Using an Exact Bi-Objective Decoder in a Memetic Algorithm for Arc-Routing (and Other Decoder-Expressible) Problems

Daniel Porumbel[1], Igor M. Coelho[2], El-Ghazali Talbi[3]

[1] CEDRIC CS Laboratory, CNAM, 292 rue Saint Martin, Paris, France (daniel.porumbel@cnam.fr)
[2] Computing Institute, Universidade Federal Fluminense Av. General Milton Tavares de Souza - Niteroi/RJ - Brazil - 24210-346
[3] University of Lille and INRIA

Abstract

We address the bi-objective Capacitated Arc Routing Problem (CARP) by considering two levels of solution interpretation: implicit and explicit solutions. An algorithm that translates implicit solutions into explicit solutions is called a decoder. In this work, the decoder takes as input a permutation of the required edges and generates a Pareto frontier of CARP solutions. While bi-objective CARP was our main focus and starting point, we could also use the proposed framework to solve a bi-objective version of the traveling salesman problem by plugging-in a different decoder. Recall that bi-objective CARP asks to service (the demands of) a set of required edges using a fleet of vehicles of limited capacity so as to minimize: (i) the total travelled distance and (ii) the length of the longest route. Any permutation s of the required edges constitutes an implicit CARP solution. The decoder constructs all non-dominated explicit solutions that service the edges in the order indicated by s, *i.e.*, the decoder is an exact algorithm that returns the optimal Pareto frontier subject to the service order s. To achieve competitive CARP results it is also important to reinforce the decoder using a local search operator that acts on explicit routes (and that may change the service order s). For nine instances, the resulting algorithm was even able to find a new total-cost upper bound, improving upon the best solutions reported in the (considerably larger) mono-objective CARP literature. This shows that (some of) the proposed ideas can also be useful in single objective optimization: the second objective can be seen as a guide for the mono-objective search process.

Keywords: combinatorial optimization, bi-objective exact decoder, bi-objective EA, permutation problem

1 Introduction

Let us first introduce the most widespread bi-objective variant of the celebrated Capacitated Arc Routing Problem (CARP). Given a graph $G(V, E)$, the goal is to find a set of *feasible routes* that service a set of required edges $E_R \subseteq E$, under the constraint that the service amount on each route cannot exceed a given (vehicle) capacity. The bi-objective variant was first introduced by [Lacomme et al., 2006] and it requires minimizing both the *total cost* (the total distance travelled by all routes) and the *makespan* (the length of the longest route). The second objective may offer a higher degree of planning flexibility, *e.g.*, enabling one to reduce working time inequality or comply with legislative shift-time limits. This second objective also measures the total time elapsed from the moment when all vehicles leave the depot until the last vehicle returns; this may be important because the return of the last vehicle might trigger some other events.[1]

[1]We can cite [Lacomme et al., 2006] for an example: “in Troyes (France), all trucks leave the depot at 6am and [the whole] 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.”

We consider two levels of solution interpretations: permutations (of E_R) and explicit feasible CARP solutions. We offer the possibility of switching from the former to the latter by applying a one-to-many exact decoder based on dynamic programming. This decoder enables us to reduce the huge CARP search space to a more reasonably-sized search space, namely the set of permutations of E_R. Thus, CARP is transformed into a sequencing or permutation problem [Campos et al., 2005, Porumbel et al., 2017, van Hoorn, 2016], *i.e.*, a problem for which the candidate solutions are encoded as (sequences or) permutations. One can thus readily use the numerous mutation or crossover operators already developed in the literature of permutation problems. The overall algorithm is based on the following three main components that can be designed and studied separately (in isolation).

1. An exact decoder based on dynamic programming that can turn any given permutation s of E_R into a Pareto frontier of explicit CARP solutions that do not dominate one another. These explicit solutions all service the required edges in the order indicated by s (Section 2.1).
2. A Local Search (LS) algorithm that may improve the minimum total cost solution returned by the above decoder. If the solution obtained after this LS services the edges in an order s' different from s, we call (again) the decoder on s', but without any LS this second time (Section 2.2).
3. An evolutionary algorithm (EA) framework in which we generalize the non-dominated sorting idea of NSGA2 [Deb et al., 2002], so as handle both implicit and explicit solutions. We are in a more complex context than in a pure NSGA2, because an implicit solution is associated to multiple explicit solutions, and so, to multiple points in the objective space (unlike in NSGA2). The description of this EA does not use any particular CARP feature, because all these features are hidden behind the (abstract) decoder. We will rather focus more general concepts, *e.g.*, we will propose techniques to preserve diversity, to control the population dynamics or to encourage young offspring against old individuals (Section 3).

The EA framework from the last above point represents the main algorithmic "backbone" of the overall solution method. The complete algorithm, hereafter referred to as `Decoder-EA`, is actually constructed by connecting the two other components (points 1 and 2 above) to this EA. One can make `Decoder-EA` solve a different bi-objective problem, by simply replacing the above decoder with a different one. Based on this idea, we present in Section 5 a `Decoder-EA` implementation that solves a bi-objective variant of the well known Traveling Salesman Problem (TSP). We did not need to change a single line of code in the software module that implements the EA framework to solve this TSP variant.

If the general mono-objective CARP has attracted the interest of tens if not hundreds of researchers, we are aware of only six papers that address the bi-objective variant considered here. The earliest work appeared in 2006 when [Lacomme et al., 2006] presented a genetic algorithm designed by adapting the NSGA2 framework [Deb et al., 2002] to CARP. An individual is given by a list of directed required edges that is evaluated using a heuristic called the Ulusoy's *giant tour split* (see [Ulusoy, 1985] or [Porumbel et al., 2017, §5.1]). Compared to this early approach, our decoder is (significantly) more complex for two reasons. First, our decoder is exact and not heuristic. Secondly, it does not use directed edges but simple edges, hence reducing the size of the search space size by a 2^n factor.[2] In addition, the LS operator is very different and so is the EA implementation. Yet, we share an important conclusion of [Lacomme et al., 2006] that the EA "must be hybridized with a local search procedure to be able to compete with state-of-the-art metaheuristics"; [Lacomme et al., 2006] present many other general considerations that are relevant in our work, *e.g.*, motivations for using an unlimited fleet (§1), reasons for choosing NSGA2 (§2), or a description of the difficulties raised by the integration of LS (§3). We also refer the reader to [Corberán et al., 2021] for an extensive review (that cover many bi-objective aspects) of the main Arc Routing developments produced up to 2021, with over 230 references.

We discuss below in chronological order the remaining papers devoted to bi-objective CARP, even if they generally rely on techniques that are only remotely related to the ones from our work, *e.g.*, they use no

[2]This 2^n reduction enabled us to even compute the optimal solution (the certified-optimal Pareto frontier) for the smallest instance with $n = 11$. We simply executed the decoder on each of the 11! permutations that cover the whole implicit space for this instance. (see Appendix A). Without the 2^n factor reduction, the search space would have been $2^{11} = 2048$ times larger!

indirect list or permutation representation, there is no decoding, the LS and the NSGA2 (when present) are applied in a different manner.

