Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computers & Operations Research 33 (2006) 3473–3493
computers & operations research
www.elsevier.com/locate/cor

A genetic algorithm for a bi-objective capacitated arc routing problem

P. Lacomme[a], C. Prins[b], M. Sevaux[c,*]

[a] *Université Blaise Pascal, LIMOS, Campus Universitaire des Cézeaux, BP 10125, F-63177 Aubière Cedex, France*
[b] *Université de Technologie de Troyes, LOSI, 12 Rue Marie Curie, BP 2060, F-10010 Troyes Cedex, France*
[c] *Université de Valenciennes, LAMIH/SP, Le Mont Houy, F-59313 Valenciennes Cedex 9, France*

Available online 9 April 2005

Abstract

The capacitated arc routing problem (CARP) is a very hard vehicle routing problem for which the objective—in its classical form—is the minimization of the total cost of the routes. In addition, one can seek to minimize also the cost of the longest trip.

In this paper, a multi-objective genetic algorithm is presented for this more realistic CARP. Inspired by the second version of the Non-dominated sorted genetic algorithm framework, the procedure is improved by using good constructive heuristics to seed the initial population and by including a local search procedure. The new framework and its different flavour is appraised on three sets of classical CARP instances comprising 81 files.

Yet designed for a bi-objective problem, the best versions are competitive with state-of-the-art metaheuristics for the single objective CARP, both in terms of solution quality and computational efficiency: indeed, they retrieve a majority of proven optima and improve two best-known solutions.

Keywords: Capacitated arc routing problem; Multi-objective optimization; Metaheuristic; Genetic algorithm

1. Introduction: The CARP and its bi-objective version

The capacitated arc routing problem (CARP) in its undirected version is defined as follows. Let $G = (V, E)$ be an undirected graph with a set V of n nodes and a set E of m edges. A fleet of identical vehicles of capacity W is based at a depot node s. The number of vehicles is either fixed or left as a decision

* Corresponding author. Tel.: +33 327 51 1326; fax: +33 327 51 1310.
E-mail address: marc.sevaux@univ-valenciennes.fr (M. Sevaux).

0305-0548/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2005.02.017

variable. A subset of τ edges, denoted as *required edges* or *tasks*, require service by a vehicle. All edges can be traversed any number of times. Each edge (i, j) has a traversal cost c_{ij} and a demand q_{ij}. All costs and demands are integers. The goal is to determine a set of vehicle trips (routes) 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 handled by any vehicle does not exceed W.

Since the CARP is $\mathcal{NP}$-hard, large-scale instances must be solved in practice with heuristics. Among fast constructive methods, one can cite for instance Path-Scanning [1], Augment-Merge [2] and Ulusoy's splitting technique [3]. Available metaheuristics include tabu search algorithms [4,5], a tabu search combined with scatter search components [6], a variable neighborhood search [7], a guided local search [8], and hybrid genetic algorithms [9,10]. All these heuristics can be evaluated thanks to tight lower bounds [11,12].

CARP problems are raised by operations on street networks, e.g. urban waste collection, snow plowing, sweeping, gritting, etc. Economically speaking, the most important application certainly is municipal refuse collection. In that case, nodes in G correspond to crossroads, while edges in E model street segments that connect crossroads. The demands are amounts of waste to be collected. The costs correspond either to distances or durations.

The single objective CARP only deals with: minimizing the total cost of the trips. In fact, most waste management companies are also interested in balancing the trips. For instance, in Troyes (France), all trucks leave the depot at 6 am and waste collection must be completed as soon as possible to assign the crews to other tasks, e.g. sorting the waste at a recycling facility. Hence, the company wishes to solve a bi-objective version of the CARP, in which both the total duration of the trips and the duration of the longest trip (corresponding to the *makespan* in scheduling theory) are to be minimized. This bi-objective CARP is investigated in this paper which extends preliminary results presented at the EMO 2003 conference [13]. As the two objectives are conflicting, the fleet size (equal to the number of trips) is not imposed, in order to get smaller makespan values. This is not a problem because fleet size is naturally bounded above by τ (when each trip contains one single task).

For a complete introduction of multi-objective optimization, we refer the reader to a recent annotated bibliography [18] which provides a suitable entry point for general definitions and good references. Concerning references on multi-objective genetic algorithms (MOGAs), see [19,20].

In the last decade, there has been a growing interest in multi-objective vehicle routing, but all published papers deal with node routing problems, e.g. [14–17]. Moreover, apart from the total cost of the trips, they only consider additional objectives such as minimizing time window violations or balancing vehicle loads, but not makespan. To the best of our knowledge, this paper presents the first study devoted to a multi-objective arc routing problem and with one objective related to the makespan. It is organized as follows.

Section 2 recalls a general framework of MOGA, called non-dominated sorting genetic algorithm (NSGA-II), and describes how to adapt it for the bi-objective CARP. The inclusion of a local search procedure is discussed in Section 3. Numerical experiments are conducted in Section 4. Section 5 brings some concluding remarks.

2. A genetic algorithm for the bi-objective CARP

Today, several MOGA frameworks are available in literature and selecting the best one for a given MOOP is not obvious. A recent survey [21] and two comparative studies [22,23] try to provide guide-

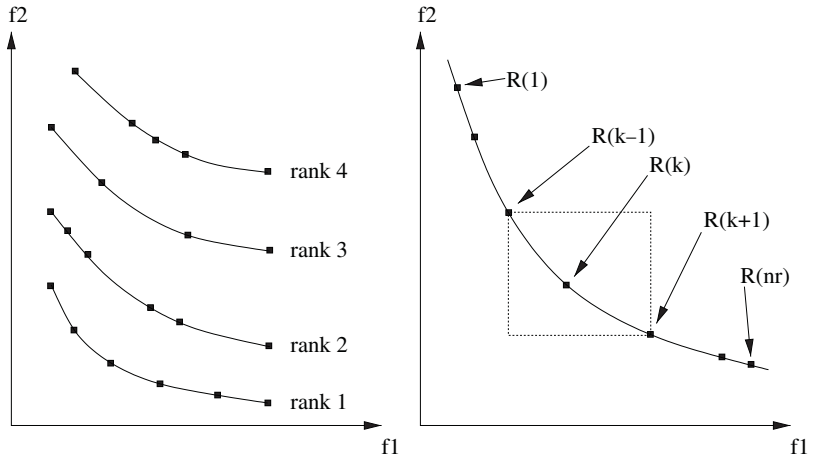

Fig. 1. Non-dominated sorting (left) and crowding distances (right).

lines for selection but these papers consider unconstrained problems, continuous objective functions, and specific sets of benchmarks. It is very difficult to draw conclusions for constrained combinatorial optimization problems.

Our choice for the bi-objective CARP finally turned to a MOGA framework called NSGA-II and designed by Deb [20,24]. The reasons are: (a) its modular and flexible structure, (b) the possibility of upgrading a single-objective GA to NSGA-II (the encoding of chromosomes and crossover operators can be recycled) and (c) its successful applications to a wide range of problems. The first subsection is necessary to introduce the non-trivial framework of NSGA-II, while correcting minor bugs and typos from [20,24]. The adaptation to the bi-objective CARP is described in the second subsection.

2.1. The NSGA-II framework

Non-dominated sorting algorithm: The NSGA-II algorithm computes successive generations on a population of solutions partitioned into *non-dominated fronts*. The non-dominated set is identified and constitutes the non-dominated front of level 1 or front 1. This set is temporarily disregarded from the population. The non-dominated set of the remaining population gives the non-dominated front of level 2, and so on. This process called *non-dominated sorting* is repeated until the partition is complete, see the left part of Fig. 1 for an example.

The method is detailed in Algorithm 1. The population is implemented as a table *pop* of *ns* solutions and $pop(i)$ denotes the ith solution. The partition is a vector *front* of *ns* lists of solutions indices, i.e. $i \in front(k)$ means that solution $pop(i)$ is currently assigned to the front of level k. Dominance between two solutions x and y is computed by the $dom(x, y)$ boolean function. $nb\_better(i)$ counts the solutions which dominate solution $pop(i)$ and $set\_worse(i)$ is the set of solutions dominated by $pop(i)$.

