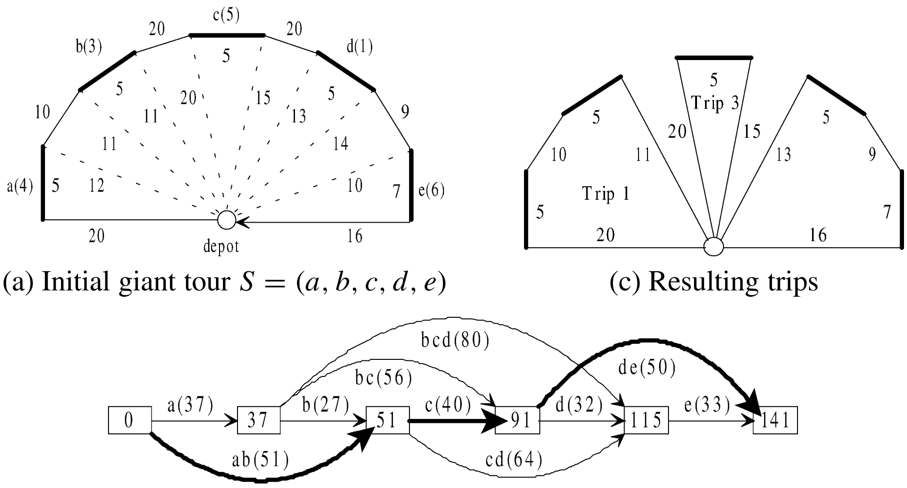
of any two trips, subject to W: e.g, in the figure, concatenating T_i then T_j yields a saving of 4. *Merge* concatenates the two trips with the largest positive saving. The process is repeated until no such concatenation is possible.

In the ECARP, each task u has two distinct costs $c(u)$ and $w(u)$. In *augment*, if trip T_i absorbs trip T_j with its task u, the total saving is now $cost(T_j) + c(u) - w(u)$ and is not always positive like in the basic CARP. In fact, some testing shows that *augment* can be suppressed without affecting average solution costs. So, we actually removed it. Moreover, matrix D is generally asymmetric for mixed networks and a trip is no longer equivalent to its mirror trip obtained by inverting the sequence of tasks. This gives up to 8 ways of concatenating two trips T_i and T_j: T_i then T_j or T_j then T_i, with each trip inverted or not. Note that a trip cannot be inverted if it contains arc-tasks, non-invertible. The extended heuristic EAM can be implemented in $O(\tau^2 \log \tau)$, i.e. $O(n^2 \log n)$ for real street networks.

3.3. Extended Ulusoy's heuristic (EUH)

The original heuristic for the basic CARP temporarily relaxes vehicle capacity W to compute a least-cost giant tour S covering the τ edge-tasks. If all edges are required, this sub-problem is an easy undirected Chinese postman problem. If not, it is an NP-hard rural postman problem that can be solved heuristically. Then, this tour is optimally split into capacity-feasible trips.

Figure 3 depicts the splitting procedure (*Split* in the sequel) for a giant tour $S = (a, b, c, d, e)$ with demands in brackets and deadheading costs, assuming $W = 9$. *Split* builds an auxiliary graph H with $\tau + 1$ nodes indexed from 0 onward. Each subsequence $(S_i, \ldots, S_j)$ corresponding to a feasible trip is modeled by one arc $(i - 1, j)$, weighted by the trip cost. A shortest path from node 0 to node τ in H (bold) indicates the optimal



(a) Initial giant tour $S = (a, b, c, d, e)$

(c) Resulting trips

(b) Auxiliary graph and shortest path (labels in each node)

Figure 3. Principle of *Split*.

splitting: 3 trips and a total cost 141. Note that H is an artificial construction standing in no relation with the CARP graph G.

In the ECARP version EUH, W but also the maximum trip cost L are relaxed to compute a good giant tour S in a mixed multigraph with forbidden turns and turn penalties, modelled by the directed multigraph G. We solve this mixed rural postman problem approximately, by running EPS (cf. 3.1) with a big value of W and L. For better results, we keep the 5 tours obtained by the 5 criteria of EPS, split them, and return the best solution. *Split* computes the load and cost of $(S_i, \ldots, S_j)$ using equations (2) and (3) and creates $(i-1, j)$ only if W and L are respected. Forbidden turns are entirely hidden in the arc-to-arc matrix D used in equation (3).

We now analyze complexity, missing in Ulusoy's paper. Path-Scanning (cf. 3.1) returns an initial giant tour in $O(\tau^2)$. Then, by construction, H is topologically sorted and contains $O(\tau^2)$ arcs. Bellman's algorithm (Cormen, Leiserson, and Rivest, 1990) can compute the shortest path in $O(\tau^2)$. The global complexity is then $O(\tau^2)$, i.e. $O(n^2)$ for a real street network with $\tau \leqslant \rho \leqslant m \approx 4n$. If the minimal demand $q_{\min}$ is large enough, a trip contains at most $\omega = \lfloor W/q_{\min} \rfloor$ tasks, H contains $O(\omega\tau)$ arcs and *Split* becomes faster, in $O(\omega\tau)$.

4. Components for memetic algorithms

This section describes the main features of our memetic algorithms: chromosome structure, chromosome evaluation, crossover operators, mutation by local search, population structure and initialization, population management. It describes several possible implementations for certain features. No computational evaluation is performed here: the best assembly of components is determined in section 5.

4.1. *Chromosomes: representation, evaluation and generation*

Most genetic algorithms for routing problems use quasi-direct representations of solutions, as sequences of tasks. A natural idea for the multi-vehicle case is to use sub-chromosomes (one per trip), separated by special symbols called *trip delimiters*. In that case, crossovers generally require a repair operator because children may contain overloaded trips. This technique is used for instance by Potvin and Bengio (1996) for the VRP with Time Windows. In our MAs, a chromosome S simply is a sequence of τ tasks, *without trip delimiters*, and with implicit shortest paths between consecutive tasks (see figure 4, presented later).

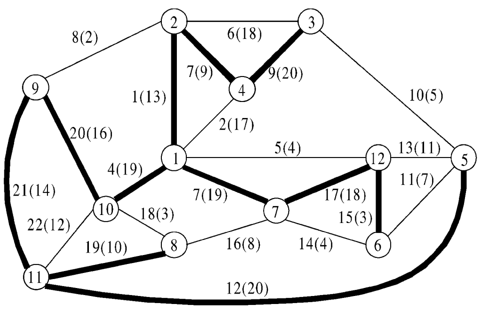
Clearly, S does not directly represent an ECARP solution but can be viewed as a giant trip ignoring capacity W and maximum trip cost L. The *Split* procedure to be described for Ulusoy's heuristic (cf. 3.3) is applied to S to get an ECARP solution. The fitness $F(S)$ of S is the total cost of this solution. Two good properties hold in our case: (1) *chromosomes are optimally evaluated with respect to their sequence*, (2) *there exists at least one optimal chromosome*, i.e., one giving an optimal solution after evaluation (consider one optimal solution and concatenate its trips).

These properties, yet trivial, are not always respected in the GAs whose chromosomes (*genotypes*) are abstractions of actual solutions (*phenotypes*). For instance, most GAs for scheduling problems use lists of operations, because it is sometimes complicated to design crossovers that directly combine two schedules. These lists can be converted into actual schedules, using a scheduling engine. The engines that build non-delay schedules are very good on average, but it is well known that the set of non-delay schedules contains no optimal solution for some instances (French, 1982). In that case, the second property does not hold.

A chromosome is created either by random generation (initial population), by crossover, or by converting an existing ECARP solution $T = (T_1, \ldots, T_K)$. In the third case, the trips are concatenated from left to right and the fitness is recomputed with *Split*, i.e. we forget $cost(T)$. There are two main reasons for this policy. First, the solution computed by *Split* is at least as good as T. Second, reproduction is based on a fitness-biased selection of parents (cf. 4.6): to be coherent, all chromosomes must be evaluated in the same way.

Compared to traditional local search, a genetic algorithm works on a population of solutions and its crossovers based on two solutions define larger neighbourhoods. This gives a spatial dimension to the search, often called *intrinsic parallelism*. Thanks to the two properties of our chromosome system, this parallelism is expected to find one optimal ECARP solution.