- The Decomposition-Based Memetic Algorithm (D-MAENS) by Mei et. al [Mei et al., 2011]. In this decomposition framework, the objective values (of a solution) can be aggregated into a unique objective value. Using N different vectors of aggregation weights, one obtains $N > 2$ mono-objective sub-problems (decomposition directions); these sub-problems are associated to N representative solutions that evolve along the search in different sub-populations [Mei et al., 2011, §(iv).a]. The bi-objective problem is thus decomposed into N mono-objective sub-problems; each one of them could be addressed by a well-established mono-objective CARP algorithm called MAENS.
- The **ϵ-constraint** method by Grandinetti et al. [Grandinetti et al., 2012]. The main idea in this work is to solve a mono-objective $\epsilon-$constrained problem for each objective, *i.e.*, minimize a chosen objective while limiting the value of each other remaining objective to a certain ϵ value. The method proceeds by gradually reducing the ϵ parameter which leads to a sequence of ϵ-constrained problems (§ 3). Each resulting mono-objective problem is associated to an Integer Linear Program (ILP) that is solved using the commercial `Cplex` solver. This ILP needs a pool of pre-existing routes that are generated using LS before calling `Cplex`.
- The Improved D-MAENS (ID-MAENS) by Shang et al [Shang et al., 2014]. This work takes the above D-MAENS to a higher level by introducing a steady-state replacement strategy and an elitist archive. Regarding the replacement, this new work proposes to replace an existing individual immediately once an offspring solution is generated, which speeds-up the convergence. Concerning the elitist strategy, an archive is introduced to record the solution that has the best value according to each decomposition direction. Recall that a decomposition direction is given by a vector of weights used in the objective aggregation described above (for D-MAENS). This new work also proposed a more dynamic method to construct sub-populations and to associate them to decompositions directions (sub-problems).
- The Improved Route Distance Grouping MAENS (IRDG-MAENS) by Shang et al [Shang et al., 2016a]. This work improves a mono-objective CARP algorithm called RDG-MAENS [Mei et al., 2014] and successfully generalizes it to bi-objective CARP. Both these algorithms use the notions of sub-populations and decomposed sub-problems (as for ID-MAENS); routes and individuals can be distributed to different sub-populations using new, fast and effective techniques.
- The Directed Evolution Immune Clonal Algorithm (DE-ICA) by Shang et al [Shang et al., 2016b]. This algorithm builds upon the theory of artificial immune systems. The proposed ICA has certain similarities to EAs, because it uses a population (of antibodies) and a notion of reproduction (of the antibodies) or mutation; however, the description of such notions lies beyond the scope of this introduction.

The remaining is organized as follows. Section 2 presents the CARP definition, the proposed bi-objective decoder and the specific LS. Section 3 describes the EA framework in a general decoder-based optimization context only characterized by the existence of a decoder that hides all CARP aspects. Section 4 reports numerical results on CARP. Section 5 presents the application of the proposed framework to a bi-objective TSP variant, including numerical results on this new problem. We conclude in Section 6. A short appendix reports the Pareto frontiers we discovered for a few small instances, including the certified optimal solutions for the smallest one; a second appendix presents a decoder execution example.

2 Bi-objective CARP definition, decoder and local search

The bi-objective CARP is defined on a graph $G = (V, E)$, where V is a set of nodes and E is a set of edges. A subset of edges $E_R \subseteq E$ with $|E_R| = n$ represents the required edges that must be serviced (once) using an unlimited fleet of homogeneous vehicles of capacity W. Each edge $\{i, j\} \in E$ has a (traversal) cost c_{ij};

the required edges $\{i, j\} \in E_R$ also have a demand q_{ij}. A feasible route starts and ends at a special depot node v_0; it can not supply an amount of service larger than W. An edge may be traversed multiple times by different routes and an edge traversed without service is called a *deadheaded edge*.

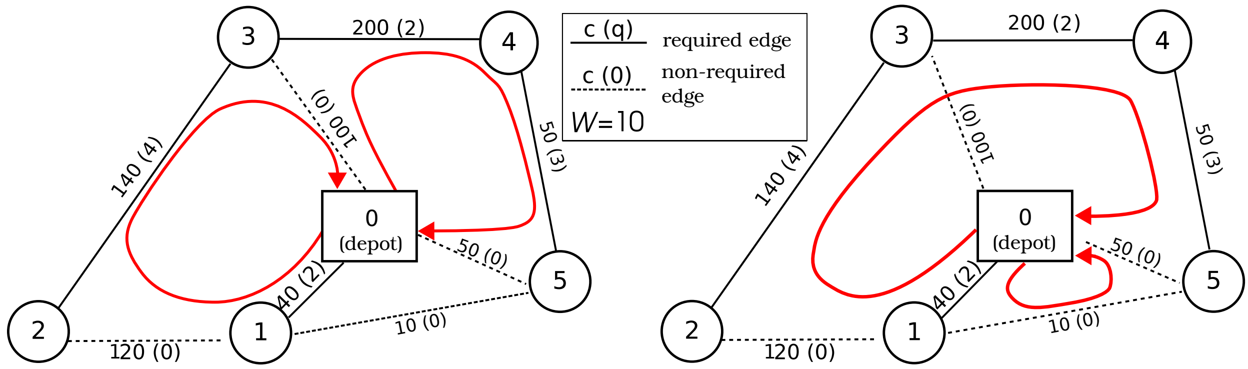


Figure 1: Two non-dominated solutions for the same instance. Each edge $\{i, j\}$ has a label "c_{ij} (q_{ij})", where c_{ij} is the traversal cost and q_{ij} is the required service (0 for non-required edges). The left solution has two routes of equal cost 400. The right solution contains a shorter route of cost 100 (only for $\{0, 1\}$) and a longer route of cost 600. The two solutions have objective values $(800, 400)$ and resp. $(700, 600)$.

Let $\mathscr{X}$ be the set of all explicit CARP feasible solutions, *i.e.*, the explicit search space. The bi-objective CARP asks to find a solution $\mathbf{x} \in \mathscr{X}$ containing a set $\mathbf{r}$ of routes that service all required edges and that minimize: (i) the total cost C^{tot} of all traversed edges and (ii) the length $C^{\max}$ of the longest route. Technically, we can write:

$$C^{\text{tot}} = \min_{\mathbf{x} \in \mathscr{X}} C^{\text{tot}}(\mathbf{x}), \text{ where } C^{\text{tot}}(\mathbf{x}) = \sum_{\mathbf{r} \in \mathbf{x}} \sum_{\{i,j\} \in \mathbf{r}} c_{ij} \tag{1a}$$

$$C^{\max} = \min_{\mathbf{x} \in \mathscr{X}} C^{\max}(\mathbf{x}), \text{ where } C^{\max}(\mathbf{x}) = \max_{\mathbf{r} \in \mathbf{x}} \sum_{\{i,j\} \in \mathbf{r}} c_{ij} \tag{1b}$$

To cope with these two conflicting goals, the decision maker may have to choose a compromise solution from a Pareto frontier of solutions that do not dominate one another. Figure 1 provides a simple example of two conflicting solutions for a small graph. An exact decoder applied on input permutation $(\{0, 1\}, \{2, 3\}, \{3, 4\}, \{4, 5\})$ would have to return these two Pareto-optimal solutions.

Section 2.1 describes the proposed decoder. While the general idea of transforming ordered lists (of tasks) into routes is relatively popular in general vehicle routing, [3] this is the first bi-objective CARP decoder that is exact and not heuristic. A second advantage compared to other approaches is using permutations of *non-directed* edges as input, which reduces the search space size by a factor of 2^n compared to more popular "split with flips" decoders that work with directed edges.

Section 2.2 describes the LS algorithm to be executed on the C^{tot}-best solution from the Pareto frontier returned by the exact decoder. After this LS round, the service order may change; in such a case, one can call again the decoder on the new permutation (the new service order) but without any LS this second time.

2.1 The Dynamic programming exact one-to-many decoder

The decoder described hereafter extends the dynamic programming scheme for mono-objective CARP from [Porumbel et al., 2017]. In both cases, the input consists of a permutation s of non-directed edges.

[3] See [Porumbel et al., 2017, §5.1] or the review [Prins et al., 2014]. As mentioned in the introduction, a variant of Ulusoy's `split` was already used in [Lacomme et al., 2006, § 2.2.1] to transform ordered lists into explicit CARP solutions. Such decoders usually return a unique solution obtained by optimizing a mono-objective criterion and they are often inexact in a bi-objective context.