The non-dominated sorting algorithm returns the number of fronts *nf*, the partition $front(1), front(2), \ldots, front(nf)$ and a vector *rank* in which $rank(i)$ denotes the index of the front that stores solution $pop(i)$. A first phase computes in $O(nc \cdot ns^2)$ *nb_better* and *set_worse* values, a second one

in $O(ns^2)$ assigns solutions to their respective fronts and computes *nf* and *rank*. The overall complexity is then $O(nc \cdot ns^2)$.

Algorithm 1 procedure *non_dominated_sort*(*pop*,*ns*,*front*,*nf*,*rank*)

```
1:  front(1) ← ∅
2:  for i ← 1 to ns do
3:      nb_better(i) ← 0
4:      set_worse(i) ← ∅
5:      for j ← 1 to ns do
6:          if dom(pop(i),pop(j)) then
7:              add j to set_worse(i)
8:          else if dom(pop(j),pop(i)) then
9:              nb_better(i) ← nb_better(i)+1
10:         end if
11:     end for
12:     if nb_better(i) = 0 then
13:         add i to front(1)
14:     end if
15: end for
16: nf ← 1
17: loop
18:     for all i in front(nf) do
19:         rank(i) ← nf
20:         for all j in set_worse(i) do
21:             nb_better(j) ← nb_better(j)-1
22:             if nb_better(j) = 0 then
23:                 add j to front(nf+1)
24:             end if
25:         end for
26:     end for
27:     exit when front(nf+1) = ∅
28:     nf ← nf+1
29: end loop
```

Crowded tournament operator: In NSGA-II, each solution is assigned a fitness equal to its non-domination level (1 being the best level). Thus, minimization of the fitness is assumed. Each generation generates new solutions by selecting parent-solutions and applying crossover and mutation operators. Each parent is selected using *binary tournament*: two solutions are randomly selected from the population and the fittest one (with the smallest rank) is kept.

This classical selection technique is modified as follows in NSGA-II: when two solutions belong to the same front, the tournament prefers the most isolated one, using a *crowding distance* described below. The crowding distance $margin(i)$ of a solution $pop(i)$ is a kind of measure of the search space around it

which is not occupied by any other solution in the population. The winner of a tournament between two solutions $pop(i)$ and $pop(j)$ can then be computed thanks to a *crowded tournament operator*, implemented as a boolean function $better(i, j)$ which returns *true* if $rank(i) < rank(j)$ or $(rank(i) = rank(j))$ and $(margin(i) > margin(j))$. The idea is to favor the best solutions, while keeping the fronts well scattered to prevent clusters of solutions.

To simplify, consider $nc = 2$ criteria like in the bi-objective CARP and a front R of nr solutions. Let $f_c^{\min}$ and $f_c^{\max}$ be the minimum and maximum values of criterion f_c in R, $c = 1, 2$. Sort R in increasing values of the first criterion and let $R(k)$ be the kth solution in the sorted front. The crowding distance of $R(k)$ is defined as follows for $1 < k < nr$:

$$margin(R(k)) = \frac{f_1(R(k)+1) - f_1(R(k)-1)}{f_1^{\max} - f_1^{\min}} + \frac{f_2(R(k)-1) - f_2(R(k)+1)}{f_2^{\max} - f_2^{\min}}. \quad (1)$$

For $k = 1$ or $k = nr$, $margin(R(k)) = \infty$. The goal of this convention is to favor the two extreme points of the front, to try to enlarge it. The right part of Fig. 1 depicts the computation of margins. The margin of $R(k)$ is nothing but half of the perimeter of the rectangle indicated by dashed lines.

Generation of new solutions: The computation of new solutions at each iteration of NSGA-II can be implemented as a procedure *add_children*, specified by Algorithm 2. The vector *rank* computed by the non_dominated sorting procedure and the vector of crowding distances *margin* are required for the crowded tournament operator. The procedure creates *ns* offsprings which are added at the end of *pop*, thus doubling its size. Any crossover or mutation operator can be used, depending on the problem at hand.

Algorithm 2 procedure *add_children*(*pop*,*ns*,*rank*,*margin*)

1: *input_ns* ← *ns*
2: **for** *ns* ← *input_ns* to 2×*input_ns* **do**
3: draw two distinct parents P1, P2 using the crowded tournament operator
4: combine P1 and P2, using a crossover operator, to get one new solution *S*
5: mutate *S* with a fixed probability
6: *pop*(*ns*) ← *S*
7: **end for**

NSGA-II overall structure: The general structure of NSGA-II is given in Algorithm 3. An initial population *pop* of *ns* random solutions is built by the *first_pop* procedure and sorted by non-domination, using Algorithm 1. The procedure *get_margins* computes for each solution $pop(i)$ its crowding distance $margin(i)$. Then, each iteration of the main loop starts by calling *add_children*, to create *ns* children which are added at the end of *pop*, see Algorithm 2. Finally, the resulting population with $2 \cdot ns$ solutions is reduced to a new population *newpop* by keeping the *ns* best solutions. To do this, fronts and margins must be updated, using *non_dominated_sort* and *get_margins*. Starting from the front of level 1, complete fronts are then transferred to *newpop* as long as possible. The first front $front(i)$ which could not be accomodated fully is truncated by keeping its most widely spread solutions. This is achieved by arranging its solutions in descending order of the crowding distance values, thanks to the *margin_sort* procedure, and by copying the best

solutions until *newpop* contains exactly *ns* solutions. Finally, *pop* receives the contents of *newpop* for the next iteration of the GA.

Algorithm 3 procedure *nsga*2()

```
1:  first_pop(pop,ns)
2:  non_dominated_sort(pop,ns,front,nf,rank)
3:  get_margins(pop,ns,front,nf,margin)
4:  repeat
5:      add_children(pop,ns,rank,margin)
6:      non_dominated_sort(pop,ns,front,nf,rank)
7:      get_margins(pop,ns,front,nf,margin)
8:      newpop ← ∅
9:      i ← 1
10:     while |newpop|+|front(i)| ⩽ ns do
11:         add front(i) to newpop
12:         i ← i+1
13:     end while
14:     missing ← ns-|newpop|
15:     if missing ≠ 0 then
16:         margin_sort(front,i,margin)
17:         for j ← 1 to missing do
18:             add the jth solution of front(i) to newpop
19:         end for
20:     end if
21:     pop ← newpop
22: until stopping_criterion
```

2.2. GA components for the bi-objective CARP

This section describes the required components to instanciate the NSGA-II framework for our bi-objective CARP. The representation of solutions as chromosomes and their evaluation come from a GA [10] which is currently one of the most effective solution methods for the single objective CARP. Therefore, only a short description is provided here: the reader is invited to refer to [9] or [10] for more details.

2.2.1. Chromosome representation and evaluation

A chromosome is an ordered list of the τ tasks, in which each task may appear as one of its two directions. Implicit shortest paths are assumed between successive tasks. The chromosome does not include trip delimiters and can be viewed as a giant tour for a vehicle with infinite capacity. An $O(\tau^2)$ procedure *Split* described in [10] is used to derive a least total cost CARP solution (subject to the given sequence), by splitting the chromosome (giant tour) into capacity-feasible tours. This technique is based on the computation of a shortest path in an auxiliary graph, in which each arc models one possible

feasible tour that can be extracted from the giant tour. The makespan (second objective) corresponds to the maximum duration of the trips computed by *Split*.

In practice, each chromosome is stored in *pop* as a record with three fields: the sequence of tasks and the values computed by *Split* for the total cost and the makespan. The associated solution with its detailed trips is not stored because it is required only at the end of the GA, to print the results. It can be extracted at that time by calling *Split* again.

Following GA terminology, each of our chromosomes represents the *genotype* of a solution, i.e. one abstraction of a solution instead of the solution itself. The actual solution (*phenotype*) is materialized by a evaluation/decoding procedure, here *Split*. The reader may be surprised by the priority given by *Split* to the total cost over the makespan. Indeed, it is possible to adapt *Split* to minimize makespan [10] or even to minimize total cost, subject to an upper bound on the makespan. In fact, all these evaluations are pertinent because they return a non-dominated solution among the possible interpretations of the same chromosome. An example of non-pertinent evaluation is to scan the chromosome, starting from the first task, and to create a new trip each time the current task does not fit vehicle capacity. Clearly, such an evaluation "wastes" the chromosome because it is dominated by *Split*.

A version of *Split* that minimizes total cost was adopted because no good rule is available to choose amongst pertinent evaluations. Giving priority in that way to the total cost has no negative impact on the overall performance of the GA. Indeed, the aim of a decoding procedure in general is to define a mapping from the set of chromosomes into the set of solutions and to assign reproducible values to the two objectives, but not to optimize in some way. Optimizing is the role of the search mechanism of the GA, with its selections that favor the fittest parents for reproduction.