Figure 4 shows a basic CARP with $W = 5$, 22 edges, and $\tau = 11$ edge-tasks with unit demands (bold) and costs (in brackets). The underlying directed graph G with



Rank	:	1	2	3	4	5		6	7	8		9	10	11 = τ
Cut at	:							p=6 ↓		q=8 ↓				
Parent P1:		31	21	20	17	15	|	07	03	12	|	23	19	26
Parent P2:		34	09	29	20	41		26	43	25		15	39	23
Child C1:		09	20	41	26	43	|	07	03	12	|	15	39	23

```
P1 split : (31,  21,  20,  17,  15),(07),(03,  12,  23,  19,  26),  F(P1) = 318
P2 split : (34,  09,  29),(20,  41,  26),(43,  25,  15,  39,  23),  F(P2) = 324
C1 split : (09,  20,  41,  26),(43,  07),(03,  12,  15,  39,  23),  F(C1) = 311
```

Figure 4. A basic CARP instance with 11 tasks and an example of LOX crossover. Each edge is given with the arc index u for direction $(i, j), i < j$. The opposite arc is $inv(u) = 22 + u$.

$m = 44$ is not shown but each edge $[i, j]$ is given with the arc index (i, j) such that $i < j$, e.g., 7 for $(2, 4)$. The index for (j, i), not shown, is by convention $22 + u$, e.g., 29 for $(4, 2)$. Three chromosomes P1, P2 and C1 are given, for the LOX crossover explained in 4.3. The three last lines give the trips and solution costs found by *Split*. Note that some tasks are treated in two different directions by P1 and P2, e.g., edge $[3, 4]$ is collected as $(3, 4)$ in P2 (arc index 9) but as $(4, 3)$ in P1 (index 31).

4.2. *Efficient splitting procedures for two objective functions*

Algorithm 2 is an $O(\tau^2)$ version of *Split* working on a given chromosome (giant tour) S. It minimizes total cost (subject to the sequence of tasks defined by the chromosome) and, as a secondary objective, the number of vehicles. It runs in $O(\tau)$ space only, by avoiding an explicit generation of the auxiliary graph H. Two labels are used for each node i of H: V_i (cost of the shortest path from 0 to i in H) and N_i (number of arcs on that path, i.e. number of trips).

Thanks to two indexes i and j, the algorithm enumerates all subsequences $(S_i, \ldots, S_j)$ of S that correspond to feasible trips and computes their loads and costs using equations (2) and (3). Instead of creating one arc $(i - 1, j)$ for a trip composed of tasks S_i to S_j, like in 3.3, the labels of j are immediately updated. At the end, the total cost $F(S)$ and the minimum number of vehicles K for that cost can be read in V_τ

```
procedure split(S)
  V(0),N(0) := 0
  for i := 1 to τ do V(i) := ∞ endfor
  for i := 1 to τ do
    load, cost := 0; j := i
    repeat
      load := load + q(S(j))
      if i = j then
        cost := D(σ,S(i)) + w(S(i)) + D(S(i),σ)
      else
        cost := cost - D(S(j-1),σ) + D(S(j-1),S(j)) + w(S(j)) + D(S(j),σ)
      endif
      if (load ⩽ W) and (cost ⩽ L) then
        VNew := V(i-1) + cost
        if (VNew < V(j)) or ((VNew = V(j)) and (N(i-1) + 1 < N(j)) then
          V(j) := VNew
          N(j) := N(i-1) + 1
        endif
        j := j + 1
      endif
    until (j > τ) or (load > W) or (cost > L)
  endfor
endproc
```

Algorithm 2. *Split* procedure minimizing total cost and number of vehicles.

```
procedure split_for_makespan(S)
  build the auxiliary graph H explicitly
  K,V(0),V2(0),N(0),N2(0) := 0
  for i := 1 to τ do V(i),V2(i) := ∞ endfor
  repeat
    K := K+1; stable := true
    for i := 0 to τ−1 do
      for each successor j of i in H with max(V(i),Z(i,j)) < V2(j) do
        V2(j)  := max(V(i),Z(i,j))
        N2(j)  := N(i)+1
        stable := false
      endfor
    endfor
    V := V2; N := N2
  until stable or (K = Kmax)
endproc
```

Algorithm 3. *Split* procedure, version minimizing makespan subject to a limited fleet.

and N_τ. If required, the ECARP solution can be extracted by tracing the shortest path back.

Algorithm 3 implements *Split* for an interesting ECARP version found in waste collection. The fleet is this time limited. The number of trips K is free but cannot exceed the fleet size $K_{\max}$. All costs are times and the goal is to minimize makespan (longest trip duration). Note that the problem is trivially solved without $K_{\max}$, by treating each task by a separate trip. The algorithm uses the same labels as algorithm 2. It computes a min-max path from node 0 to node τ in the auxiliary graph H, here explicitly generated. Z_{ij} is the weight of arc (i, j) in H.

Each iteration of the *repeat* computes in $O(\tau^2)$ shortest paths in H with at most K arcs. It scans the arcs of H and stores improved label values in V2 and N2. V2 and N2 are copied into V and N at the end of the iteration. The algorithm stops when all labels are stable (checked with the boolean *stable*) or when $K = K_{\max}$. The chromosome S is *infeasible* if $V_\tau = \infty$. If not, the minimal makespan for S and the number of trips actually used are given by V_τ and N_τ. Since a shortest path from node 0 to node τ in H may have up to τ arcs, the algorithm runs in $O(\min(\tau, K_{\max}) \cdot \tau^2)$. Note that algorithm 3 can be simply adapted to minimize total cost (but subject to a limited fleet, contrary to algorithm 2), by replacing $\max(V_i, Z_{ij})$ by $V_i + Z_{ij}$.

4.3. Crossovers

Our chromosomes without trip delimiters can undergo classical crossovers for permutation chromosomes. The resulting children are immediately evaluated with *Split*. We tried the linear order crossover LOX and the order crossover OX (e.g., see Oliver, Smith, and Holland, 1987). LOX is designed for linear chromosomes (chromosomes coding objects that clearly have one begin and one end, like hamiltonian paths), while OX rather

```
function OX(P1,P2):chromosome
  //initialise miss to false:  no task is already in C
  for u := 1 to τ do miss(pack(u)) := true endfor
  //draw the substring P1(p)...P1(q) and copy it into C
  draw p in [1,τ]
  if p = 1 then draw q in [1,τ -1] else draw q in [p,τ] endif
  for i := p to q do
    C(i) := P1(i)
    miss(pack(P1(i))) := false
    if inv(P1(i)) ≠ 0 then miss(pack(inv(P1(i)))) := false
  endfor
  //browse P2 circularly, starting after the substring, to complete C.
  //C is also completed circularly, starting after the substring.
  i,j    := (q mod τ) + 1 //for LOX: i,j := 1
  istart := i
  repeat
    if miss(pack(P2(i))) then
      //for LOX, replace this comment by: if j = p then j := q + 1 endif
      C(j) := P2(i)
      miss(pack(P2(i))) := false
      if inv(P2(i)) ≠ 0 then miss(pack(inv(P2(i)))) := false
      j := (j mod τ) + 1
    endif
    i := (i mod τ) + 1
  until i = istart
  return C
endfunc
```

Algorithm 4. Adaptation of the OX crossover for the ECARP.

concerns circular permutations (like TSP tours). Intuitively, the best choice that will be confirmed in section 5 should be OX, because the chromosome before splitting may be viewed as a circular object (giant trip).

Given two parents P1 and P2 with τ tasks, both crossovers draw two cutting sites p and q with $1 \leqslant p \leqslant q \leqslant \tau$. To get the first child C1, LOX copies $\mathrm{P1}(p), \ldots, \mathrm{P1}(q)$ into $\mathrm{C1}(p), \ldots, \mathrm{C1}(q)$. P2 is then swept from left to right and the tasks missing in C1 are used to fill $\mathrm{C1}(1), \ldots, \mathrm{C1}(p-1)$ then $\mathrm{C1}(q+1), \ldots, \mathrm{C1}(\tau)$. In OX, the sequence for C1 is $\mathrm{P1}(p), \ldots, \mathrm{P1}(q)$ followed by $\mathrm{P2}(q+1), \ldots, \mathrm{P2}(\tau), \mathrm{P2}(1), \ldots, \mathrm{P2}(p-1)$, with restriction that tasks from P2 are taken only if missing in C1. However, C1 is interpreted as a circular list and the result stored such that $\mathrm{C1}(p) = \mathrm{P1}(p)$. For both crossovers, the other child C2 is obtained by exchanging the roles of P1 and P2.