2.1.1 Notations, definitions and example

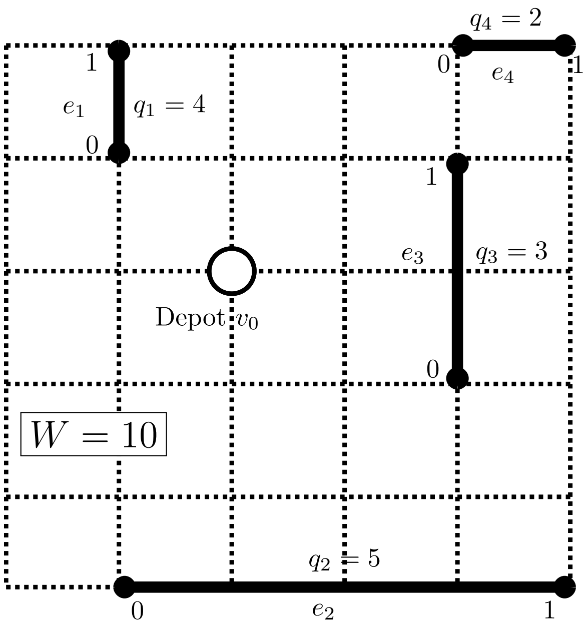
Definition 1. *Let $\mathscr{S}$ be the encoded search space that contains all permutations of E_R. Let $\mathscr{X}$ be the explicit search space. We consider a decoder function $\mathscr{D} : \mathscr{S} \to 2^{\mathscr{X}}$ that maps any permutation $s \in \mathscr{S}$ to a set of explicit solutions from $\mathscr{X}$.*

Given input permutation $s \in \mathscr{S}$, the proposed decoder returns a set $\mathscr{D}(s)$ of explicit solutions that service the edges E_R in the order imposed by s and that are Pareto non–dominated: there is no $x, x' \in \mathscr{D}(s)$ such that $x \preceq x'$, *i.e.*, such that $C^{\text{tot}}(x) \leq C^{\text{tot}}(x')$ and $C^{\max}(x) \leq C^{\max}(x')$. The decoder is exact in the sense that the Pareto frontier $\mathscr{D}(s)$ is optimal to the given CARP instance subject to the service order s.

We now need more detailed notations, see also Figure 2 to follow them more easily.

Definition 2. *Given permutation $s = (e_1, e_2, \ldots, e_k, \ldots, e_n)$ with $e_k = \{i, j\}$, we define the following notations:*

- *the cost of travelling along $e_k = \{i, j\}$ is $c_k = c_{ij}$.*
- *the demand of edge $e_k = \{i, j\}$ is q_k*
- *the end points of edge $e_k = \{i, j\}$ are denoted by e_k^0 and e_k^1, such that $e_k^0 = i$ and $e_k^1 = j$.*
- *$\texttt{len}(e_k)$ is the maximum number of edges that can be serviced from edge e_k onwards without exceeding the capacity W, i.e., $\texttt{len}(e_k) = \max\{\ell : q_k + q_{k+1} + \cdots + q_{k+\ell-1} \leq W\}$.*
- *$R(e_k, \ell)$ is the minimum cost of a complete route that services ℓ edges (in any sense) starting from e_k, i.e., start from the depot, service e_k, e_{k+1}, e_{k+2}, $\ldots$, $e_{k+\ell-1}$ and return to the depot.*
- *$D^0(e_k, \ell)$ and $D^1(e_k, \ell)$ represent the minimum cost of a route that services $e_k, e_{k+1}, e_{k+2}, \ldots, e_{k+\ell-1}$ (in any sense) and finishes at the end point $e^0_{k+\ell-1}$ or resp. $e^1_{k+\ell-1}$ of edge $e_{k+\ell-1}$;*



e_k	$\texttt{len}(e_k)$
e_1	2
e_2	3
e_3	2
e_4	1

e_k	$e_{k+\ell-1}$: e_1	e_2	e_3	e_4
e_1	6	18		
e_2		14	16	20
e_3			8	12
e_4				10

$R(e_k, \ell)$ for $k \in [1..4]$ and $\ell \in [1..\texttt{len}(e_k)]$

e_k	$e_{k+\ell-1}$: e_1	e_2	e_3	e_4
e_1	$(4,3)$	$(16,12)$		
e_2		$(10,8)$	$(15,13)$	$(16,15)$
e_3			$(5,5)$	$(8,7)$
e_4				$(6,5)$

$\big(D^0(e_k, \ell), D^1(e_k, \ell)\big)$ for $k \in [1..4]$ and $\ell \in [1..\texttt{len}(e_k)]$.

Figure 2: A bi-objective CARP instance and the values of the notations from Definition 2. There are 4 required edges in bold, each one having two end vertices (see labels 0 and 1 on each bold edge); we thus have a total of 9 vertices including the depot. We consider each two vertices are linked by a (required or non-required) edge whose traversal cost is given by the Manhattan distance between them.

Let us first focus on Figure 2 to familiarize with the above notations. The incremental calculation of $D^0(e_k, \ell), D^1(e_k, \ell)$ and $R(e_k, \ell)$ for $k \in [1..n]$ and $\ell \in [1..\texttt{len}(e_k)]$ will naturally give rise to a dynamic programming scheme. Let us exemplify this calculation for input permutation $s = (e_1, e_2, e_3, e_4)$ and $k = 1$. First, we clearly have $\texttt{len}(e_1) = 2$ because $q_1 + q_2 = 9 \leq W$ and $q_1 + q_2 + q_3 = 12 > W$ (recall $W = 10$). We have to first determine the best routes for $\ell = 1$ and then the best routes for $\ell = 2$.

1.a) We have $D^0(e_1, 1) = 4$ because this corresponds to a route that goes from the depot v_0 to e_1^1 (*i.e.*, end 1 of e_1), services e_1 and stops at e_1^0 (*i.e.*, end 0 of e_1), which generates a walk of cost $3 + 1 = 4$. An analogous argument is used to calculate $D^1(e_1, 1) = 2 + 1 = 3$.

1.b) We have $R(e_1, 1) = 6$ because this corresponds to a route that starts from the depot, travels to either end of e_1, services e_1 and comes back to v_0. We actually have $D^0(e_1, 1) + 2 = D^1(e_1, 1) + 3 = 6$.

2.a) We now move to $\ell = 2$, *i.e.*, to routes servicing two edges.

– To calculate $D^0(e_1, 2)$, recall that e_1 has to be serviced before e_2 because the given permutation is (e_1, e_2, e_3, e_4). One can finish a (partial) trip in e_2^0 either:
  (i) by coming from e_1^0 after having serviced e_1 (*i.e.*, continue the path of $D^0(e_1, 1)$), or
  (ii) by coming from e_1^1 after having serviced e_1 (*i.e.*, continue the path of $D^1(e_1, 1)$).

  In both cases, the route has to first visit e_2^1 before servicing e_2 to end up in e_2^0. Thus, the above two choices lead to the formula below, where $\texttt{spath}: V \times V \to \mathbb{R}$ is a function encoding the shortest path in G (here given by the Manhattan distance).

$$D^0(e_1, 2) = \begin{array}{l} \min \Big(D^0(e_1, 1) + \texttt{spath}(e_1^0, e_2^1) + c_2, \\ \qquad D^1(e_1, 1) + \texttt{spath}(e_1^1, e_2^1) + c_2 \Big) \end{array} \begin{array}{l} = \min(4 + 8 + 4, \\ \qquad 3 + 9 + 4) \end{array} = 16 \tag{2}$$

– An analogous argument leads to $D^1(e_1, 2) = 12$.