2.2.2. *Initial population and clone management*

Most published MOGAs start from an initial population *pop* made of *ns* random chromosomes, but including a few good solutions always accelerates the convergence. The procedure *first_pop* in Algorithm 3 uses three such solutions computed by classical CARP heuristics, namely Path-Scanning [1], Augment-Merge [2] and Ulusoy's heuristic [3]. Each heuristic solution is converted into a chromosome by concatenating its trips. The chromosome is then re-evaluated by *Split* because the resulting solution sometimes dominates the heuristic solution.

Although this is not part of the NSGA-II framework, a sufficient condition is used to prevent identical solutions (*clones*): in the initial population and in the subsequent iterations of the GA, there can be at most one solution with two given values of the objectives, i.e. two solutions cannot occupy the same point in the graphical representation of the fronts (remember that all costs are integers). This technique was included to avoid a progressive colonization of the population by clones (typically 20–30% after 100 iterations).

2.2.3. *Crossover operators*

The advantage of chromosomes without trip delimiters is to allow the use of classical crossovers designed for permutation chromosomes, like the *Linear Order Crossover* (LOX) and the *Order Crossover* (OX). Like the single-objective CARP studied in [10], the OX crossover gave the best results after some preliminary testing. This crossover must be slightly adapted because the chromosomes are not exactly permutations: each required edge may appear as one of its two directions. Implementation details can be found in [10].

Hence, the crossover used in the *add_children* procedure of Algorithm 2 is in fact OX. One of the two children is selected at random and is evaluated by *Split*. It is added to *pop* only if no solution exists with the same objective values, else it is discarded. Because of such rejections, more than *ns* crossovers may be required to double the size of *pop*: the FOR loop of Algorithm 2 must be replaced by a REPEAT ... UNTIL $ns = 2 \times input\_ns$.

2.2.4. Stopping conditions

There is no standard technique to define reliable stopping conditions for stopping a MOGA. In combinatorial optimization and in the single-objective case, metaheuristics can be stopped after a fixed number of iterations without improving the current best solution, or by using a variance criterion. The ways of extending these criteria to several objectives still raise endless discussions in the MOO community. The underlying problem, discussed in the numerical experiments of Section 4, is to define a pertinent comparison between the final sets of efficient solutions computed by two different algorithms on the same instance. For this study, the MOGA is simply stopped after a fixed number of iterations. More sophisticated stopping criteria had no noticeable effect on the final results.

3. Local search procedures

In single-objective optimization, it is well known that a standard GA must be hybridized with a local search procedure to be able to compete with state-of-the-art metaheuristics like tabu search. Such GAs, called *memetic algorithms* by Moscato [25], have raised a growing interest in the last decade: for instance, a dedicated conference (WOMA) is now entirely devoted to them. Hybridization is still rarely used in multi-objective optimization, and only a few authors like Jaszkiewicz [26] start investigating this area. Several difficulties are raised, for instance: (a) the notion of local search must be clarified when several objectives must be optimized, (b) several locations are possible in the existing MOGA frameworks for calling a local search, and (c) the local search must not disturb the search mechanism of the MOGA, for instance by concentrating the solutions into small clusters. This section describes the moves and the general structure of our local search procedure, the criteria used to accept a move, and the possible ways of integrating local search in the MOGA.

3.1. Moves tested and general structure

Our local search procedures work on the individual routes, not on the sequence of tasks defined by a chromosome. Hence, the input chromosome is first converted into a set of routes. The moves include the removal of one or two consecutive tasks from a route, with reinsertion at another position, and 2-opt moves. All moves may involve one or two routes and the two traversal directions of an edge are tested in the reinsertions. Each iteration of the local search scans all these moves in $O(\tau^2)$ to find the *first* improving move (the criteria for deciding if there is an improvement are discussed in the next subsection). The whole local search stops when no more improvement can be found. The trips are then concatenated into a chromosome (giant tour), which is reevaluated by *Split*.

3.2. Acceptance criteria for a move

Four acceptance criteria $accept(S, S')$ were tested to accept a move that transforms the incumbent solution S into a neighbor solution S'. This has lead to four local search procedures called LS1 to LS4. In what follows, $f_1(S)$ and $f_2(S)$, respectively, denote the total cost and the makespan of solution S and w_1 is a real number in [0, 1].

*LS*1: $accept(S, S') = f_1(S') - f_1(S) < 0$;

*LS*2: $accept(S, S') = f_2(S') - f_2(S) < 0$;

*LS*3: $accept(S, S') = ((f_1(S') - f_1(S) \leqslant 0)$ and $(f_2(S') - f_2(S) < 0))$ or $((f_1(S') - f_1(S) < 0)$ and $(f_2(S') - f_2(S) \leqslant 0))$;

*LS*4: $accept(S, S') = w_1 \cdot (f_1(S') - f_1(S)) + (1 - w_1) \cdot (f_2(S') - f_2(S)) < 0$.

LS1 performs its descent on the total cost, LS2 on the makespan, while LS3 accepts the new solution if and only if it *dominates* the current one. The last local search is a weighted sum method which scalarizes the two objectives into a single objective, by multiplying each objective by a user-supplied weight. In these four versions, no normalization of the objectives is required because the total duration of the trips and the makespan are commensurable and of the same order of magnitude.

3.3. Location of local search in the MOGA

The easiest way to include a local search procedure in the MOGA probably is to replace the mutation in Algorithm 2. The mutation can be suppressed because diversification is inherent to the NSGA-II framework, thanks to crowding distances. However, a systematic local search plays against a good dispersal of solutions in the population. This is why, in practice, an offspring undergoes a non-systematic local search with a fixed probability p_{LS}, typically 10%. This location of the local search was the only one tested for LS1, LS2 and LS3. It is called *local search on children* in the numerical experiments of Section 4.

In LS4, the weight w_1 that determines the descent direction must be computed. A random selection as suggested in [27] gives poor results. Murata et al. [28] have designed a MOGA (not based on NSGA-II) for a pattern classification problem, in which a local search like LS4 is applied to all solutions of front 1, with a descent direction which depends on the location of solutions on this front, see Fig. 2. The aim is to improve the front, with emphasis on the two extreme solutions, while preserving the spacing between solutions. To compute w_1 for a given solution S, these authors use Eq. (2) in which $f_c^{\min}$ and $f_c^{\max}$, respectively, denote the minimum and maximum values of criterion f_c, $c = 1, 2$.

$$w_1 = \left(\frac{f_1(S) - f_1^{\min}}{f_1^{\max} - f_1^{\min}} \right) \Bigg/ \left(\frac{f_1(S) - f_1^{\min}}{f_1^{\max} - f_1^{\min}} + \frac{f_2(S) - f_2^{\min}}{f_2^{\max} - f_2^{\min}} \right). \tag{2}$$

For the instances tested in Section 4 for the bi-objective CARP, the first front contains on average less than $\frac{1}{10}$ of the solutions and applying LS4 to the first front only is not agressive enough. However, by computing $f_1^{\min}$, $f_1^{\max}$, $f_2^{\min}$ and $f_2^{\max}$ on the whole population instead of the first front, LS4 can be applied to any solution and with a descent direction that depends on its position in *pop*. This option was finally selected for our MOGA.

Two possible locations can be used for LS4. As for LS1, LS2 and LS3, it can be called in *add_children* with a given probability p_{LS}, in lieu of mutation (*local search on children*). In that case, $f_1^{\min}$, $f_1^{\max}$,

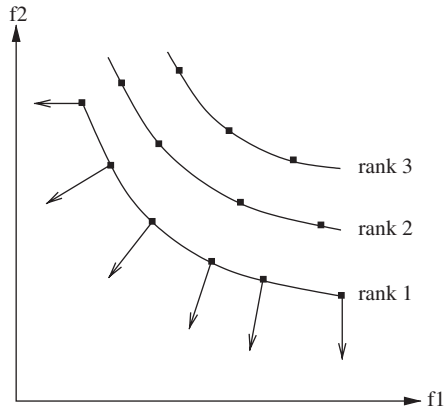

Fig. 2. Variable descent directions in Murata's MOGA.

$f_2^{\min}$ and $f_2^{\max}$ are computed on the whole population, at the beginning of the procedure, and the descent direction for child S is computed using Eq. (2).