In the ECARP, a task u is "missing" in C1 if both u and $inv(u)$ are not yet in C1. Algorithm 4 is an ad-hoc version of OX, implemented as a function of two parents P1, P2 that returns a child C (the other child is obtained by swapping the parents in the call). An $O(\tau)$ complexity is achieved using a table *pack* that maps the indexes of arcs of R (in $1, \ldots, m$) into $1, \ldots, \tau$. *Pack* is built once for all in $O(m)$, when initializing the MA.

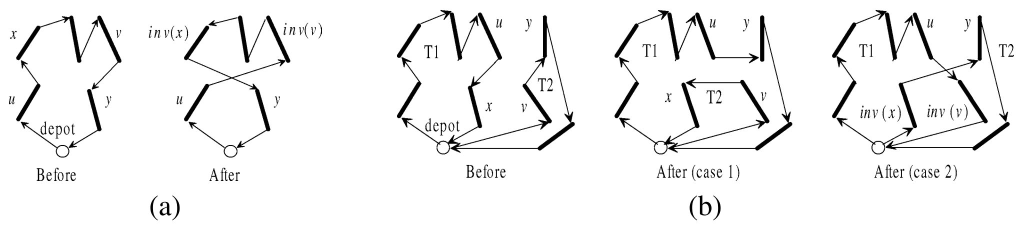


Figure 5. 2-opt moves on one trip (a) and two trips (b). Thick lines correspond to edge-tasks, thin lines to shortest deadheading paths.

The boolean vector *miss* records the tasks missing in C. The algorithm avoids $p = 1$ and $q = \tau$ at the same time, to ensure C $\neq$ P1. The only two lines to be changed for LOX are indicated in comments.

4.4. *Mutation by local search*

In combinatorial optimization, it is well known that the basic GA (Holland, 1975) with simple mutations cannot compete with simulated annealing (SA) and tabu search (TS). To be effective, the generic GA must be hybridized with a local search, giving a *hybrid GA* or *memetic algorithm* (MA) (Moscato, 1999). With a given probability, each child in our MA is converted into an ECARP solution to undergo a local search LS.

LS performs successive *phases* that scan in $O(\tau^2)$ all pairs of tasks (u, v) to try the following moves, in which x (resp. y) is the task serviced after u (resp. v) in the trip of u (resp. v). Each phase ends by performing the first improving move detected or when all pairs (u, v) are examined. LS stops when a phase reports no improvement. The final ECARP solution is converted into a chromosome, as explained in 4.1. Here are the types of moves examined:

N_1: invert task u in its trip if it is an edge-task, i.e., replace u by $inv(u)$ in the trip,

N_2: move task u after task v, or before v if v is the first task of its trip,

N_3: move adjacent tasks (u, x) after task v, or before v if v is the first task of its trip,

N_4: swap tasks u and v,

N_5: two-opt moves (explained in *figure 5*).

Each move type involves *one trip* or *two distinct trips*. Moreover, when moving an edge-task in N_1–N_4, its service direction may be inverted or not. For instance, N_4 comprises in fact four swapping cases: u and v may be replaced by v and u, $inv(v)$ and u, v and $inv(u)$, or $inv(v)$ and $inv(u)$. In N_5, some moves may require the inversion of a substring of tasks (cf. figure 5): they are discarded if the substring contains arc-tasks (not invertible).

4.5. *Population structure and initialization*

The population is implemented as an array Π of nc chromosomes, kept sorted in increasing cost order to ease the selection process described in 4.6. In traditional GAs, identical solutions or *clones* may appear, leading to a premature convergence. The phenomenon worsens in MAs because the local search quickly compresses Π in a reduced cost interval. A possible remedy is to forbid clones. Exact clone detection can be performed efficiently, e.g., using hashing techniques (Cormen, Leiserson, and Rivest, 1990). We adopted an approximate but faster system in which all individuals have *distinct costs*. Let UB be an upper bound on solution costs and *used* a boolean vector, indexed from 0 to UB, such that $used(c) = true$ iff Π contains an individual of cost c. We know in $O(1)$ if a new chromosome S can be added to Π by checking that $used(F(S)) = false$. A crossover is said *unproductive* if its children cannot be kept because of duplicate costs. This concerns a minority of crossovers if nc is not too large (cf. section 5).

Π is initialized with random chromosomes. Because of clones, when nc is too large or the problem very small, many draws may be required to add a new distinct chromosome to the k ones already generated, $k = 1, 2, \ldots, nc - 1$. In practice, we try a fixed number of times mnt to generate the next chromosome and truncate Π to $nc = k$ if all draws fail. It is also possible to include in Π a few good heuristic solutions, for instance computed by EPS, EAM or EUH (cf. section 3). These solutions must be converted into chromosomes, as explained in 4.1.

4.6. *Incremental memetic algorithms*

The basic iteration of an *incremental GA* selects two chromosomes to undergo crossover and mutation. The resulting children immediately replace some existing chromosomes in Π. In a *generational GA* (4.7), the basic iteration (called *generation*) performs a massive reproduction involving all chromosomes. The resulting children are either stored in another population array used for the next generation, or added to Π. In the second case, Π is reduced from $2nc$ to nc individuals at the end of each generation, by eliminating the nc worst solutions.

We designed incremental versions with two types of selection. The first type (Reeves, 1995) selects the rank i of P1 with probability $2(nc - i + 1)/(nc(nc + 1))$. Since Π is sorted in increasing cost order (4.5), the probability of drawing an individual with median cost is roughly $1/nc$, the probability of drawing the fittest Π_1 is doubled $2/(nc + 1)$, while the probability of drawing the worst individual $\Pi(nc)$ is only $2/(nc(nc + 1))$. The rank of P2 is drawn uniformly with a probability $1/nc$. The second type is *binary tournament*. Two chromosomes are randomly selected and the least-cost one is kept for P1. The process is repeated to get P2.

An OX or LOX crossover (4.3) is applied to (P1, P2). One child C is selected at random and undergoes a mutation by local search (4.4) with a given probability. Two replacement strategies were tested: C replaces either the worst individual $\Pi(nc)$ or one $\Pi(k)$ above the median cost, i.e., with $k \geqslant \Pi(\lfloor nc/2 \rfloor)$. Note that both methods preserve the best solution. If no duplicate cost appears, the child mutated or not enters Π and one productive iteration is counted. If not, the child is rejected and the iteration is unproductive.

```
procedure partial_replacement(Π,nc,nrep)
  done := 0 //number of solutions currently replaced
  repeat
    generate a population Ω with nrep distinct costs not present in Π
    sort Ω in increasing cost order
    k := 0
    repeat
      k := k + 1
      if F(Ω(k)) < F(Π(nc)) then
        Π(nc) := Ω(k); done := done + 1; re-sort Π
      else
        cross Ω(k) with each individual of (Π ∪ Ω)\{Ω(k)}
        C := best child with a cost not present in Π
        if F(C) < F(Π(nc)) then
          Π(nc) := C; done := done + 1; re-sort Π
        endif
      endif
    until (done = nrep) or (k = nrep)
  until done = nrep
endproc
```

Algorithm 5. Partial replacement procedure used in restarts.

Our incremental MAs perform a main phase stopped after a given number of productive crossovers, after a given number of productive crossovers without improving Π_1, or when reaching a lower bound *LB* (in that case, Π_1 is of course optimal). More instances are solved by adding a fixed number of short restarts, based on a *partial replacement procedure* (Cheung, Langevin, and Villeneuve, 2001). Each restart stops after a fixed number of crossovers or by reaching *LB*. In section 5, the same number of restarts and the same length per restart are allocated to all instances. Since *LB* is reached in the main phase for a majority of standard instances, restarts are not always used. Section 5 clearly indicates the number of allowed restarts, the number of crossovers allowed per restart, and the numbers of restarts and crossovers actually performed.