2.b) The calculation of $R(e_1, 2)$ uses the values $D^0(e_1, 2)$ and $D^1(e_1, 2)$ determined above. We can write:

$$R(e_1, 2) = \begin{array}{l} \min \Big(D^0(e_1, 2) + \texttt{spath}(e_2^0, v_0), \\ \qquad D^1(e_1, 2) + \texttt{spath}(e_2^1, v_0) \Big) \end{array} \begin{array}{l} = \min(16 + 4, \\ \qquad 12 + 6), \end{array} = 18 \tag{3}$$

The above formulae (2)–(3) illustrate the general calculation of $D^1(e_1, \ell)$, $D^2(e_1, \ell)$ and $R(e_1, \ell)$ from the previously-computed $D^0(e_1, \ell - 1)$ and $D^1(e_1, \ell - 1)$ $\forall\, \ell \in [2..\texttt{len}(e_1)]$. After a few notational translations, we can actually apply the above (2)-(3) to any $\ell \in [2..\texttt{len}(e_1)]$:

$$D^0(e_1, \ell) = \min \Big(D^0(e_1, \ell - 1) + \texttt{spath}(e_{\ell-1}^0, e_\ell^1), D^1(e_1, \ell - 1) + \texttt{spath}(e_{\ell-1}^1, e_\ell^1) \Big) + c_\ell \tag{4a}$$

$$R(e_1, \ell) = \min \Big(D^0(e_1, \ell) + \texttt{spath}(e_\ell^0, v_0), D^1(e_1, \ell) + \texttt{spath}(e_\ell^1, v_0) \Big) \tag{4b}$$

These two formulae actually constitute the dynamic programming recursion that calculates R and D level by level, by iteratively increasing ℓ. The above calculation of the values $D^0(e_1, \ell)$, $D^1(e_1, \ell)$ and $R(e_1, \ell)$ for $\ell = 1$ and $\ell = 2$ is only an example of applying this recursion on ℓ, *i.e.*, only showing how to advance from $\ell = 1$ to $\ell = 2$. We could not move to $\ell = 3$ because $\texttt{len}(e_1) = 2$. The values thus calculated are visible in the first row from the tables D and R in the right part of Figure 2; one may check and verify the numbers in these two tables to gain full familiarity with the recursion.

The above formulae can be used to generate only routes that start by servicing e_1. To construct full solutions, we need chain (or put together) more routes that start at different edges e_k, for $k \in [1..n]$. For instance, a full Pareto-optimal solution for $n = 4$ may contain two routes: one of cost $R(e_1, 1)$ that services only one edge and one of cost $R(e_2, 3)$ that services three edges. To compute D and R for $k > 1$, it is actually enough to add an offset (of $k - 1$) to the subscript of each occurrence of "e" and "c" in (4a)-(4b), see Section 2.1.2 for the exact resulting formulae. There is no connexion between the calculation of D and R for two different values of k.

2.1.2 The complete pseudo-code

The pseudo-code consists of three major steps clearly emphasized in Algorithm 1. The first step simply initializes R, D^0 and D^1 as two-dimensional data structures; they are actually implemented as arrays of arrays, *e.g.*, D^0 is an array with $n = |E_R|$ positions, and, for each $k \in [1..n]$, the array $D^0[k]$ has $\texttt{len}(e_k)$ elements. The first part of Algorithm 1 provides full details on this initialization.

The second major step actually implements a generalization of (4a)-(4b). More exactly, Line 10 is obtained from (4a) by performing the following replacements: $e_1 \to e_{k+1}$, $e_{\ell-1} \to e_{k+\ell-1}$, and $c_\ell \to c_{k+\ell}$. Line 12 is obtained from (4b) by performing the above replacements and also $e_\ell \to e_{k+\ell}$.

Algorithm 1: Dynamic Programming Multi-Objective Decoder

Input: permutation $(e_1 \dots e_n)$ of the required edge E_R

// STEP 1: INITIALIZE len, R, D^0 AND D^1
1 $\texttt{len}, R, D^0, D^1 \leftarrow$ arrays with n positions;
2 **for** $k = 1$ **to** n **do**
3 $\quad \texttt{len}(e_k) \leftarrow \max\{\ell : \sum_{i=0}^{\ell-1} q_{k+i} \leq W\}$ // *max served edges*;
4 $\quad R(e_k)$, $D^0(e_k)$, $D^1(e_k) \leftarrow$ arrays with $\texttt{len}(e_k)$ elements // *R, D^0 and D^1 become matrices*;

// STEP 2: COMPUTE R, D^0 AND D^1
5 **for** $k = 0$ **to** $n - 1$ **do**
6 $\quad D^0(e_{k+1}, 1) \leftarrow \texttt{spath}(v_0, e^1_{k+1}) + c_{k+1}$ // *$\texttt{spath}(v_i, v_j)$ is the shortest path from v_i to v_j, $\forall v_i, v_j \in V$* ;
7 $\quad D^1(e_{k+1}, 1) \leftarrow \texttt{spath}(v_0, e^0_{k+1}) + c_{k+1}$;
8 $\quad R\ (e_{k+1}, 1) \leftarrow \min\{D^0(e_{k+1}, 1) + \texttt{spath}(e^0_{k+1}, v_0), D^1(e_{k+1}, 1) + \texttt{spath}(e^1_{k+1}, v_0)\}$;
9 $\quad$ **for** $\ell = 2$ **to** $\texttt{len}(e_{k+1})$ **do**
10 $\quad\quad D^0(e_{k+1}, \ell) \leftarrow \min(D^0(e_{k+1}, \ell - 1) + \texttt{spath}(e^0_{k+\ell-1},\ e^1_{k+\ell}) + c_{k+\ell},$
$\quad\quad\quad D^1(e_{k+1}, \ell - 1) + \texttt{spath}(e^1_{k+\ell-1},\ e^1_{k+\ell}) + c_{k+\ell})$; // *this implements* (4a)
11 $\quad\quad D^1(e_{k+1}, \ell) \leftarrow \min(D^0(e_{k+1}, \ell - 1) + \texttt{spath}(e^0_{k+\ell-1},\ e^0_{k+\ell}) + c_{k+\ell},$
$\quad\quad\quad D^1(e_{k+1}, \ell - 1) + \texttt{spath}(e^1_{k+\ell-1},\ e^0_{k+\ell}) + c_{k+\ell})$;
12 $\quad\quad R(e_{k+1}, \ell) \leftarrow \min(D^0(e_{k+1}, \ell) + \texttt{spath}(e^0_{k+\ell}, v_0),\ D^1(e_{k+1}, \ell) + \texttt{spath}(e^1_{k+\ell}, v_0))$; // *implement* (4b)

// STEP 3: CONSTRUCT NON-DOMINATED SOLUTIONS BY CHAINING INDIVIDUAL ROUTES RECORED IN R
13 $\texttt{sol} \leftarrow$ array indexed by $k \in [0..n]$ // *$\texttt{sol}[k]$ is a Pareto set of pairs of objective values (C^{tot}, C^{max})*
14 $\texttt{sol}[0] \leftarrow \{(0, 0)\}$ // *$k = 0$ means nothing serviced yet; $(0, 0)$ means both costs are 0*
15 **for** $k = 0$ **to** $n - 1$ **do**
16 $\quad$ **forall** $(C^{\mathbf{tot}}, C^{\mathbf{max}}) \in \texttt{sol}[k]$ **do**
17 $\quad\quad$ **for** $\ell = 1$ **to** $\texttt{len}(e_{k+1})$ **do**
18 $\quad\quad\quad C^{\text{tot}}_{+} \leftarrow C^{\text{tot}} + R(e_{k+1}, \ell)$ // *add a new route that services $e_{k+1}, e_{k+2}, \dots, e_{k+\ell}$*;
19 $\quad\quad\quad C^{\max}_{+} \leftarrow \max\big(C^{\max}, R(e_{k+1}, \ell)\big)$;
20 $\quad\quad\quad$ **if** $\nexists\, (C^{\mathbf{tot}}_{\mathbf{old}}, C^{\mathbf{max}}_{\mathbf{old}}) \in \texttt{sol}[k + \ell])$ **such that** $(C^{\mathbf{tot}}_{\mathbf{old}}, C^{\mathbf{max}}_{\mathbf{old}}) \prec (C^{\mathbf{tot}}_{+}, C^{\mathbf{max}}_{+})$ **then**
21 $\quad\quad\quad\quad \texttt{sol}[k + \ell] \leftarrow \texttt{sol}[k + \ell] \cup (C^{\text{tot}}_{+}, C^{\max}_{+}) \setminus \{(C^{\text{tot}}_{\text{old}}, C^{\max}_{\text{old}})\ :\ (C^{\text{tot}}_{+}, C^{\max}_{+}) \prec (C^{\text{tot}}_{\text{old}}, C^{\max}_{\text{old}})\}$;
22 **return** $\texttt{sol}[n]$