The other location consists in applying periodically LS4 to a fraction of the fronts, just after the non-dominated sort in Algorithm 3. Since this local search phase may change the fronts, it must be followed by a second call to *non_dominated_sort*. To keep approximately an overall local search rate of 10%, LS4 is applied every k iterations of the MOGA to $k \times ns/10$ solutions of *pop*, for instance 10% of solutions every iteration or 20% every two iterations. In fact, better results were obtained by applying LS4 to all solutions every $k = 10$ iterations, probably because of a better oscillation between diversification and intensification. We call this way of including LS4 *periodic local search*.

4. Computational experiments

All developments were done in Delphi 7 (Borland®) and tested on a Pentium IV laptop computer clocked at 1.8 GHz under Windows 2000. The following subsections describe the instances used, the MOGA versions selected, the evaluation criteria, the results, and two graphical examples.

4.1. Instances tested

The computational evaluation is based on three sets of standard CARP instances which are used by most authors working on the single-objective CARP [4–12]. The first set corresponds to the 23 *gdb* files from Golden, DeArmon and Baker [1], with 7–27 nodes and 11–55 edges, all required. The original set contains 25 files but two instances (*gdb*8 and *gdb*9) are inconsistent and never used. The second set gathers the 34 *val* files proposed by Belenguer and Benavent in [12], which have 24–50 nodes and 34–97 edges, all required. The last set provides the 24 *egl* instances generated by Belenguer and Benavent [12], by perturbing the demands in three rural networks supplied by Eglese. These *egl* files are bigger (77–140 nodes, 98–190 edges) and in some instances not all edges are required. All these files can be downloaded from the internet, at address http://www.uv.es/~belengue/carp.html.

Table 1
MOGA versions tested

Version	Iterations	LS type	LS location
MO1	100	None	Irrelevant
MO2	100	1	On children, rate 10%
MO3	100	2	On children, rate 10%
MO4	100	3	On children, rate 10%
MO5	200	3	On children, rate 10%
MO6	100	4	On children, rate 10%
MO7	200	4	On children, rate 10%
MO8	100	4	Periodic, every 10 iterations
MO9	200	4	Periodic, every 10 iterations

4.2. Test protocol and parameters

A preliminary testing phase was required to fix the population size, the crossover (OX or LOX), the number of iterations and to evaluate the impact of good heuristic solutions in the initial population. This has lead to the following features which are shared by all MOGA versions in the sequel: a population of $ns=60$ solutions with three good initial solutions, the OX crossover and 100–200 iterations corresponding to 6000 to 12,000 crossovers. Path-Scanning [1], Augment-Merge [2] and Ulusoy's heuristic [3] were selected to provide good initial solutions. They can be discarded without changing the quality of final solutions, but in that case the MOGA converges more slowly, in 300–400 iterations.

A basic version of the MOGA without local search, called MO1, was selected as a reference algorithm. Eight other versions listed in Table 1 were prepared for comparison. They differ from MO1 by the number of iterations and the kind of local search.

4.3. Evaluation criteria

Comparing two different algorithms for solving the same problem is a central issue in multi-objective optimization, especially when the final choices of the decision maker are unknown. Several criteria were published, either to evaluate the quality of front 1 at the end of one given algorithm, or to provide a relative comparison between the final fronts computed by two algorithms.

This study uses a simple measure proposed by Riise [29] to compare two fronts. Given a front F to be evaluated and a reference front R (defined by MO1 in our case), a kind of distance is measured between each solution of F and its projection onto the extrapolated reference front. This extrapolated front is a piecewise continuous line (in the bi-objective case) going through each solution and extended beyond the two extreme solutions, like in Fig. 3. These distances are multiplied by -1 when the solution to compare is below the extrapolated front, like d_4 in the figure. The proposed measure $\mu(F, R)$ is the sum of the signed distances between the solutions of F and their respective projections on the reference front, e.g. $\mu = d_1 + d_2 + d_3 - d_4$ in the figure.

Hence, for one given instance, a version of the MOGA with a final front F outperforms MO1 with its final front R if $\mu(F, R) < 0$. Since μ depends on the number of solutions in F, a normalized measure $\bar{\mu}(F, R)=\mu(F, R)/|F|$ will be also used. The measure μ is a bit arbitrary and of course subject to criticism.

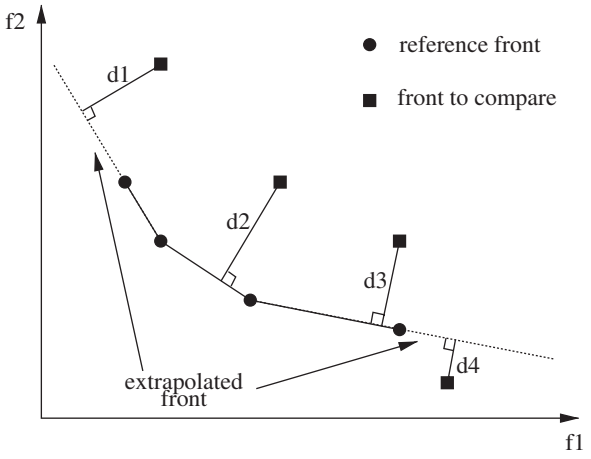

Fig. 3. Examples of distances to the extrapolated front.

For example, how can we measure the difference between two fronts, when one is well spread while the other displays several gaps between clusters of solutions? Nevertheless, its merit is to exist and to help us rank the tested protocols. Because of no a priori knowledge of the solutions which lie on the efficient front, such measures should be completed by a visual inspection.

Very recently, some authors coming from the combinatorial optimization community like Jaszkiewicz [27] have claimed that the authors of multi-objective metaheuristics should prove to be competitive to single objective metaheuristics in terms of quality of solutions and computational efficiency. Since several metaheuristics are available for the single objective CARP [4–10], they can be used to evaluate the total cost of the leftmost solution of front 1, thus completing the measures μ and $\bar{\mu}$. Two metaheuristics were selected for that purpose in our computational evaluation: CARPET, the most efficient tabu search algorithm, designed by Hertz et al. [5], and the memetic algorithm (MA) proposed by Lacomme et al. [10].

Our tables of results compare also the leftmost and rightmost solutions with two lower bounds. The lower bound LB_1 for the total cost is the one proposed by Belenguer and Benavent [12]. Concerning makespan, a trivial lower bound LB_2 corresponds to the longest trip, when each trip performs one single task: $LB_2 = \max\{D_{si} + c_{ij} + D_{js} | [i, j] \in E\}$, where s is the depot node and D_{ij} the cost of a shortest path from node i to node j in G.

4.4. Results

4.4.1. Format of tables

For each set of benchmarks, the next subsections present one synthetic table with the results obtained by CARPET, the MA and each MOGA version, see Table 2 for instance. This table indicates for each algorithm the following *average* values: the number of solutions in the final set of non-dominated solutions F, the deviation $Dev\ LB_1$ of the leftmost solution to the lower bound on the total cost LB_1, the number of times LB_1 *hits* this bound is reached (number of proven optima), the deviation $Dev\ LB_2$ of the rightmost solution to the makespan lower bound LB_2, the number of proven optimal makespans LB_2 *hits*, the

Table 2
Average results for the 23 *gdb* files

Method	$\|F\|$	Dev LB_1	LB_1 hits	Dev LB_2	LB_2 hits	μ	$\bar{\mu}$	Time (s)
CARPET	—	0.48	18	—	—	—	—	5.01
MA	—	0.15	21	—	—	—	—	2.94
MO1	3.87	3.81	5	29.64	9	0.00	0.00	1.94
MO2	3.52	0.47	17	30.43	7	−18.15	−4.61	6.01
MO3	3.78	3.40	5	24.97	11	−5.56	−1.74	4.56
MO4	3.26	0.56	16	22.07	11	−22.82	−6.48	6.53
MO5	3.30	0.56	16	20.28	11	−23.64	−6.45	11.53
MO6	3.39	0.56	16	22.00	10	−25.14	−6.39	6.46
MO7	3.43	**0.48**	16	21.33	11	−24.07	−6.35	12.94
MO8	3.39	**0.39**	**18**	21.18	12	−23.94	−6.45	7.17
MO9	3.43	**0.36**	**18**	20.95	12	−23.49	−6.53	13.79

measure μ for a reference front provided by MO1, the normalized measure $\bar{\mu}$ and the CPU time in seconds on a 1.8 GHz Pentium-IV PC. For CARPET and the MA, the only applicable columns are *Dev* LB_1, LB_1 *hits* and the running time. These fields are in boldface when a MOGA is at least as good as a single objective metaheuristic.