Algorithm 5 adapts Cheung's procedure to replace *nrep* chromosomes in our population Π. Each set Ω is built by generating random chromosomes until *nrep* distinct cost values not already present in Π are obtained. This process never fails in practice because nc is small (see 5.2) and *nrep* is even smaller (typically, $nrep = \lfloor nc/5 \rfloor - \lfloor nc/3 \rfloor$). Compared to a blind replacement, the procedure preserves the best solution and never degrades the worst cost. According to Cheung et al., it gives better final solutions for a given CPU time.

4.7. Generational memetic algorithms

We also tested a generational scheme described by Hartmann (1998). Each generation randomly partitions Π into pairs. Each pair undergoes a crossover. All children are added to Π, giving $2nc$ chromosomes, and Π is reduced by keeping the nc best solutions. This method was adapted as follows for populations with distinct costs: the enlarged population is sorted in increasing cost order, and one representative is kept for the nc smallest costs. When several chromosomes have the same cost, better diversity is achieved by keeping the most recent one.

5. Computational evaluation

5.1. Implementation and benchmarks used

All algorithmic components are implemented in the Pascal-like language Delphi 5 and tested on a 1 GHz Pentium-3 PC under Windows 98. The computational evaluation uses three sets of benchmark problems downloadable at `http://www.uv.es/~belengue/carp.html`.

The first set (*gdb* files) contains 25 undirected instances built by DeArmon (1981), with 7–27 nodes and 11–55 edges. Instances 8 and 9 are never used because they contain inconsistencies. The second set (*val* files) contains 34 undirected instances designed by Belenguer and Benavent (2003) to evaluate a cutting plane algorithm. These files have 24–50 nodes and 34–97 edges. In these two first sets, all edges are required: each instance is in fact a UCPP (*Undirected Capacitated Chinese Postman Problem*), a special case of the CARP.

The third set (*egl* files) provides 24 undirected instances built by Belenguer and Benavent (2003). They are called Eglese instances by these authors, because they are based on the road network of the county of Lancashire (UK), used by Eglese (1994) for a winter gritting problem. Belenguer and Benavent have generated 12 files per area, by varying the vehicle capacity W and the percentage of required edges These instances are very interesting for their realism, their large size (77–140 nodes, 98–190 edges), and also because they contain true CARPs and not only UCPPs like in *gdb* and *val* sets.

5.2. Best components, standard setting of parameters and stopping criteria

The best selection of components was determined during an initial testing phase on *gdb* files. We started from a basic incremental MA, with a population of $nc = 50$ random chromosomes without clones, the Reeves selection, the LOX crossover and a local search rate $p_m = 0.02$. Like in 4.6, one child is randomly selected after each crossover to replace one chromosome drawn above the median cost. This MA stops when a lower bound is reached or after 5000 crossovers. The list of experiments and resulting decisions are summarized in table 2.

As pointed out by Barr et al. (1995), an acceptable testing of metaheuristics must distinguish "standard" results, reported for one setting of parameters, and "best results" found using various combinations of parameters. The standard setting is important for comparisons with other methods and to give an idea about performance in operational conditions, e.g., when an executable file with frozen parameters is used or when it is too long to try different settings. Our *standard setting* (table 3) has also been found during the preliminary testing. It is the one giving the best average solution values when applied

Table 2
Experiments for selecting best components.

No	Experiment	Impact on solution costs	Decision
1	Inhibit local search LS	Increase	Keep local search
2	Clones always accepted (*used* ignored)	Increase	Clones never accepted
3	Test various combinations (nc, p_m)	Best one is $nc = 30$, $p_m = 0.1$	Use $nc = 30$, $p_m = 0.1$
4	Switch to a generational MA	Slight increase	Keep incremental MA
5	Binary tournament selection	Slight decrease	Use binary tournament
6	OX crossover	Slight decrease	Use OX
7	Keep 2 children after OX, not one	Slight increase	Keep one child C
8	C replaces worst solution only	Increase	Not adopted
9	C rejected if $F(C) >$ a given cost	Increase + high rejection rate	Not adopted
10	EPS, EAM, EUH in initial Π	Slight decrease	Use EPS, EAM, EUH
11	Apply LS to initial Π	Increase	No LS on initial Π
12	Add restarts	Decrease	Restarts added

Table 3
Standard setting of parameters.

Name	Role	Value
nc	population size	30
UB	largest cost used (dimension of vector *used* defined in 4.5)	50000
mnt	max no. of attempts to get each initial random chromosome	50
p_m	local search rate in main phase	0.1
mnpi	max no. of productive crossovers in main phase	20000
mnwi	max no. of productive crossovers without changing $\Pi(1)$, in main phase	6000
mnrs	max no. of restarts	20
nrep	no. of solutions replaced in each restart (partial replacement procedure)	8
p_r	local search rate in restarts	0.2
rnpi	max no. of productive crossovers per restart	2000
rnwi	max no. of productive crossovers without changing $\Pi(1)$, per restart	2000

to all *gdb* instances. The size of *used* (see 4.5), $UB = 50000$, corresponds to the largest cost found in the initial populations of all instances (around 33000 for some *egl* files), multiplied by a security factor 1.5.

Algorithm 6 illustrates the structure of the best resulting MA and the *stopping criteria*. The procedure *initialize* builds the initial population. The main phase is a call to the procedure *search* (MA basic loop) with a local search rate p_m. This phase ends after *mnpi* productive iterations (crossovers), after *mnwi* non-improving crossovers, or when a lower bound *LB* is reached. The MA stops there if $F(\Pi(1)) = LB$. If not, it executes a restart loop limited to *mnrs* iterations. Each restart calls the replacement procedure of algorithm 5 and the procedure *search*, but this time with the stronger local search rate p_r and the reduced numbers of crossovers *rnpi* and *rnwi*. *Search* and the restart loop may stop at any time by reaching *LB*.

5.3. *Results for gdb files*

Table 4 gathers the results for *gdb* files. Firstly, we describe the table format, shared by the three sets of benchmarks. After the file name, the number of nodes n and the

```
main program
  initialize(Π,nc,used,UB,mnt)                          //initialize population
  if F(Π(1)) > LB then begin                            //if LB not reached
    search(Π,nc,used,LB,pm,mnpi,mnwi)                   //perform main phase
    restarts := 0                                       //initialize restart counter
    while (restarts < mnrs) and (F(Π(1)) > LB) do       //perform restarts
      restarts := restarts + 1                          //count one restart
      partial_replacement(Π,nc,nrep)                    //cf. algorithm 5
      search(Π,nc,used,LB,pr,rnpi,rnwi)                 //intensive short phase
    endwhile
  endif
endmain

procedure initialize(Π,nc,used,UB,mnt)
  for k := 1 to UB do used(k) := false endfor           //cost values used, cf. 4.5
  k := 0                                                //no of chromosomes built
  get solutions of EPS,EAM and EUH as H(1),H(2),H(3)    //heuristics of Section 3
  for i := 1 to 3 do                                    //try to put solutions in Π
    convert H(i) into a chromosome S; split(S)          //reevaluate,see why in 4.1
    if not used(F(S)) then                              //if cost not duplicated
      k := k + 1; Π(k) = S; used(F(S)) := true          //add S to Π
    endif
  endfor
  repeat                                                //generate random solutions
    try := 0                                            //initialize no of attempts
    repeat                                              //loop on attempts
      try := try + 1                                    //count one attempt
      generate S at random; split(S)                    //build a random chromosome
    until (not used(F(S))) or (try = mnt)               //until OK or failed
    if not used(F(S)) then                              //if cost not duplicated
      k := k + 1; Π(k) = S; used(F(S)) := true          //add S to Π
    endif
  until (k = nc) or (used(F(S))                         //Π filled or fail
  if used(F(S)) then nc := k endif                      //actual population size
  sort Π in increasing cost order                       //sort for replacements
endproc

//pls: LS rate, mpi: max.  no of productive Xovers, mwi: idem, without
improvement procedure search(Π,nc,used,LB,pls,mpi,mwi)
  npi := 0                                              //productive crossovers
  nwi := 0                                              //idem, without improvement
  repeat                                                //MA search loop
    select parents P1,P2 by binary tournament           //selection, cf. 4.6
    C := OX(P1,P2)                                      //crossover, cf. 4.3
    split(C)                                            //evaluation (algorithm 2)
    select k at random in [⌊nc/2⌋,nc]                   //Π(k) to be deleted (4.6)
    if random < pls then                                //local search LS required?
      M := LS(C)                                        //apply LS, cf. 4.4
      split(M)                                          //reevaluate, see why in 4.1
      //Π(k) will be replaced in priority by M if M is not a clone,
      //otherwise by the child before its mutation
      if (not used(F(M))) or (F(M) = F(Π(k))) then C := M endif
    endif
    if (not used(F(C))) or (F(C) = F(Π(k))) then        //accept replacement
      npi := npi + 1                                    //count one productive xover
      if F(C) < F(Π(1)) then nwi := 0 else nwi := nwi + 1 endif
      used(F(Π(k))) := false; used(F(C)) := true        //update costs in use
      Π(k) := C                                         //perform replacement
      re-sort Π                                         //keep Π sorted
    endif
  until (npi = mpi) or (nwi = mwi) or (F(Π(1)) = LB)
endproc
```