The third major step uses the route costs calculated above (as recorded in R) to incrementally construct full solutions. We use a table of partial solutions $\texttt{sol}$ indexed by $k \in [0..n]$ such that $\texttt{sol}[k]$ represents all non-dominated partial solutions that service all edges $e_1, e_2, \dots e_k$ (or that service nothing if $k = 0$). More technically, $\texttt{sol}[k]$ is a Pareto frontier of objective value pairs $(C^{\text{tot}}, C^{\max})$, each pair corresponding to a non-dominated partial solution. The main operation of this last step consists of expanding this set of partial solutions by incrementally inserting new routes: given a solution of $\texttt{sol}[k]$ that services $e_1, e_2, \dots e_k$ at Line 16, we use Lines 18-19 to insert a new route that services the edges $e_{k+1},\ e_{k+2},\ \dots e_{k+\ell}$ and that costs $R(e_{k+1}, \ell)$. If the resulting solution is not dominated by a solution that already exists in $\texttt{sol}[k+\ell]$ (see the **if** at Line 20), then this solution is added to $\texttt{sol}[k + \ell]$ at Line 21. At the same time, Line 21 removes all existing solutions $(C^{\text{tot}}_{\text{old}}, C^{\max}_{\text{old}})$ that are dominated by the new solution. Eventually, the last line returns the Pareto frontier $\texttt{sol}[n]$ which contains all non-dominated solutions that service all clients $[1..n]$.

The complexity of Algorithm 1 (mostly due to Step 3) depends linearly on n (at Line 15), on the maximum size of a Pareto frontier $\max\{|\texttt{sol}[k]| : k \in [1..n]\}$ (at Line 16) and on the maximum length of a route (Line 17). Finally, Appendix B provides an execution example for input permutation (e_1, e_2, e_3, e_4) and the instance from Figure 2.

Theorem 1. *Algorithm 1 generates all non-dominated solutions that service the clients in the order indicated by the input permutation (i.e., the decoder is exact).*

Proof. We can consider the input permutation is $(e_1, e_2, \ldots, e_n)$; this does not reduce generality, because all arguments below remain valid up to a reordering of the edges.

We first prove that $D^0(e_k, \ell)$ and $D^1(e_k, \ell)$ represent (the states associated to) the minimum-cost route that services $e_k, e_{k+1}, e_{k+2}, \ldots, e_{k+\ell-1}$ and finishes at the end point $e^0_{k+\ell-1}$ or resp., $e^1_{k+\ell-1}$ of edge $e_{k+\ell-1}$ ($\forall k \in [1..n]$). For $\ell = 1$, this is clearly true given how Lines 6–7 of Algorithm 1 simply compute these minimum-cost routes with only one serviced edge e_k. We prove by induction on ℓ that this property remains true for $\ell > 1$. What is the shortest path that services $e_k, e_{k+1}, \ldots, e_{k+\ell-1}$ and ends up at vertex $e^0_{k+\ell-1}$ associated to state $D^0(e_k, \ell)$? This vertex $e^0_{k+\ell-1}$ has to be reached after servicing the last edge $e_{k+\ell-1}$ coming from its other end vertex $e^1_{k+\ell-1}$; and this other end vertex can only be reached by extending a path associated to state $D^0(e_k, \ell-1)$ or to state $D^1(e_k, \ell-1)$. Both possibilities are covered by the recursion at Line 10, meaning that $D^0(e_k, \ell)$ correctly represent the shortest path that services $e_k, e_{k+1}, \ldots, e_{k+\ell-1}$ and ends up at $e^0_{k+\ell-1}$. An analogous argument can show the same property for $D^1(e_k, \ell)$ instead of $D^0(e_k, \ell)$.

We now show that $R(e_k, \ell)$ is the minimum-cost complete route that services all edges e_k, e_{k+1}, e_{k+2}, $\ldots$, $e_{k+\ell-1}$ in this order ($\forall k \in [1..n]$); since the route is complete in this case, it has to return to the depot in the end. And it can only return to the depot by extending a path associated to $D^0(e_k, \ell)$ or $D^1(e_k, \ell)$. Both possibilities are covered by Line 12.

We still have to prove that the last major step of Algorithm 1 generates all non-dominated Pareto optimal solutions. Assume for the sake of contradiction that there is a non-dominated solution that is not covered. The cost of such solution can be written under the form

$$R(e_1, \ell_1) + R(e_{1+\ell_1}, \ell_2) + R(e_{1+\ell_1+\ell_2}, \ell_3) + \cdots + R(e_{1+\ell_1+\ell_2+\cdots+\ell_{m-1}}, \ell_m), \tag{5}$$

where m is the number of routes in this full non-dominated solution, so that $\ell_1 + \ell_2 + \cdots + \ell_m = n$ (meaning that these m routes service n edges). Keep in mind that all R values from the above sum are correctly calculated by dynamic programming as described above. And recall that, in Step 3 of Algorithm 1, $\texttt{sol}[0], \texttt{sol}[1], \texttt{sol}[2], \ldots, \texttt{sol}[n]$ are meant to contain the sets of Pareto-optimal solutions that service the first 0, 1, 2, $\ldots$, n edges respectively. The hypothesis assumed for the sake of contradiction reduces to the fact that $\texttt{sol}[n]$ does not contain the solution associated to above sum (5).

If we restrict above sum (5) to the first $m' < m$ routes, we still obtain a complete non-dominated solution that only services the first $\ell_1 + \ell_2 + \cdots + \ell_{m'} < n$ edges. Let us now take m' to be the smallest value in $[1..m]$ such that $\texttt{sol}[\ell_1 + \ell_2 + \cdots + \ell_{m'}]$ *does not contain* the given optimal solution restricted to the first m' routes in (5). Since m' is minimal with this property, we can use that $\texttt{sol}[\ell_1 + \ell_2 + \cdots + \ell_{m'-1}]$ *does contain* the given optimal solution restricted to the first $m'-1$ routes. But since $R(e_{1+\ell_1+\ell_2+\cdots+\ell_{m'-1}}, \ell_{m'})$ is correctly computed, this optimal solution of $m'-1$ routes can naturally extend to the solution of m' routes by applying Lines 18-19 for $k = \ell_1 + \ell_2 + \cdots + \ell_{m'-1}$. Thus, there is no way Algorithm 1 could miss the solution of m' routes, which is a contradiction. The hypothesis assumed for the sake of contradiction that there is a full optimal solution not covered by Algorithm 1 has to be false.

Finally, the fact that $\texttt{sol}[n]$ contains only non-dominated solutions simply comes from the fact that $\texttt{sol}[0]$, $\texttt{sol}[1], \ldots$ $\texttt{sol}[n]$ represent by definition (and are implemented to record) Pareto frontiers of non-dominated solutions only. Combining this with the previous paragraph, $\texttt{sol}[n]$ has to contain exactly the list of non-dominated solutions of the given CARP instance subject to the service order imposed by the input permutation. $\square$

2.2 The Local Search phase

We consider the first objective C^{tot} to be more important than $C^{\max}$, and so, we here propose a Local Search (LS) algorithm that attempts to improve the C^{tot}–best solution returned by the decoder (from Section 2.1).