For each synthetic table, the results of the best MOGA version in terms of $\bar{\mu}$ are listed instance by instance in Tables 5–7. The three first columns indicate the instance name, the number of nodes n and the number of edges m. The number of tasks τ is cited only for *egl* files, because it is always equal to the number of edges in the other benchmarks. These columns are followed by the total cost and the running time achieved by CARPET (listed in [5], except the *egl* files whose solution values are given in [12], but without running times) and by the memetic algorithm [10].

In order to have a fair comparison with the MOGAs, the solution values are given for the standard setting of parameters of these metaheuristics and the running times are scaled for the 1.8 GHz PC used in this study. The MA in [10] was executed on a 1 GHz Pentium-III PC and its running times are here multiplied by 0.555. CARPET was tested on a 195 MHz Silicon Graphics Indigo-2 workstation which is 5.43 times slower than the Pentium-III. Hence, the execution times provided by Hertz et al. are multiplied by $0.555/5.43 = 0.1022$ in our study. The best-known solution values (total cost) found by CARPET and the MA with various settings of parameters are given in the *BKS* column.

The eight remaining columns concern the MOGA: the number of efficient solutions $|F|$, the total costs and makespans for the leftmost and rightmost solutions (f_1^{left}, f_2^{left}, f_1^{right} and f_2^{right}), the running time in seconds and the measures μ and $\bar{\mu}$. Each detailed table ends with three lines that provide for each column its average value (deviation to a bound, running time or measure), the worst value and the number of proven optima.

4.4.2. Results for gdb files

They are given in Table 2. All versions with a local search (MO2 to MO9) are much better than MO1. As expected, MO2 with its local search LS1 on the total cost yields a good leftmost point but a poor rightmost point, while the contrary holds for MO3 with its LS2 on makespan. MO4 with its local search LS3 (accepting a move if the result dominates the incumbent solution) is almost as good as CARPET but

Table 3
Average results for the 34 *val* files

Method	$\|F\|$	Dev LB_1	LB_1 hits	Dev LB_2	LB_2 hits	μ	$\bar{\mu}$	Time (s)
CARPET	—	1.90	15	—	—	—	—	35.48
MA	—	0.61	22	—	—	—	—	21.31
MO1	5.91	8.52	1	37.73	5	0.00	0.00	3.21
MO2	4.76	**1.23**	**17**	32.06	5	−85.43	−16.18	27.80
MO3	6.06	7.99	3	30.90	7	−7.33	−2.31	11.74
MO4	4.88	**1.36**	**15**	21.65	8	−109.05	−20.87	32.50
MO5	5.29	**1.01**	**19**	20.59	9	−125.98	−21.37	65.20
MO6	5.00	**1.16**	**15**	21.50	10	−113.07	−21.30	38.20
MO7	5.24	**0.99**	**18**	19.62	10	−121.70	−21.85	76.17
MO8	5.09	**1.33**	**16**	20.19	10	−114.11	−20.61	44.53
MO9	5.32	**1.13**	**17**	19.43	10	−124.03	−21.51	83.06

performing 200 iterations instead of 100 (MO5) brings no improvement. The versions which periodically apply the directional local search LS4 to all solutions (MO8 and 9), respectively, outperform MO6 and MO7 in which LS4 is called on children. The best versions for 100 and 200 iterations are, respectively, MO8 and MO9, they have little effect on the two measures but improve the two extreme solutions. They outperform CARPET for the average deviation to LB_1 but do not find more optima and are a bit slower.

The results of the best version MO9 are detailed in Table 5. They show that MO9 is very robust, since its worst deviation to LB_1 is 2.23% vs. 4.62% for CARPET. No total cost obtained by MO9 is improved by the other versions. In particular, the two instances *gdb*10 and *gdb*14 remain open.

4.4.3. Results for val files

The synthetic results of Table 3 show the same hierarchy as for *gdb* files: the MOGAs with LS4 outperform the MOGAs with LS3, which in turn outperform the versions with single objective local searches and the version without local search. But, this time, LS4 gives better results when it is called on children (MO6–MO7). The convergence seems slower, since noticeable improvements can be found by allowing 200 iterations. The average running time strongly increases but is still reasonable (1.3 min maximum). It is mainly consumed by the calls to local searches. All MOGAs with LS3 or LS4 (MO4 to MO9) outperform CARPET for the average deviation to LB_1 (MO4 is even a bit faster) but 200 iterations are required to find more optima.

Roughly speaking, the best version MO7 halves the average deviation to LB_1 obtained by CARPET, at the expense of a double running time. Its results are detailed for each instance in Table 6. In addition to the 18 optimal total costs listed in this table, the other versions have found optima for the following instances: *val*2*b* (259), *val*2*c* (457) and *val*10*c* (446). Moreover, they improve MO7 for *val*4*d* (536 vs. 539), *val*5*d* (593 vs. 595) and *val*9*d* (397 vs. 399). Unfortunately, no open instance is broken and best-known solutions are not improved.

4.4.4. Results for egl files

The lower bound LB_1 proposed by Belenguer and Benavent is never tight on the *egl* files. This is why the number of optimal total costs is replaced in Table 4 by the number of best-known solutions retrieved

Table 4
Average results for the 24 *egl* files

Method	$\|F\|$	Dev LB_1	BKS	Dev LB_2	LB_2 hits	μ	$\bar{\mu}$	Time (s)
CARPET	—	4.74	0	—	—	—	—	Unknown
MA	—	2.47	19	—	—	—	—	292.77
MOGA1	5.83	13.24	0(0)	1.87	15	0.00	0.00	2.77
MOGA2	5.29	**4.24**	**1(0)**	0.94	16	−1234.01	−256.65	158.61
MOGA3	6.67	13.32	0(0)	0.17	20	−63.93	−15.98	35.80
MOGA4	5.21	**4.47**	0(0)	0.27	19	−1333.52	−335.11	148.50
MOGA5	6.00	**3.84**	**1(0)**	0.10	21	−1586.62	−360.97	263.49
MOGA6	5.33	**4.47**	**1(0)**	0.08	22	−1435.73	−324.27	153.48
MOGA7	5.83	**4.00**	**1(0)**	0.07	22	−1573.80	−302.97	290.55
MOGA8	5.96	**4.11**	**1(0)**	0.06	22	−1592.01	−344.66	159.01
MOGA9	5.79	**3.69**	**2(2)**	0.05	23	−1531.94	−350.76	268.99

BKS gives the number of best-known solutions retrieved or (in brackets) improved by the MOGA.

or (in brackets) improved (column *BKS*). All MOGAs except MO1 and MO3 outperform CARPET in terms of average deviation to the lower bound on the total cost and retrieve some best-known solutions (CARPET: 0). MO9 is even able to improve two best-known solutions, namely *egl-e3-c* (10349 vs. 10369) and *egl-s3-b* (14004 vs. 14028). Contrary to *gdb* and *val* files, the MOGAs reach the optimal makespan for a majority of instances and become faster than the memetic algorithm. In particular, the last MOGA finds all optimal makespans except one. Average running times are still acceptable: less than 5 min. For the detailed results in Table 7, we preferred to select MO9 with the lowest deviation to LB_1 and its two improved instances, rather than MO5 which has a slightly better measure $\bar{\mu}$.

4.5. Graphical examples

Graphical representations are well suited to visualize the behavior of bi-objective optimization methods. The kind of progression performed by our MOGAs is illustrated in Fig. 4. The graphic represents in the objective space the random initial solutions (symbol +), the three good heuristic solutions (symbol ×) and the final population (symbol ∗) computed by MO9 on the *egl-e3-a* instance (77 nodes, 98 edges and 87 tasks). Remember that MO9 is the version which performs 200 iterations and applies LS4 to all solutions, every 10 iterations.

One can notice that the initial heuristic solutions are already quite close to the final population. As already mentioned, adding such solutions is very useful to accelerate the search. Fig. 5 provides a magnified representation of the final population. A dashed line is used to emphasize efficient solutions.