Algorithm 6. Best MA structure with initialization and search procedure.

Table 4
Computational results for *gdb* files.

File	n	τ	*LB*	Carpet	Time	Best-known	EPS	EAM	EUH	Std MA	Rstrts	Xovers	Time*	Time	Best MA
gdb1	12	22	316	316*	3.15	316* abcd	350	349	330	316*	0	8	0.00	0.00	316*
gdb2	12	26	339	339*	5.17	339* abd	366	370	353	339*	0	2303	0.44	0.44	339*
gdb3	12	22	275	275*	0.07	275* abcd	293	319	297	275*	0	179	0.06	0.06	275*
gdb4	11	19	287	287*	0.09	287* abcd	287*	302	320	287*	0	0	0.00	0.00	287*
gdb5	13	26	377	377*	5.59	377* abd	438	423	407	377*	0	779	0.11	0.11	377*
gdb6	12	22	298	298*	0.85	298* abd	324	340	318	298*	0	1183	0.17	0.17	298*
gdb7	12	22	325	325*	0.00	325* abcd	363	325*	330	325*	0	0	0.05	0.05	325*
gdb10	27	46	344	352	61.00	348 bd	463	393	388	350	20	47187	0.66	39.82	348
gdb11	27	51	303	317	53.91	303* d	354	352	358	303*	1	10708	7.09	7.09	303*
gdb12	12	25	275	275*	1.55	275* abcd	295	300	283	275*	0	236	0.06	0.06	275*
gdb13	22	45	395	395*	2.29	395* abd	447	449	413	395*	0	2000	1.26	1.26	395*
gdb14	13	23	450	458	20.63	458 abd	581	569	537	458	20	46397	0.06	9.78	458
gdb15	10	28	536	544	2.42	538 d	563	560	552	536*	10	24466	7.42	7.42	**536***
gdb16	7	21	100	100*	0.48	100* abcd	114	102	104	100*	0	48	0.05	0.05	100*
gdb17	7	21	58	58*	0.00	58* abcd	60	60	58*	58*	0	0	0.00	0.00	58*
gdb18	8	28	127	127*	1.70	127* abcd	135	129	132	127*	0	81	0.06	0.06	127*
gdb19	8	28	91	91*	0.00	91* abcd	93	91*	93	91*	0	0	0.05	0.05	91*
gdb20	9	36	164	164*	0.28	164* abcd	177	174	172	164*	0	141	0.11	0.11	164*
gdb21	8	11	55	55*	0.20	55* abcd	57	63	63	55*	0	4	0.00	0.00	55*
gdb22	11	22	121	121*	9.50	121* abd	132	129	125	121*	0	1150	0.33	0.33	121*
gdb23	11	33	156	156*	1.13	156* abcd	176	163	162	156*	0	257	0.17	0.17	156*
gdb24	11	44	200	200*	3.38	200* abcd	208	204	207	200*	0	3046	3.35	3.35	200*
gdb25	11	55	233	235	34.37	233* c	251	237	239	233*	10	24567	51.19	51.19	233*
Average				0.48%	9.02	0.14%	10.4%	8.4%	6.4%	0.15%	2.7	7162	3.16	5.29	0.13%
Worst				4.62%	61.00	1.78%	34.6%	24.2%	19.3%	1.78%	20	47187	51.19	51.19	1.78%
Optima				18		20	1	2	1	21					21
Best				19		22	1	2	1	22					23

Lower bounds from Benavent and Belenguer (to appear), except for gdb14 (Amberg and Voß, 2002).

Heuristics cited for published best-known solutions: (a) Belenguer, Benavent, and Cognata (1997), (b) Hertz, Laporte, and Mittaz (2000), (c) Pearn (1989), (d) Lacomme, Prins, and Ramdane-Chérif (2001).

Asterisks denote proven optima, grey cells indicate solutions improved compared to the preliminary GA of Lacomme, Prins, and Ramdane-Chérif (2001), boldface show new best solutions.

Running times in seconds on a 1 GHz Pentium-III PC. Original times for Carpet have been scaled.

Improvement for gdb10 in "Best MA" column obtained by using LOX crossover instead of OX.

number of tasks τ, the 4th column gives the bound obtained by Belenguer and Benavent (2003), except for *gdb14* where it is improved by Amberg and Voß (2002). The two next columns *Carpet* and *Time* show the cost reached with standard parameters by Carpet, the best TS heuristic available for the CARP (Hertz, Laporte, and Mittaz, 2000) and the running time in seconds, scaled for the 1 GHz Pentium-III PC used for the MAs. According to SPEC (2001), the power index for the 195 MHz SGI Indigo-2 workstation used by Carpet is 8.88 for integer computations. SPEC does not report benchmarks beyond 500 MHz for the Pentium-III, but we found 41.7 for a 866 MHz at `http://you.genie.co.uk/peterw/service/compare.htm`, corresponding approximately to 48.2 for 1 GHz. So, we have divided the original Carpet times by $48.2/8.88 = 5.43$.

The best-known solutions before this paper are listed in column *Best-known*. The *EPS*, *EAM* and *EUH* columns report solution costs computed by the extended versions of Path-Scanning, Augment-Merge and Ulusoy's heuristic (cf. section 3). Note that this is the first evaluation of Ulusoy's method on standard benchmarks. Then, the table provides the costs obtained by the MA with standard parameters (*Std MA*), the number of restarts used *Rstrts*, the overall number of productive crossovers *Xovers*, the running time until last improvement *Time**, the overall running time *Time*, and the best cost found using various settings (*Best MA*).

Asterisks denote proven optima, grey cells signal solutions that are improved compared to the GA of Lacomme, Prins, and Ramdane-Chérif (2001), and boldface indicate new best solutions. The last four rows indicate for each column: (a) the average value, given as a deviation to LB in % when the column concerns solution costs (*Average*), (b) the worst value (*Worst*), (c) the number of proven optima (*Optima*) and (d) the number of best-known solutions found (*Best*).

EUH outperforms the other basic heuristics EPS and EAM. The standard MA is at least as good as Carpet in all cases. Compared to Carpet, four instances are improved (10, 11, 15, 25), the average and worst deviations to LB are more than halved and the average running time is 40% smaller. Compared to our first GA (Lacomme, Prins, and Ramdane-Chérif, 2001) needing 21 seconds at 500 MHz on average, the MA runs twice as fast and improves two instances (15, 25). Instance *gdb15* is optimally solved for the first time. Note that these excellent results are achieved without restarts for 18 out of 23 instances (i.e., the lower bound is reached during the main phase). Using several settings (*Best MA*), the MA improves only its solution to *gdb10* but finally finds all best solutions. These results show that *gdb* instances are no longer hard enough for testing CARP metaheuristics.

5.4. *Results for val files*

Table 5 reuses the format of table 4 to present the results for *val* files. The best lower bounds are all obtained by Belenguer and Benavent (2003). Note that the bound 138 they gave to Hertz, Laporte, and Mittaz (2000) for instance *val3c* was not correct, due to an error in their lower bound procedure. The correct value is in fact 137. The *val* files seem empirically harder than *gdb* files: the average deviations to *LB* grow for all

Table 5
Computational results for *val* files.