2.2.1 The neighborhood

We use two neighborhood relations associated to two operators (moves): route rotation and (unequal) block swap. All these moves can be executed in constant time but the number of potential positions on which they can be applied can be quadratic with regards to n.

Route rotation is a simple operator that acts on individual routes. Each route $\mathbf{r}$ is interpreted as a closed walk $v_0 \to v_1 \to v_2 \to \dots v_{|\mathbf{r}|} \to v_0$. Using a similar approach as in [Ulusoy, 1985, §3.3], one can re-locate the depot v_0 and place it before any index $i \in [2..|\mathbf{r}|]$ to obtain a new route $v_0 \to v_i \to v_{i+1} \to \dots v_{|\mathbf{r}|} \to v_1 \to v_2 \dots \to v_{i-1} \to v_0$. The cost variation resulting from this operation can be computed in constant time, because it only requires determining the cost of disconnecting and "re-linking" the connexion points $v_0, v_1,\ v_{i-1},\ v_i$ and v_r. When taking all routes into account, this operator can be applied on $O(n)$ positions.

Block swap is an operator that simply swaps two route blocks of arbitrary lengths; these blocks may belong to the same route or to different routes. Let us focus on the following two routes, using notation "$\rightsquigarrow$" to indicate a shortest path linking two vertices (with no service) and "$\rightarrow$" to indicate a serviced edge.

- $v_0 \dots \rightsquigarrow \underbrace{u_1 \to v_1}_{\text{edge 1}} \rightsquigarrow \underbrace{u_2 \to v_2}_{\text{edge 2}} \rightsquigarrow \underbrace{u_3 \to v_3}_{\text{edge 3}} \rightsquigarrow \dots v_0$
- $v_0 \dots \rightsquigarrow \underbrace{\overline{u}_1 \to \overline{v}_1}_{\text{edge 1}} \rightsquigarrow \underbrace{\overline{u}_2 \to \overline{v}_2}_{\text{edge 2}} \rightsquigarrow \dots v_0$

A block swap may simply take a block $u_{i+1} \to v_{i+1} \rightsquigarrow u_{i+2} \to v_{i+2} \dots \rightsquigarrow u_{i+\delta} \to v_{i+\delta}$ of the first route and swap it with a block $\overline{u}_{j+1} \to \overline{v}_{j+1} \rightsquigarrow \overline{u}_{j+2} \to \overline{v}_{j+2} \dots \rightsquigarrow \overline{u}_{j+\overline{\delta}} \to \overline{v}_{j+\overline{\delta}}$ of the second route. Notice that in the above example we have $\delta = 3$ and $\overline{\delta} = 2$ (and $i = j = 0$). For $\delta = \overline{\delta} = 1$, this is simply equivalent to swapping two required edges. One may also reverse one of the blocks or both of them when this improves the total cost. For any given $i,\ j,\ \delta$ and $\overline{\delta}$, the objective function variation can be calculated in constant time, because one only has to evaluate the cost evolution resulting from disconnecting the two blocks at their end points and re-connecting them at their new places. The capacity constraint can also be checked in constant time. The number of potential $i,\ j,\ \delta$ and $\overline{\delta}$ values on which this operator can be applied belongs to $O(n^2\delta^2_{\max})$, where δ_{max} is the maximum size of an existing block (we use $\delta_{\max} = 50$ in our experiments).

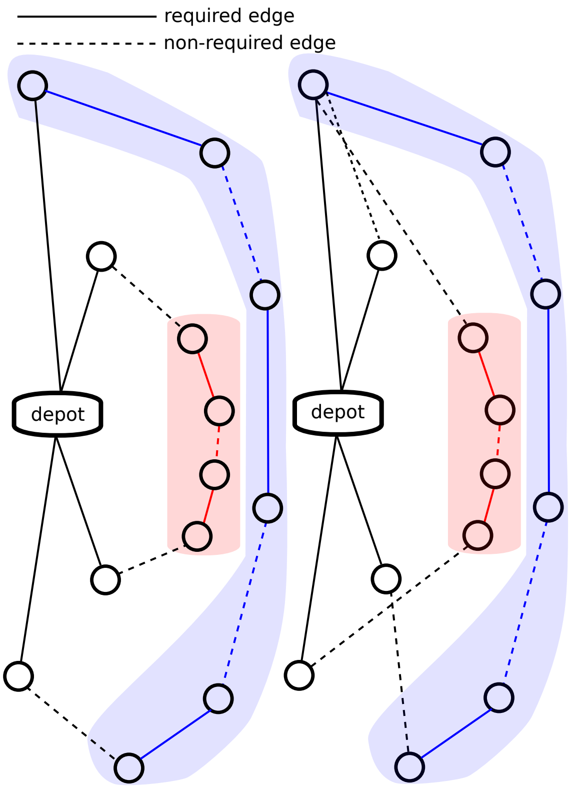


Figure 3: Block swap operator example. The red block (2 required edges) and the blue block (3 required edges) from the two routes at left are swapped from one route to another. We obtain the two routes at right, after having reconnected the two blocks to the edges emerging from the depot. Both resulting routes will pass through the top-left vertex and they are linked there as shown in the right figure.

The implementation of the above block swap operator was the most difficult programming task in the whole project; it is significantly more complex than implementing a route rotation or a block swap with

$\delta = \overline{\delta}$. Handling the right data structures to record routes of evolving length may be an elaborate task; an array-type data structure is not enough for this. However, preliminary experiments show that the effort pays off; this is a very aggressive tool for finding high-quality solutions. We tested several other neighborhoods before restricting to the presented ones, see also Section 4.2 for numerical tests on a block swap neighborhood with $\delta = \overline{\delta}$. We do not think we could have ever improved upon the best C^{tot} upper bound reported in the mono-objective literature without swapping blocks of arbitrary lengths as described above.

2.2.2 Complete specification of the LS algorithm

The LS algorithm consists of the following two stages described below in more detail.

strictly descent loop This first phase applies moves that strictly decrease the total cost, so as to search for local optima in the basin of attraction of the input solution (returned by the decoder).

unequal block swapping Perform `iters_ls`$= 5 + \left\lfloor \frac{n}{8} \right\rfloor$ iterations that swap unequal blocks, allowing side steps.

Stage 1 starts with the following operation: go through all pairs of edges and execute any edge swap (*i.e.*, a block swap with $\delta = \overline{\delta} = 1$) that strictly improves the total cost. This operation is repeated as long as there exists at least a swap that strictly improves the total cost, a bit like in a bubble search algorithm. One then performs all route rotations that lead to a better solution. The whole procedure in this stage is repeated as long as the solution reported in the end is strictly better than the starting solution.

Stage 2 executes `iters_ls` iterations of the following procedure: scan all pairs of blocks and perform all swaps that do not increase the C^{tot} cost. Recall it is also possible to reverse one of the blocks (or even both) when this improves the cost.[4] Neutral steps are thus accepted for the first time, which may lead the LS to solutions outside the basin of attraction of the starting solution. If at some iteration all possible block swaps would strictly increase the cost, we consider that the search is (almost) stuck in a local optimum. This calls for a small perturbation: go through all pairs of edges and swap them with a probability of 0.1, which amounts to eventually dislocating around 10% of the edges. This stage is not used for the very small instances ($n < 50$).