The impact of the other key-component, the local search procedure, can be illustrated on the same instance by comparing the efficient set computed by the basic version MO1 (no local search, 100 iterations) with the one obtained by the best version MO9 (periodic local search LS4, 200 iterations). See Fig. 5. Even if the total number of iterations is increased for MO1, the results are only slightly improved. As for single objective GAs, hybridization with local search is necessary to obtain efficient MOGAs. Here for instance, MO9 decreases the best total cost by 10.55% (from 6659 to 5956) and the best makespan by 10.97% (from 921 to 820). Moreover, more efficient solutions are obtained and they are better spread, thus providing the decision maker with a wider choice.

Table 5
Detailed results of MO9 for *gdb* files

File	n	m	LB_1	LB_2	CARPET	Time	MA	Time	BKS	$\|F\|$	f_1^{left}	f_2^{left}	f_1^{right}	f_2^{right}	Time	μ	$\bar{\mu}$
gdb1	12	22	316	63	316*	1.75	316*	0.00	316*	3	316*	74	337	63*	8.36	−30.08	−10.03
gdb2	12	26	339	59	339*	2.87	339*	0.24	339*	3	339*	69	395	59*	10.27	−18.86	−6.29
gdb3	12	22	275	59	275*	0.04	275*	0.03	275*	4	275*	65	339	59*	8.57	−31.44	−7.86
gdb4	11	19	287	64	287*	0.05	287*	0.00	287*	3	287*	74	350	64*	7.07	−11.79	−3.93
gdb5	13	26	377	64	377*	3.10	377*	0.06	377*	6	377*	78	447	64*	10.46	−12.78	−2.13
gdb6	12	22	298	64	298*	0.47	298*	0.09	298*	4	298*	75	351	64*	8.16	−40.47	−10.12
gdb7	12	22	325	57	325*	0.00	325*	0.03	325*	3	325*	68	381	61	8.55	−14.02	−4.67
gdb10	27	46	344	38	352	33.85	350	22.10	348	4	350	44	390	38*	22.45	−19.35	−4.84
gdb11	27	51	303	37	317	29.92	303*	3.93	303*	3	309	43	333	37*	27.17	−31.77	−10.59
gdb12	12	25	275	39	275*	0.86	275*	0.03	275*	4	275*	71	297	54	10.24	−33.43	−8.36
gdb13	22	45	395	43	395*	1.27	395*	0.70	395*	5	395*	81	421	64	33.68	−105.14	−21.03
gdb14	13	23	448	93	458	11.45	458	5.43	458	4	458	97	547	93*	7.26	−64.54	−16.13
gdb15	10	28	536	128	544	1.34	536*	4.12	536*	1	544	128*	544	128*	9.03	−10.22	−10.22
gdb16	7	21	100	15	100*	0.27	100*	0.03	100*	3	100*	21	112	17	7.66	−1.36	−0.45
gdb17	7	21	58	8	58*	0.00	58*	0.00	58*	2	58*	15	60	13	9.66	−4.36	−2.18
gdb18	8	28	127	14	127*	0.94	127*	0.03	127*	4	127*	27	135	19	10.75	−13.78	−3.44
gdb19	8	28	91	9	91*	0.00	91*	0.03	91*	1	91*	15	91*	15	12.81	−2.77	−2.77
gdb20	9	36	164	19	164*	0.16	164*	0.06	164*	3	164*	33	178	27	18.05	−19.44	−6.48
gdb21	8	11	55	17	55*	0.11	55*	0.00	55*	2	55*	21	63	17*	4.80	−3.47	−1.73
gdb22	11	22	121	20	121*	5.27	121*	0.18	121*	5	121*	36	131	20*	8.10	−16.61	−3.32
gdb23	11	33	156	15	156*	0.63	156*	0.09	156*	4	156*	30	160	22	14.57	−10.69	−2.67
gdb24	11	44	200	12	200*	1.88	200*	1.86	200*	4	200*	26	207	20	24.03	−21.02	−5.25
gdb25	11	55	233	13	235	19.08	233*	28.41	233*	4	235	23	241	20	35.45	−22.96	−5.74
Average					0.48%	5.01	0.15%	2.94	0.13%	3.43	0.36%	47.09%	10.70%	20.95%	13.79	−23.49	−6.53
Worst					4.62%	33.85	1.78%	28.41	1.78%	6	2.23%	116.67%	23.27%	66.67%	35.45	−1.36	−0.45
Optima					18		21		21		18	1	1	12			

BKS is the best solution value (total cost) found by CARPET and the MA using several settings of parameters.
CPU times in seconds on a 1.8 GHz Pentium-IV PC. Times for CARPET and MA have been scaled. See Section 4.4.1 for details.
* indicates an optimal solution.

Table 6
Detailed results of MO7 for *val* files

File	n	m	LB_1	LB_2	CARPET	Time	MA	Time	BKS	$\|F\|$	f_1^{left}	f_2^{left}	f_1^{right}	f_2^{right}	Time	μ	$\bar{\mu}$
val1a	24	39	173	40	173*	0.02	173*	0.00	173*	1	173*	58	173*	58	20.82	−21.08	−21.08
val1b	24	39	173	40	173*	9.26	173*	8.02	173*	6	173*	61	204	42	21.39	−46.67	−7.78
val1c	24	39	235	40	245	93.20	245	28.67	245	2	245	41	248	40*	14.25	−60.72	−30.36
val2a	24	34	227	71	227*	0.17	227*	0.05	227*	6	227*	114	270	90	15.92	−39.62	−6.60
val2b	24	34	259	71	260	13.02	259*	0.22	259*	5	260	101	306	78	15.83	−26.45	−5.29
val2c	24	34	455	71	494	31.66	457	21.76	457	1	463	71*	463	71*	9.49	−7.69	−7.69
val3a	24	35	81	27	81*	0.77	81*	0.05	81*	4	81*	41	88	31	16.40	−40.05	−10.01
val3b	24	35	87	27	87*	2.79	87*	0.00	87*	4	87*	32	105	27*	14.88	−23.83	−5.96
val3c	24	35	137	27	138	41.66	138	28.23	138	1	138	27*	138	27*	10.39	−14.00	−14.00
val4a	41	69	400	80	400*	28.32	400*	0.72	400*	4	400*	134	446	92	89.86	−187.06	−46.77
val4b	41	69	412	80	416	75.66	412*	1.21	412*	9	412*	105	468	83	79.25	−402.04	−44.67
val4c	41	69	428	80	453	70.06	428*	19.11	428*	10	430	99	482	80*	73.68	−474.05	−47.40
val4d	41	69	520	80	556	233.56	541	103.26	530	1	539	80*	539	80*	50.42	−24.43	−24.43
val5a	34	65	423	72	423*	3.80	423*	1.86	423*	4	423*	141	474	96	72.75	−143.50	−35.88
val5b	34	65	446	72	448	41.40	446*	1.04	446*	8	446*	112	506	86	73.83	−151.41	−18.93
val5c	34	65	469	72	476	53.27	474	101.01	474	9	474	96	541	80	67.82	−189.67	−21.07
val5d	34	65	571	72	607	224.11	581	90.74	581	4	595	81	686	72*	51.83	−44.57	−11.14
val6a	31	50	223	45	223*	3.89	223*	0.17	223*	5	223*	75	259	56	37.08	−33.93	−6.79
val6b	31	50	231	45	241	26.94	233	67.34	233	7	233	68	263	50	34.93	−50.39	−7.20
val6c	31	50	311	45	329	85.18	317	52.23	317	5	317	55	329	45*	21.88	6.48	1.30
val7a	40	66	279	39	279*	6.59	279*	1.97	279*	3	279*	85	289	59	72.00	−44.27	−14.76
val7b	40	66	283	39	283*	0.02	283*	0.44	283*	4	283*	58	299	51	79.89	−81.59	−20.40
val7c	40	66	333	39	343	121.44	334	101.17	334	5	335	50	352	40	61.00	−9.03	−1.81
val8a	30	63	386	67	386*	3.84	386*	0.66	386*	7	386*	129	429	87	70.02	−172.40	−24.63
val8b	30	63	395	67	401	81.46	395*	9.95	395*	9	395*	100	455	79	66.16	−227.49	−25.28
val8c	30	63	517	67	533	147.40	527	71.46	527	7	545	74	610	67*	43.29	−180.52	−25.79
val9a	50	92	323	44	323*	28.51	323*	18.29	323*	3	326	82	333	68	171.97	−109.49	−36.50
val9b	50	92	326	44	329	59.89	326*	29.39	326*	3	326*	82	340	58	170.31	−108.22	−36.07
val9c	50	92	332	44	332*	56.44	332*	71.19	332*	12	332*	69	389	51	175.60	−330.71	−27.56
val9d	50	92	382	44	409	353.28	391	211.13	391	6	399	50	434	44*	134.13	−106.55	−17.76
val10a	50	97	428	47	428*	5.52	428*	25.48	428*	5	428*	143	449	91	203.20	−203.37	−40.67
val10b	50	97	436	47	436*	18.43	436*	4.67	436*	5	436*	111	459	77	194.28	−175.23	−35.05
val10c	50	97	446	47	451	93.47	446*	17.30	446*	7	448	93	498	66	205.89	−169.32	−24.19
val10d	50	97	524	47	544	156.31	530	215.04	528	6	537	61	595	54	149.31	−245.01	−40.83
Average					1.90%	63.87	0.61%	38.35	0.54%	5.24	0.99%	52.68%	10.69%	19.62%	76.17	−121.70	−21.85
Worst					8.57%	353.28	4.26%	215.04	4.26%	12	5.42%	204.26%	20.69%	93.62%	205.89	6.48	1.30
Optima					15		22		22		18	3	1	10			

BKS is the best solution value (total cost) found by CARPET and the MA using several settings of parameters.