File	n	τ	LB	Carpet	Time	Best-known	EPS	EAM	EUH	Std MA	Rstrts	Xovers	Time*	Time	Best MA
val1a	24	39	173	173*	0.02	173* abd	186	190	173*	173*	0	0	0.00	0.00	173*
val1b	24	39	173	173*	9.26	173* abd	209	196	197	173*	3	11102	8.02	8.02	173*
val1c	24	39	235	245	93.20	245 bd	331	294	280	245	20	46554	0.27	28.67	245
val2a	24	34	227	227*	0.17	227* abd	259	238	255	227*	0	128	0.05	0.05	227*
val2b	24	34	259	260	13.02	259* abd	284	275	281	259*	0	375	0.22	0.22	259*
val2c	24	34	455	494	31.66	457 bd	516	533	515	457	20	50933	8.08	21.76	457
val3a	24	35	81	81*	0.77	81* abd	88	86	85	81*	0	34	0.05	0.05	81*
val3b	24	35	87	87*	2.79	87* abd	99	96	99	87*	0	5	0.00	0.00	87*
val3c	24	35	137	138	41.66	138 abd	158	152	153	138	20	47291	0.49	28.23	138
val4a	41	69	400	400*	28.32	400* abd	451	443	436	400*	0	235	0.72	0.72	400*
val4b	41	69	412	416	75.66	412* abd	487	487	468	412*	0	503	1.21	1.21	412*
val4c	41	69	428	453	70.06	428* ad	539	483	486	428*	2	10391	19.11	19.11	428*
val4d	41	69	520	556	233.56	530 d	656	631	608	541	20	50746	6.37	103.26	530
val5a	34	65	423	423*	3.80	423* abd	476	466	451	423*	0	908	1.86	1.86	423*
val5b	34	65	446	448	41.40	446* abd	508	487	486	446*	0	471	1.04	1.04	446*
val5c	34	65	469	476	53.27	474 abd	544	509	504	474	20	46197	0.44	101.01	474
val5d	34	65	571	607	224.11	581 d	720	667	660	581	20	56410	11.32	90.74	581
val6a	31	50	223	223*	3.89	223* abd	271	241	243	223*	0	89	0.17	0.17	223*
val6b	31	50	231	241	26.94	233 ad	274	247	253	233	20	53508	6.48	67.34	233
val6c	31	50	311	329	85.18	317 bd	381	365	367	317	20	53548	52.23	52.23	317
val7a	40	66	279	279*	6.59	279* abd	326	306	293	279*	0	927	4.66	1.97	279*
val7b	40	66	283	283*	0.02	283* abd	353	314	295	283*	0	137	0.44	0.44	283*
val7c	40	66	333	343	121.44	334 abd	394	387	381	334	20	51426	60.53	101.17	334
val8a	30	63	386	386*	3.84	386* abd	433	412	432	386*	0	277	0.66	0.66	386*
val8b	30	63	395	401	81.46	395* abd	455	426	439	395*	0	7038	9.95	9.95	395*
val8c	30	63	517	533	147.40	527 d	596	604	603	527	20	51659	62.83	71.46	527
val9a	50	92	323	323*	28.51	323* abd	358	348	345	323*	0	4156	18.29	18.29	323*
val9b	50	92	326	329	59.89	326* abd	352	358	350	326*	0	7832	29.39	29.39	326*
val9c	50	92	332	332*	56.44	332* abd	394	368	368	332*	4	14512	71.19	71.19	332*
val9d	50	92	382	409	353.28	391 d	492	436	462	391	20	55936	76.62	211.13	391
val10a	50	97	428	428*	5.52	428* abd	453	453	452	428*	0	4884	25.48	25.48	428*
val10b	50	97	436	436*	18.43	436* abd	474	460	457	436*	0	961	4.67	4.67	436*
val10c	50	97	446	451	93.47	446* ad	503	478	496	446*	0	4200	17.30	17.30	446*
val10d	50	97	524	544	156.31	530 d	614	590	589	530	20	52710	182.85	215.04	**528**
Average				1.90%	63.87	0.55%	16.8%	11.4%	10.9%	0.61%	7.3	56410	18.61	38.35	0.54%
Worst				8.57%	353.28	4.26%	40.9%	25.1%	20.9%	4.26%	20	20179	182.85	215.04	4.26%
Optima				15		22	0	0	1	22					22
Best				17		33	0	0	1	32					34

Format explained under table 4. "Best MA" column: 530 for val4d found by applying LS to each initial solution, 528 for val10d found by using an exact detection of clones.

algorithms and 15 instances out of 34 require restarts. The average running time is now 38 seconds, but the last improvement is found earlier (compare $18.61/38.35 = 49\%$ for val-files against $3.16/5.29 = 60\%$ for gdb files). Among the constructive heuristics, EUH performs better than EPS and EAM. Again, compared to Carpet, the standard MA provides identical or better solutions, divides by two the average and worst deviations to *LB* and runs 40% faster.

Using several settings, the MA yields all best solutions, improves the preliminary GA of Lacomme, Prins, and Ramdane-Chérif (2001) three times, and finds a new best solution for *val10d*. This solution was obtained by trying an exact clone detection: two chromosomes S_1 and S_2, converted by *Split* into ECARP solutions, are clones if each trip of S_1 can be retrieved in the trips of S_2 with the same sequence of tasks or, for undirected instances, with an inverted sequence. However, the simpler clone detection based on distinct costs was kept because it provides slightly better results on average. The reason probably resides in the integer costs and in the relatively small range of solution values in the instances tested, favouring a better dispersal of solutions.

5.5. *Results for egl files*

Table 6 shows the results for these files constructed by Belenguer and Benavent from Eglese's data. The number of edges $m/2$, often greater than τ, is now mentioned. The *Carpet* column reports unpublished results of Carpet, computed by Mittaz on behalf of Belenguer and Benavent. The running times are unknown. Since Carpet is here the only heuristic compared with the MA, the redundant *Best-known* column is removed.

The *egl* files seem much harder than the previous files: the average deviation to *LB* augments for all algorithms and *LB* is never reached. Of course, the reason is perhaps inherent to the bound and/or to the heuristics. E.g., according to Belenguer and Benavent, the partial graph of required edges is sometimes disconnected and their bound does not exploit this property. All in all, EUH remains the best simple heuristic, the standard MA outperforms Carpet 19 times and the best MA improves all solution values, proving that Carpet finds no optimal solution. The price to pay is a larger average running time (9 min): the instances are bigger and, since *LB* is never reached, the MA performs in all cases its 20 restarts.

5.6. *Makespan minimization*

The flexibility of the memetic algorithm is illustrated here by minimizing a different objective function for *gdb* files: the duration of the longest trip (*makespan*), subject to a limited number of vehicles. The two main changes in the MA are to replace algorithm 2 by algorithm 3 (see 4.2) for *Split* and to use the new objective function in the local search LS. Let q_{tot} be the total demand. The fleet size $K_{\max}$ (see 4.2) is set to the smallest possible value $\lceil q_{\text{tot}}/W \rceil$. This bound is tight for *gdb* files, since it is always reached by the MAs minimizing total cost.

A relatively simple lower bound LB2 to the optimal makespan can be computed as follows. The duration of a trip containing only one task u is $cost(u) = D(\sigma, u) + w(u) +$

Table 6
Computational results for *egl* files.