3 The decoder-based EA framework in the implicit space

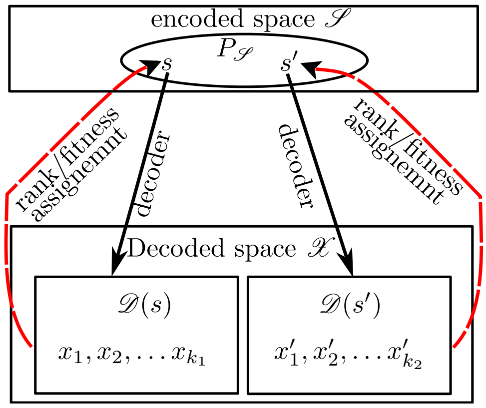
We here introduce a bi-objective EA in a decoder-based optimization framework, only characterized by the existence of an (exact) decoder that hides all problem-specific (CARP) features. A particular aspect in this context it that one has to associate multiple pairs of objective values to each genotype solution, *i.e.*, there are multiple 2D points in the objective space for each implicit solution. This brings certain challenges comparing to a standard bi-objective EA in which each genotype solution is associated to a unique point in the objective space. We will pay particular attention to two components that are very relevant in bi-objective optimization [Talbi, 2009]: (1) assigning fitness and ranks to (genotype) solutions and (2) keeping the diversity high at all levels to prevent a limited amount of gene patterns from monopolizing the population over too many generations.

Section 3.1 below addresses the first component: the fitness (and rank) assignment. Its goal is to compare and rank a set of genotype solutions (individuals) with conflicting objective values. The resulting ranking can be used for multiple purposes: to decide what individuals to accept in the population, what individuals need to be replaced and when to apply mutations. In our case, we will generalize the non-dominated sorting mechanism of NSGA, organizing the individuals into a hierarchy of fronts of different ranks.

[4] To reduce the running time, we make an exception: after swapping a block $[a, b]$ (*i.e.*, servicing all edges from index a to index b) with some block $[\overline{a}, \overline{b}]$, we do not allow the block $[a, b]$ to be swapped again with a block with $\overline{b} - \overline{a}$ edges or less.

3.1 Ranking implicit solutions using an extended non-dominated sorting

Recalling Definition 1, we consider a decoder function $\mathscr{D}$ that maps an implicit (genotype) solution $s \in \mathscr{S}$ to a set of decoded complete solutions $\mathscr{D}(s)$ from the explicit space $\mathscr{X}$. We have to design a method for ranking the implicit solutions of a population $P_{\mathscr{S}}$ based on the corresponding explicit solutions from $\mathscr{X}$ and their objective values. A bird's-eye view of this process is exemplified in the right figure. Considering a small population $P_{\mathscr{S}}$ with $|P_{\mathscr{S}}| = 2$, the genotype solutions $s, s' \in \mathscr{S}$ are decoded into sets $\mathscr{D}(s)$ and $\mathscr{D}(s')$. After calculating all objective values of all explicit solutions from these two sets, one has to assign ranks to s and s', *i.e.*, to decide which one is preferable.



We propose the following (fitness) evaluation approach. Given a population $P_{\mathscr{S}} \subset \mathscr{S}$, we first decode all elements of $P_{\mathscr{S}}$ and then compute their objective values. We thus obtain a number of 2-dimensional (2D) points in the objective space that all emerge from $P_{\mathscr{S}}$. These 2D points are then partitioned into a hierarchy of Pareto frontiers of different ranks (levels of (non-)domination) as in a pure NSGA2. More exactly, the front of rank 1 contains all 2D points that are completely non-dominated in the objective space. Then, the front of rank $k = 2, 3, \ldots$ contains all 2D points that are dominated only by 2D points of better rank (lower than k). We then define the (top) rank $\texttt{rank}_{\texttt{top}}(s)$ of a given $s \in P_{\mathscr{S}}$ as the minimum (best) rank value of a 2D point (in the objective space) that emerged from s.

Remark 1. *A second evaluation step is often needed, whenever one needs to distinguish between implicit solutions that have the same (top) rank. For each $s \in P_{\mathscr{S}}$, the top-rank size $\texttt{size}_{\texttt{top}}(s)$ is defined as the number of 2D points of rank $\texttt{rank}_{\texttt{top}}(s)$ that originate from s. This quality measure will be used in Section 3.4 (point 1.(b)) to perform a roulette wheel survival selection based on the idea that higher quality implicit solutions generate more top-rank 2D points.*

Figure 4 gives an example of the above evaluation process on a small population $P_{\mathscr{S}} = \{s, s', s''\}$. Notice that an implicit solution is always evaluated with regards to the whole population to which it belongs. Even if s'' is preferable to s in the given population, this is no longer true if we remove s' from $P_{\mathscr{S}}$ because the 2D points a and b would become non-dominated in the objective space.

3.2 The general design and overall pseudo-code

As hinted above, the *non-domination sorting* is an important concept used in NSGA2 [Deb et al., 2002] to organize the population into a hierarchy of fronts (sets) such that all solutions from a given front F_k dominate all solutions of inferior rank (from fronts $F_{k+1}, F_{k+2}, \ldots$); the top-right part of Figure 4 gives an example showing how a rank–1 front $\{c, d, f\}$ dominates a rank–2 front $\{a, b, e\}$. While this non-domination sorting is performed in the objective space, recall we extended it in Section 3.1 to evaluate solutions from the implicit space. The non-domination sorting idea is actually the only major NSGA2 feature that we use (and extend) in our work. For instance, unlike other CARP algorithms [Lacomme et al., 2006], we do not resort to the crowding distance metric which is used in NSGA2 to induce a preference for more isolated solutions along the Pareto frontier. In our case, whenever we have to distinguish between two solutions s and s' that belong to the same front (in the sense that $\texttt{rank}_{\texttt{top}}(s) = \texttt{rank}_{\texttt{top}}(s')$), we compare $\texttt{size}_{\texttt{top}}(s)$ and $\texttt{size}_{\texttt{top}}(s')$ as indicated in Remark 1 above.

Algorithm 2 presents the overall pseudo-code of the proposed $\texttt{Decoder-EA}$. At iteration $\texttt{it} = 1$, it starts with a genotype population $P^1_{\mathscr{S}}$ of $\texttt{pop}_{\texttt{size}}$ random implicit solutions which are all decoded (Line 2) into an explicit population $P^1_{\mathscr{X}}$. The outer `while` loop performs the following. First, it generates an offspring population $Q^{\texttt{it}}_{\mathscr{S}}$ of $\texttt{pop}_{\texttt{size}}$ implicit solutions by crossover at Line 5. The next line decodes these implicit solutions into explicit solutions followed by the LS call at Line 7. Recall (Section 2.2) that the LS may