CPU times in seconds on a 1.8 GHz Pentium-IV PC. Times for CARPET and MA have been scaled. See Section 4.4.1 for details.

* indicates an optimal solution.

Table 7
Detailed results of MO9 for *egl* files

File	n	m	τ	LB_1	LB_2	CARPET	MA	Time	BKS	$\|F\|$	f_1^{left}	f_2^{left}	f_1^{right}	f_2^{right}	Time	μ	$\bar{\mu}$
egl-e1-a	77	98	51	3515	820	3625	3548	74.26	3548	4	3548	943	3824	820*	31.39	−1431.21	−357.80
egl-e1-b	77	98	51	4436	820	4532	4498	69.48	4498	3	4525	839	4573	820*	25.70	−753.88	−251.29
egl-e1-c	77	98	51	5453	820	5663	5595	71.18	5595	2	5687	836	5764	820*	23.33	−177.03	−88.51
egl-e2-a	77	98	72	4994	820	5233	5018	152.58	5018	5	5018	953	6072	820*	77.93	−1269.08	−253.82
egl-e2-b	77	98	72	6249	820	6422	6340	153.41	6340	6	6411	864	6810	820*	65.47	−1581.27	−263.54
egl-e2-c	77	98	72	8114	820	8603	8415	129.63	8395	4	8440	854	8651	820*	56.02	−230.42	−57.61
egl-e3-a	77	98	87	5869	820	5907	5898	242.00	5898	12	5956	917	7935	820*	149.30	−1860.85	−155.07
egl-e3-b	77	98	87	7646	820	7921	7822	255.35	7816	7	7911	872	8455	820*	104.96	−1592.97	−227.57
egl-e3-c	77	98	87	10 019	820	10 805	10 433	206.35	10 369	4	10 349	864	10 511	820*	95.80	−827.25	−206.81
egl-e4-a	77	98	98	6372	820	6489	6461	291.87	6461	11	6548	890	7362	820*	166.66	−2003.50	−182.14
egl-e4-b	77	98	98	8809	820	9216	9021	312.85	9021	5	9116	874	9584	820*	145.44	−389.35	−77.87
egl-e4-c	77	98	98	11 276	820	11 824	11 779	252.38	11 779	1	11 802	820*	11 802	820*	111.18	−1041.39	−1041.39
egl-s1-a	140	190	75	4992	912	5149	5018	208.61	5018	11	5102	1023	6582	924	91.23	−1895.27	−172.30
egl-s1-b	140	190	75	6201	912	6641	6435	208.77	6435	7	6500	984	8117	912*	83.35	−1486.99	−212.43
egl-s1-c	140	190	75	8310	912	8687	8518	165.55	8518	5	8694	946	9205	912*	71.72	−1413.13	−282.63
egl-s2-a	140	190	147	9780	979	10 373	9995	874.36	9995	13	10 207	1058	12 222	979*	497.49	−1233.84	−94.91
egl-s2-b	140	190	147	12 886	979	13 495	13 174	760.50	13 174	10	13 548	1058	14 334	979*	516.85	−496.91	−49.69
egl-s2-c	140	190	147	16 221	979	17 121	16 795	746.93	16 715	4	16 932	1040	16 975	979*	399.14	−4042.88	−1010.72
egl-s3-a	140	190	159	10 025	979	10 541	10 296	1070.50	10 296	10	10 456	1099	12 605	979*	699.12	−3899.11	−389.91
egl-s3-b	140	190	159	13 554	979	14 291	14 053	1064.01	14 028	6	14 004	1040	15 103	979*	609.54	−6439.40	−1073.23
egl-s3-c	140	190	159	16 969	979	17 789	17 297	874.30	17 297	4	17 825	998	18 043	979*	493.60	−748.77	−187.19
egl-s4-a	140	190	190	12 027	1027	13 036	12 442	1537.59	12 442	3	12 730	1040	12 912	1027*	838.24	−255.12	−85.04
egl-s4-b	140	190	190	15 933	1027	16 924	16 531	1430.26	16 531	1	16 792	1027*	16 792	1027*	720.31	−271.94	−271.94
egl-s4-c	140	190	190	20 179	1027	21 486	20 832	1495.02	20 832	1	21 309	1027*	21 309	1027*	381.86	−1425.00	−1425.00
Average						4.74%	2.47%	526.99	2.40%	5.79	3.69%	6.31%	12.94%	0.05%	268.99	−1531.94	−350.77
Worst						8.61%	4.46%	1537.59	4.46%	13	5.85%	16.22%	35.20%	1.32%	838.24	−177.03	−49.69
Nb BKS retrieved						0	19		24		2						
Nb BKS improved											2						
Nb Optima												3		23			

BKS is the best solution value (total cost) found by CARPET and the MA using several settings of parameters.
CPU times in seconds on a 1.8 GHz Pentium-IV PC. Times for CARPET and MA have been scaled.
* indicates an optimal solution.

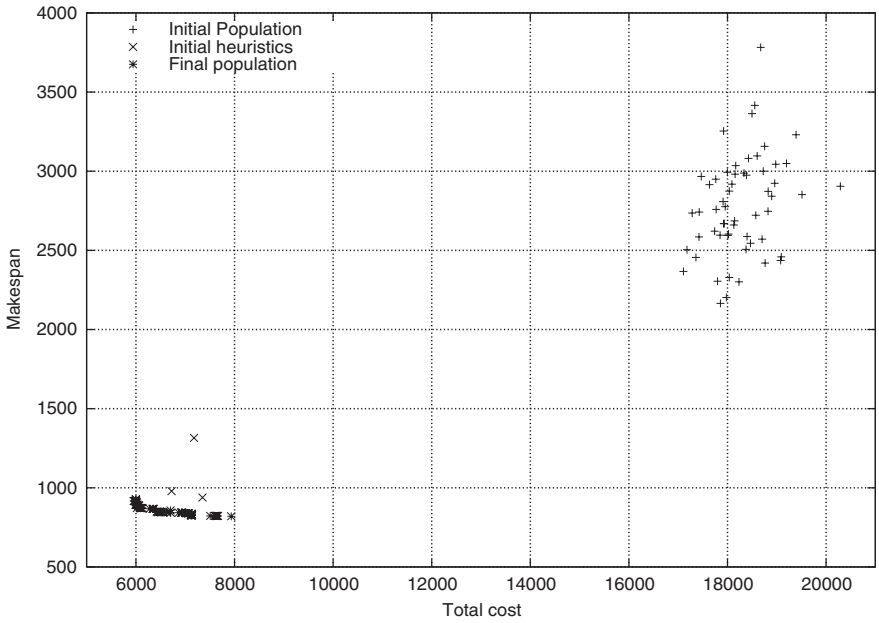

Fig. 4. Convergence of MO9 on instance *egl-e3-a*.