File	n	$m/2$	τ	*LB*	Carpet	EPS	EAM	EUH	Std MA	Rstrts	Xovers	Time*	Time	Best MA
egl-e1-A	77	98	51	3515	3625	4115	4605	3952	**3548**	20	47325	1.48	74.26	**3548**
egl-e1-B	77	98	51	4436	4532	5228	5494	5054	**4498**	20	46249	48.39	69.48	**4498**
egl-e1-C	77	98	51	5453	5663	7240	6799	6166	**5595**	20	46712	39.98	71.18	**5595**
egl-e2-A	77	98	72	4994	5233	6458	6253	5716	**5018**	20	47290	20.60	152.58	**5018**
egl-e2-B	77	98	72	6249	6422	7964	7923	7080	**6340**	20	57657	22.19	153.41	**6340**
egl-e2-C	77	98	72	8114	8603	10313	10453	9338	8415	20	60000	27.52	129.63	**8395**
egl-e3-A	77	98	87	5869	5907	7454	7350	6723	**5898**	20	54522	24.44	242.00	**5898**
egl-e3-B	77	98	87	7646	7921	9900	9244	8713	7822	20	59860	173.18	255.35	**7816**
egl-e3-C	77	98	87	10019	10805	12672	12556	11641	10433	20	52412	111.50	206.35	**10369**
egl-e4-A	77	98	98	6372	6489	7527	7798	7231	**6461**	20	48101	275.50	291.87	**6461**
egl-e4-B	77	98	98	8809	9216	10946	10543	10223	**9021**	20	60000	291.49	312.85	**9021**
egl-e4-C	77	98	98	11276	11824	13828	13623	13165	**11779**	20	51186	77.83	252.38	**11779**
egl-s1-A	140	190	75	4992	5149	6382	6143	5636	**5018**	20	52115	15.88	208.61	**5018**
egl-s1-B	140	190	75	6201	6641	8631	7992	7086	**6435**	20	54924	21.42	208.77	**6435**
egl-s1-C	140	190	75	8310	8687	10259	10338	9572	**8518**	20	49068	160.38	165.55	**8518**
egl-s2-A	140	190	147	9780	10373	12344	11672	11475	**9995**	20	56695	795.10	874.36	**9995**
egl-s2-B	140	190	147	12886	13495	16386	15178	14845	**13174**	20	60000	641.58	760.50	**13174**
egl-s2-C	140	190	147	16221	17121	20520	19673	19290	16795	20	59606	743.69	746.93	**16715**
egl-s3-A	140	190	159	10025	10541	13041	11957	11956	**10296**	20	50853	651.03	1070.50	**10296**
egl-s3-B	140	190	159	13554	14291	17377	15891	15663	14053	20	57886	1043.58	1064.01	**14028**
egl-s3-C	140	190	159	16969	17789	21071	19971	20064	**17297**	20	60000	622.58	874.30	**17297**
egl-s4-A	140	190	190	12027	13036	15321	14741	13978	**12442**	20	60000	1529.57	1537.59	**12442**
egl-s4-B	140	190	190	15933	16924	19860	19172	18612	**16531**	20	60000	1184.52	1430.26	**16531**
egl-s4-C	140	190	190	20179	21486	25921	24175	23727	**20832**	20	60000	1464.26	1495.02	**20832**
Average					4.74%	26.4%	22.8%	15.4%	2.47%	20	54685	416.15	526.99	2.40%
Worst					8.61%	39.2%	31.0%	19.3%	4.46%	20	60000	1529.57	1537.59	4.46%
Best					0	0	0	0	19					24

All lower bounds from Belenguer and Benavent (2003).
Boldface indicate best-known solutions improved by the MA. Running times in seconds on a 1 GHz Pentium-III PC.
The five improvements in "Best MA" column were obtained by a generational version of the MA.

Table 7
Makespan optimization subject to a limited fleet for *gdb* files.

File	n	τ	LB2	β	q_{tot}/W	γ	EUH	Std MA	Rstrts	Xovers	Time*	Time
gdb1	12	22	64	63	4.40	64	84	66	10	30152	1.43	10.32
gdb2	12	26	59	59	5.20	57	81	60	10	31479	3.13	17.03
gdb3	12	22	59	59	4.40	55	74	60	10	26110	3.24	10.38
gdb4	11	19	72	64	3.80	72	98	74	10	26487	0.11	6.04
gdb5	13	26	64	64	5.20	63	88	69	10	26054	8.85	13.02
gdb6	12	22	64	64	4.40	60	75	68	10	26476	0.22	10.44
gdb7	12	22	65	57	4.40	65	81	68	10	29679	1.31	11.20
gdb8	27	46	38	38	9.22	35	54	40	10	26791	11.43	33.12
gdb9	27	51	37	37	9.56	31	69	39	10	26224	28.67	48.28
gdb10	12	25	69	39	3.70	69	86	73	10	26477	0.22	10.43
gdb11	22	45	79	43	4.48	79	98	82	10	30544	26.47	44.98
gdb12	13	23	93	93	6.06	64	124	96	10	26490	0.11	6.97
gdb13	10	28	128	128	5.98	90	178	140	10	27866	1.37	19.39
gdb14	7	21	20	15	4.24	20	27	21	10	26158	0.16	24.99
gdb15	7	21	15	8	3.03	15	16	15*	0	3626	4.83	4.83
gdb16	8	28	26	14	4.83	26	40	27	10	27132	0.94	22.14
gdb17	8	28	19	9	4.10	19	22	19*	0	148	0.49	0.49
gdb18	9	36	33	19	4.14	33	40	34	10	27915	3.90	46.90
gdb19	8	11	19	17	2.44	19	24	21	10	26026	0.06	25.00
gdb20	11	22	31	20	3.96	31	45	32	10	27860	1.15	16.75
gdb21	11	33	26	15	5.70	26	50	29	10	26264	0.38	30.65
gdb22	11	44	25	12	7.59	25	45	28	10	30963	17.13	107.71
gdb23	11	55	24	13	9.85	24	39	30	10	31342	21.59	117.38
Average							39.0%	5.9%	9.1	25577	5.96	27.76
Worst							92.3%	25.0%	10.0	31479	28.67	117.38

See section 5.6 for comments.

$D(u, \sigma)$, according to equation (3). So, the minimum duration $\delta(u)$ of a trip reduced to one task u is either $cost(u)$, if u is an arc-task, or $\min\{cost(u), cost(inv(u))\}$, if u is an edge-task. A first bound to the makespan is obtained by computing the maximum of these costs for all tasks: $\beta = \max\{\delta(u) \mid u \in A,\ q(u) > 0\}$. A second bound is $\gamma = \lceil LB/K_{\max} \rceil$, where LB is the lower bound for the total cost whose values are listed in table 4. Finally, $\text{LB2} = \max\{\beta, \gamma\}$.

The results are summarized in Table 7. All MA parameters are taken from table 3, except the maximum number of restarts *mnrs*, now set to 10. The only heuristic used for the initial population is the extended Ulusoy's method (EUH, see 3.3). Path-Scanning and Augment-Merge are discarded because they often lead to infeasible solutions. Two optima are found and the average deviation to LB2 is nearly 6%. This gap probably comes from the weakness of the bound: the last improvement is obtained early (5.96 seconds on average) compared to the overall running time (27.76 seconds), indicating that other solutions could be optimal.

Table 8
Comparison between the standard MA and Carpet.

	DeArmon 23 pbs		Benavent 34 pbs		Eglese 24 pbs	
Criterion	Carpet	MA	Carpet	MA	Carpet	MA
Avg. dev. to *LB* %	0.48	0.15	1.90	0.61	4.74	2.47
Max. dev. to *LB* %	4.62	1.78	8.57	4.26	8.61	4.46
No. of proven optima	18	21	15	22	0	0
No. of best solutions	19	22	17	32	0	19
Avg. running time (s)	9.02	5.29	63.87	38.35	unknown	526.99

5.7. *Performance overview*

Table 8 compares performance criteria between the memetic algorithm and Carpet, executed with their respective standard parameters: the average and worst deviations to *LB*, the number of proven optima (when *LB* is reached), the number of best-known solutions retrieved, and the average running time on a 1 GHz Pentium-III PC. The standard setting of parameters seems to be *extremely robust*: it gives the best average results for the three sets of instances and its solutions are improved only 8 times out of 81 by trying different settings. Finally, 26 best-known solutions are improved and all other best-known solutions are retrieved.

The MA confirms the interest of a GA template already applied successfully to the open-shop scheduling problem by Prins (2000). Indeed, this earlier GA shares some common features with our MA for the ECARP: a small population with distinct solutions, a few good solutions in the initial population, an improvement procedure used as mutation operator. This shows that powerful hybrid genetic algorithms can be designed thanks to a synergic effect between several simple improvement ideas.

6. Conclusion

The best memetic algorithm for the CARP presented in this paper outperforms all known heuristics on three sets of benchmarks available in public, even when it is executed with one single setting of parameters. This excellent performance results from a combination of several key-features. In spite of simple chromosomes (without trip delimiters) and crossovers, each child is optimally evaluated thanks to the *Split* procedure and can be strongly improved by local search. Small populations of distinct solutions avoid a possible premature convergence. A few good initial solutions are computed via classical heuristics. The incremental management of population and the partial replacement technique used for restarts accelerate the decrease of the objective function value. The absence of complicated techniques must also be underlined.

Moreover, the MA is already designed for tackling several extensions like mixed networks, parallel arcs and turn penalties. We just checked its correct execution on a few instances constructed by hand from a city map. It is too early to provide a computational evaluation for these extensions: more instances must be prepared, appropriate lower bounds must be developed, while no other algorithm is available for comparison. All these tasks are in progress, in particular a random generator for large scale realistic street networks.

More generally, this paper reflects the increasing research activity on arc routing problems. This activity is confirmed by two very recent articles, published during the last revision of this paper, which bring two other new metaheuristics for solving the CARP: Beullens et al. (2003) describe a guided local search, while Greistorfer (2003) proposes a tabu search procedure that makes use of the scatter search paradigm.

Acknowledgments

The authors wish to thank the two anonymous referees who performed an extremely detailed analysis of the paper and suggested many useful improvements.

References

Amberg, A., W. Domschke, and S. Voß. (2000). “Multiple Center Capacitated Arc Routing Problems: A Tabu Search Algorithm Using Capacitated Trees.” *European Journal of Operational Research* 124, 360–376.

Amberg, A. and S. Voß. (2002). “A Hierarchical Relaxations Lower Bound for the Capacitated Arc Routing Problem.” In R.H. Sprague (Hrsg.), *Proceedings of the 35th Annual Hawaii International Conference on System Sciences, DTIST02*. Piscataway: IEEE, pp. 1–10.

Barr, R.S., B.L. Golden, J.P. Kelly, M.G.C. Resende, and W.R. Stewart Jr. (1995). “Designing and Reporting on Computational Experiments with Heuristic Methods.” *Journal of Heuristics* 1, 9–32.

Belenguer, J.M., E. Benavent, and F. Cognata. (1997). “Un Metaheuristico Para el Problema de Rutas por Arcos con Capacidades.” In *23th National SEIO Meeting*, Valencia, Spain.

Belenguer, J.M. and E. Benavent. (2003). “A Cutting Plane Algorithm for the Capacitated Arc Routing Problem.” *Computers and Operations Research* 30(5), 705–728.

Belenguer, J.M. and E. Benavent. (1998). “The Capacitated Arc Routing Problem: Valid Inequalities and Facets.” *Computational Optimization and Applications* 10, 165–187.

Benavent, E., V. Campos, and A. Corberán. (1992). “The Capacitated Arc Routing Problem: Lower Bounds.” *Networks* 22, 669–690.

Benavent, E. and D. Soler. (1999). “The Directed Rural Postman Problem with Turn Penalties.” *Transportation Science* 33(4), 408–418.

Beullens, P., L. Muyldermans, D. Cattrysse, and D. Van Oudheusden. (2003). “A Guided Local Search Heuristic for the Capacitated Arc Routing Problem.” *European Journal of Operational Research* 147(3), 629–643.

Cheung, B.K.S., A. Langevin, and B. Villeneuve. (2001). “High Performing Evolutionary Techniques for Solving Complex Location Problems in Industrial System Design.” *Journal of Intelligent Manufacturing* 12(5–6), 455–466.

Corberán, A., R. Martí, and A. Romero. (2000). “Heuristics for the Mixed Rural Postman Problem.” *Computers and Operations Research* 27, 183–203.

Cormen, T.H., C.L. Leiserson, and M.L. Rivest. (1990). *Introduction to Algorithms*. MIT Press.

DeArmon, J.S. (1981). “A Comparison of Heuristics for the Capacitated Chinese Postman Problem.” Master’s Thesis, The University of Maryland at College Park, MD, USA.

Eglese, R.W. (1994). “Routing Winter Gritting Vehicles.” *Discrete Applied Mathematics* 48(3), 231–244.

Eglese, R.W. and L.Y.O. Li. (1996). "A Tabu Search Based Heuristic for Arc Routing with a Capacity Constraint and Time Deadline." In I.H. Osman and J.P. Kelly (eds.), *Metaheuristics: Theory and Applications*. Kluwer, pp. 633–650.

French, S. (1982). *Sequencing and Scheduling*. Chichester: Ellis Horwood.

Ghianni, G., G. Improta, and G. Laporte. (2001). "The Capacitated Arc Routing Problem with Intermediate Facilities." *Networks* 37(3), 134–143.

Golden, B.L. and R.T. Wong. (1981). "Capacitated Arc Routing Problems." *Networks* 11, 305–315.

Golden, B.L., J.S. DeArmon, and E.K. Baker. (1983). "Computational Experiments with Algorithms for a Class of Routing Problems." *Computers and Operation Research* 10(1), 47–59.

Greistorfer, P. (2003). "A Tabu Scatter Search Metaheuristic for the Arc Routing Problem." *Computers and Industrial Engineering* 44(2), 249–266.

Hartmann, S. (1998). "A Competitive Genetic Algorithm for Resource-Constrained Project Scheduling." *Naval Research Logistics* 45(7), 733–750.

Hertz, A., G. Laporte, and P. Nanchen-Hugo. (1999). "Improvement Procedures for the Undirected Rural Postman Problem." *INFORMS Journal on Computing* 11(1), 53–62.

Hertz, A., G. Laporte, and M. Mittaz. (2000). "A Tabu Search Heuristic for the Capacitated Arc Routing Problem." *Operations Research* 48(1), 129–135.

Hirabayashi, R., Y. Saruwatari, and N. Nishida. (1992). "Tour Construction Algorithm for the Capacitated Arc Routing Problem." *Asia–Pacific Journal of Operational Research* 9, 155–175.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press.

Lacomme, P., C. Prins, and W. Ramdane-Chérif. (2001). "A Genetic Algorithm for the CARP and its Extensions." In E.J.W. Boers et al. (eds.), *Applications of Evolutionnary Computing*, Lecture Notes in Computer Science, Vol. 2037. Springer, pp. 473–483.

Moscato, P. (1999). "Memetic Algorithms: A Short Introduction." In D. Corne, M. Dorigo, and F. Glover (eds.), *New Ideas in Optimization*. McGraw-Hill, pp. 219–234.

Mourão, M.C. and T. Almeida. (2000). "Lower-Bounding and Heuristic Methods for a Refuse Collection Vehicle Routing Problem." *European Journal of Operational Research* 121, 420–434.

Oliver, I.M., D.J. Smith, and J.R.C. Holland. (1987). "A Study of Permutation Crossover Operators on the Traveling Salesman Problem." In J.J. Grefenstette (ed.), *Proceedings of the 2nd Int. Conf. on Genetic Algorithms*. Hillsdale, NJ: Lawrence Erlbaum, pp. 224–230.

Pearn, W.L. (1989). "Approximate Solutions for the Capacitated Arc Routing Problem." *Computers and Operations Research* 16(6), 589–600.

Pearn, W.L. (1991). "Augment-Insert Algorithms for the Capacitated Arc Routing Problem." *Computers and Operations Research* 18(2), 189–198.

Potvin, J.-Y. and S. Bengio. (1996). "The Vehicle Routing Problem with Time Windows – Part II: Genetic Search." *INFORMS Journal on Computing* 8(2), 165–172.

Prins, C. (2000). "Competitive Genetic Algorithms for the Open-Shop Scheduling Problem." *Mathematical Methods of Operations Research* 51, 540–564.

Reeves, C.R. (1995). "A Genetic Algorithm for Flowshop Sequencing." *Computers and Operations Research* 22(1), 5–13.

SPEC. (2001). "Standard Performance Evaluation Corporation." `http://www.spec.org`

Ulusoy, G. (1985). "The Fleet Size and Mix Problem for Capacitated Arc Routing." *European Journal of Operational Research* 22, 329–337.