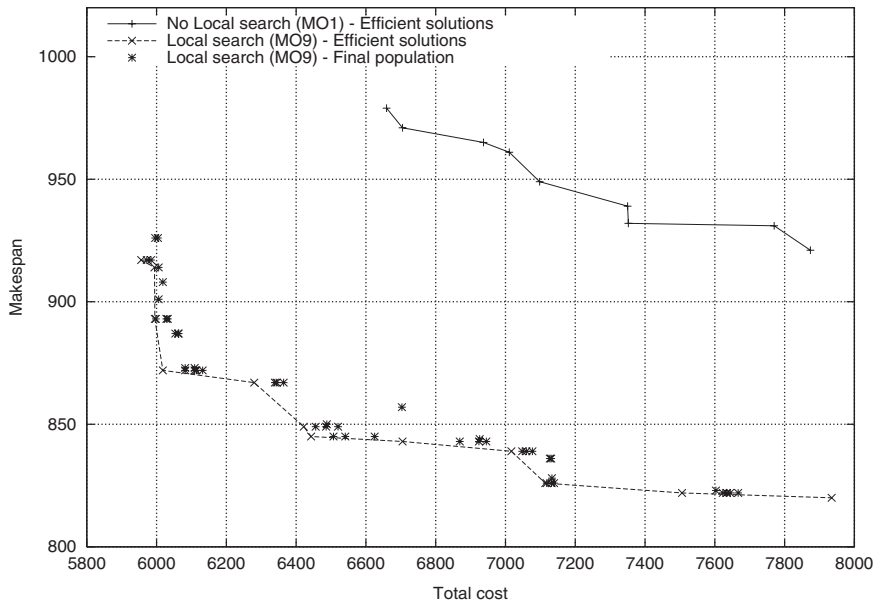

Fig. 5. Impact of local search for instance *egl-e3-a*.

5. Concluding remarks

In industry, most decision makers wish to take several criteria into account and the bi-objective CARP addressed in this article is a typical example. On the other hand, the combinatorial optimization community can help research in multi-objective optimization, by bringing a rich toolbox, very efficient algorithms, and a stricter methodology to evaluate and compare different algorithms.

In this paper, the implementation of the NSGA-II multi-objective genetic algorithm alone could not efficiently solve the bi-objective CARP. Two key-components were required to enhance performance: using good heuristics (Path-Scanning, Augment-Merge and Ulusoy's heuristic) in the initial population and adding a local search able to improve solutions for both criteria. It is worth noticing that the chromosome encoding, the crossover operator and the two proposed improvements come from a memetic algorithm successfully developed for the single objective CARP. The main difficulty was to select a local search adapted to the multi-objective case and to find its best location in the MOGAs.

An intensive testing on three sets of benchmarks, with a strict comparison completing a distance measure with lower bounds and metaheuristics for the single objective case, proves the validity and the efficiency of the proposed approach. In particular, the best versions of our MOGAs are still competitive with single objective metaheuristics. We even believe that any GA or memetic algorithm for a single objective problem could be generalized in the same way to handle several objectives, while keeping its efficiency, although this should be confirmed with other combinatorial optimization problems.

The bi-objective CARP was formulated according to a real case found in the town of Troyes (France) and in several other French cities. It is foreseen to apply the best MOGA to Troyes.

References

[1] Golden BL, DeArmon JS, Baker EK. Computational experiments with algorithms for a class of routing problems. Computers and Operations Research 1983;10(1):47–59.
[2] Golden BL, Wong RT. Capacitated arc routing problems. Networks 1981;11:305–15.
[3] Ulusoy G. The fleet size and mix problem for capacitated arc routing. European Journal of Operational Research 1985;22:329–37.
[4] Belenguer JM, Benavent E, Cognata F. Un metaheuristico para el problema de rutas por arcos con capacidades. In: Proceedings of the 23th national SEIO meeting, Valencia, Spain, 1997.
[5] Hertz A, Laporte G, Mittaz M. A tabu search heuristic for the capacitated arc routing problem. Operations Research 2000;48(1):129–35.
[6] Greistorfer P. A tabu-scatter search metaheuristic for the arc routing problem. Computers and Industrial Engineering 2003;44(2):249–66.
[7] Hertz A, Mittaz M. A variable neighborhood descent algorithm for the undirected capacitated arc routing problem. Transportation Science 2001;35(4):425–34.
[8] Beullens P, Muyldermans L, Cattrysse D, Van Oudheusden D. A guided local search heuristic for the capacitated arc routing problem. European Journal of Operational Research 2003;147(3):629–43.
[9] Lacomme P, Prins C, Ramdane-Chérif W. A genetic algorithm for the capacitated arc routing problem and its extensions. In: Boers EJW., et al., editor. Applications of evolutionary computing, Lecture Notes in Computer Science, vol. 2037. Berlin: Springer; 2001. p. 473–83.
[10] Lacomme P., Prins C., Ramdane-Chérif W. Competitive memetic algorithms for arc routing problems. Annals of Operations Research 2004;131:159–85.
[11] Amberg A, Voß S. A hierarchical relaxations lower bound for the capacitated arc routing problem. In: Sprague RH. (Hrsg.), editor. Proceedings of the 35th annual Hawaii international conference on systems sciences, Piscataway: IEEE; 2002. p. DTIST02:1–10.

[12] Belenguer JM, Benavent E. A cutting plane algorithm for the capacitated arc routing problem. Computers and Operations Research 2003;30(5):705–28.
[13] Lacomme P, Prins C, Sevaux M. Multi-objective capacitated arc routing problem. In: Fonseca CM., et al., editor. Evolutionary multi-criterion optimization (Proceedings of EMO 2003, Faro, Portugal), Lecture Notes in Computer Science, vol. 2632. Berlin: Springer; 2003. p. 550–64.
[14] Park YB, Koelling CP. An interactive computerized algorithm for multicriteria vehicle routing problems. Computers and Industrial Engineering 1989;16:477–90.
[15] Corberan A, Fernandez E, Laguna M, Martí R. Heuristic solutions to the problem of routing school buses with multiple objectives. Journal of the Operational Research Society 2002;53(4):427–35.
[16] Hong SC, Park YB. A heuristic for bi-objective vehicle routing problem with time windows constraints. International Journal of Production Economics 1999;62:249–58.
[17] Sessomboon W, Watanabe K, Irohara T, Yoshimoto K. A study on multi-objective vehicle routing problem considering customer satisfaction with due-time (the creation of pareto optimal solutions by hybrid genetic algorithm). Transactions of the Japan Society of Mechanical Engineers, 1998.
[18] Ehrgott M, Gandibleux X. Multiobjective combinatorial optimization. In: Ehrgott M., Gandibleux X, editors. International series in operations research and management science, vol. 52. Dordrecht: Kluwer; 2002. p. 369–444.
[19] Coello Coello CA, Van Veldhuizen DA, Lamont GB. Evolutionary algorithms for solving multi-objective problems. New York: Kluwer; 2002.
[20] Deb K. Multi-objective optimization using evolutionary algorithms. Chichester, UK: Wiley; 2001.
[21] Coello Coello CA. An updated survey of GA-based multiobjective optimization techniques. ACM Computing Surveys 2000;32(2):109–43.
[22] Deb K. Multi-objective genetic algorithms: problem difficulties and construction of test problems. Evolutionary Computation 1999;7(3):205–30.
[23] Zitzler E, Deb K, Thiele L. Comparison of multiobjective evolutionary algorithms: empirical results. Evolutionary Computation 2000;8(2):173–95.
[24] Deb K, Pratap A, Agarwal S, Meyarivan T. A fast and elitist multi-objective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation 2002;6(2):182–97.
[25] Moscato P. Memetic algorithms: a short introduction. In: Corne D, Dorigo M, Glover F, editors. New ideas in optimization. Maidenhead, UK: McGraw-Hill; 1999. p. 219–34.
[26] Jaszkiewicz A. Genetic local search for multiple objective combinatorial optimization. European Journal of Operational Research 2001;137(1):50–71.
[27] Jaszkiewicz A. Do multiple objective metaheuristics deliver on their promises? a computational experiment on the set covering problem. IEEE Transactions on Evolutionary Computation 2003;7(2):133–43.
[28] Murata T, Nozawa H, Ishibuchi H, Gen M. Modification of local search directions for non-dominated solutions in cellular multiobjective genetic algorithms for pattern classification problems. In: Fonseca CM., et al., editor. Evolutionary multi-criterion optimization (Proceedings of EMO 2003, Faor, Portugal), Lecture Notes in Computer Science, vol. 2632. Berlin: Springer; 2003. p. 593–607.
[29] Riise A. Comparing genetic algorithms and tabu search for multi-objective optimization. In: Abstract conference proceedings. Edinburgh, UK: IFORS; 2002. p. 29.