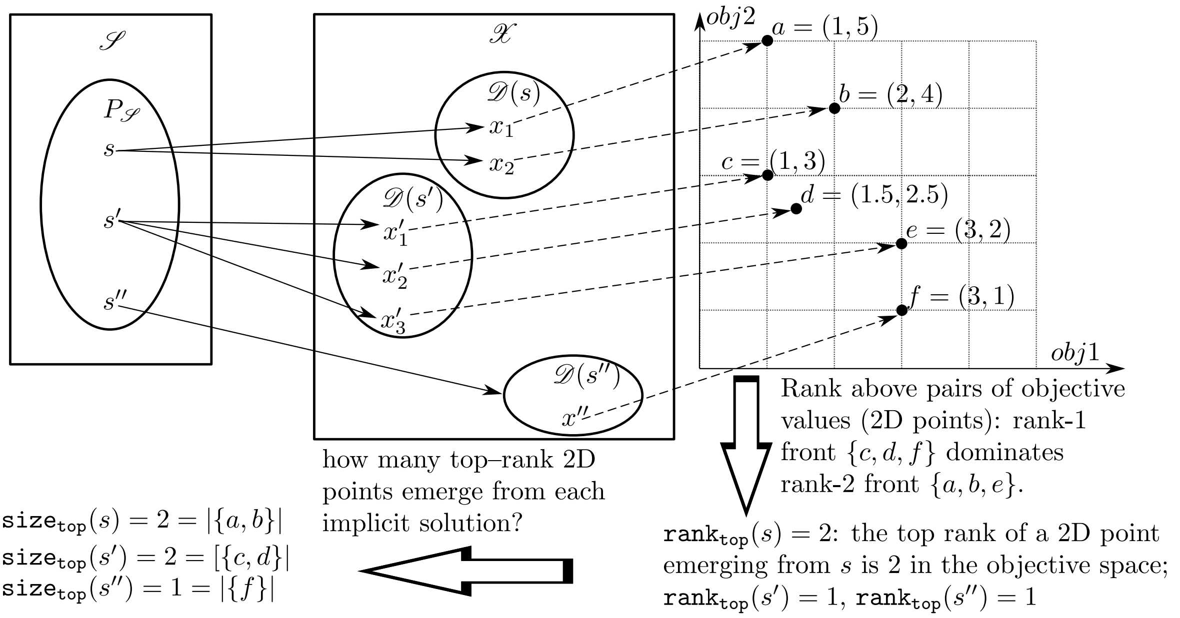


Figure 4: Illustration of our extended non-dominated sorting. The implicit solutions from population $P_{\mathscr{S}}$ are decoded into explicit solutions (solid arrows) which are then evaluated in the objective space (dashed arrows). The larger arrows illustrate the calculation of $\texttt{rank}_{\texttt{top}}(s)$ and $\texttt{size}_{\texttt{top}}(s)$ for all $s \in P_{\mathscr{S}}$.

Algorithm 2: `Decoder-EA`

1 $P^1_{\mathscr{S}} \leftarrow$ `RandomPop()`;
2 $P^1_{\mathscr{X}} \leftarrow$ `Decode(`$P^1_{\mathscr{S}}$`)`; *// each explicit solution has a label pointing to the implicit solution that generated it*
3 `it` $\leftarrow 1$;
4 **while** stopping criterion not met **do**
5 $\quad Q^{\texttt{it}}_{\mathscr{S}} \leftarrow$ `Crossover(`$P^{\texttt{it}}_{\mathscr{S}}$`)`;
6 $\quad Q^{\texttt{it}}_{\mathscr{X}} \leftarrow$ `Decode (`$Q^{\texttt{it}}_{\mathscr{S}}$`)`;
7 $\quad Q^{\texttt{it}}_{\mathscr{X}}, Q^{\texttt{it}}_{\mathscr{S}} \leftarrow$ `LocalSearch (`$Q^{\texttt{it}}_{\mathscr{X}}$`)`;
8 $\quad$ `updateFrontBestSols(`$Q^{\texttt{it}}_{\mathscr{X}}$`)`; *// the best non-dominated solutions ever generated*
9 $\quad P^{\texttt{it}+1}_{\mathscr{S}}, P^{\texttt{it}+1}_{\mathscr{X}} \leftarrow \{\}$;
10 $\quad$ `rank` $\leftarrow 1$;
11 $\quad$ **while** $|P^{\texttt{it}+1}_{\mathscr{S}}| < \texttt{pop}_{\texttt{size}}$ **do**
12 $\quad\quad F_{\mathscr{S}} \leftarrow$`getFront(`$P^{\texttt{it}}_{\mathscr{S}}, Q^{\texttt{it}}_{\mathscr{S}}, P^{\texttt{it}}_{\mathscr{X}}, Q^{\texttt{it}}_{\mathscr{X}}$`, rank)`; *// get only the implicit solutions for the current rank*
13 $\quad\quad P^{\texttt{it}+1}_{\mathscr{S}} \leftarrow P^{\texttt{it}+1}_{\mathscr{S}} \bigcup \{F_{\mathscr{S}}\}$;
14 $\quad\quad P^{\texttt{it}+1}_{\mathscr{X}} \leftarrow P^{\texttt{it}+1}_{\mathscr{X}} \bigcup$ `explicitSolutions(`$F_{\mathscr{S}}$`)`; *// add the explicit solutions from $P^{it}_{\mathscr{X}}$ associated to $F_{\mathscr{S}}$*
15 $\quad\quad$ `rank` $\leftarrow$ `rank` $+ 1$;
16 $\quad$ `it` $\leftarrow$ `it` $+ 1$;
17 **return** the Pareto optimal solutions constructed along the iterations via Line 8

change the implicit solution on which it is applied; this update is back propagated and may change $Q^{\texttt{it}}_{\mathscr{S}}$. Line 8 checks if the Pareto frontier of the best objective values ever generated may be enriched by some new offspring solution; this frontier will be returned by the last line of the algorithm.

The core of the overall algorithm is the inner **while** loop that constructs the next-generation populations $P^{\texttt{it}+1}_{\mathscr{S}}$ and $P^{\texttt{it}+1}_{\mathscr{X}}$. This construction uses the extended non-dominated sorting presented in Section 3.1. More exactly, the repeated call to `getFront(...)` at Line 12 retrieves one by one a sequence of fronts of implicit solutions that have an increasingly weaker rank; these fronts are iteratively added to the next-generation populations using Lines 13-14. If the size of the current front is larger than the number of remaining places in the new population (*i.e.*, $\texttt{pop}_{\texttt{size}} - \left|P_{\mathscr{S}}^{\texttt{it}+1}\right|$), then `getFront(...)` may actually return a reduced front. This may only happen at the last iteration of the inner **while** loop. Although all solutions s from this last front have the same $\texttt{rank}_{\texttt{top}}(s)$ value, they can be distinguished using their different $\texttt{size}_{\texttt{top}}(s)$ values; we use a roulette wheel selection to decide which solutions may survive (see point 1.(b) in Section 3.4 below).

As a side note, Algorithm 2 has to continuously maintain a link between the explicit solutions $P_{\mathscr{X}}^{\texttt{it}}$ and the implicit solutions $P_{\mathscr{S}}^{\texttt{it}}$. All routines that work with explicit solutions (*e.g.*, see the `LocalSearch` and `getFront` calls) have to be able to access the implicit solution associated to each explicit solution.

3.3 The crossover and the parent selection

After trying multiple ideas[5] we decided to only use the very simple one–point permutation crossover. This crossover simply takes the first $\lfloor \frac{n}{2} \rfloor$ positions of the first parent permutation and directly passes them to the offspring; we say $\lfloor \frac{n}{2} \rfloor$ is the split point. The remaining elements are inherited from the second parent in the order in which they appeared there. For example, the crossover of $[1, 2, 3, 4, 5, 6]$ and $[6, 4, 3, 2, 1, 5]$ would result in $[1, 2, 3, 6, 4, 5]$. We mention a small adaptation that is specific to Arc-Routing. After the decoder and the LS phase, we can easily identify the routes of the C^{tot}–best decoded solution and "pay attention" not to break such a route, especially if it is not very short. More exactly, if such a route covers an interval of indexes $[n/2 - \delta_1,\ n/2 + \delta_2]$ with $\delta_1 + \delta_2 > 5$, then the split point becomes $n/2 + \delta_2$ instead of $n/2$.

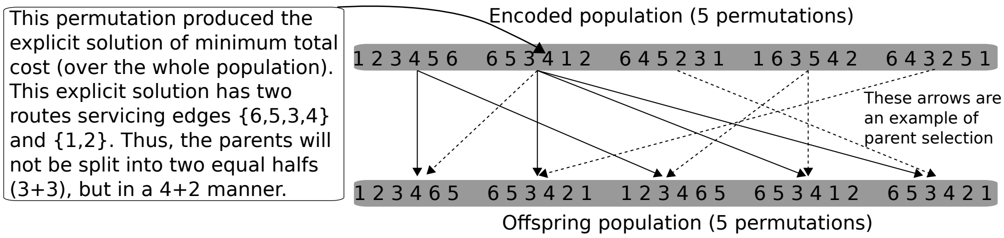


Figure 5: Example showing how the crossover would be applied on a population of size $\texttt{pop}_{\texttt{size}} = 5$. The first parent is indicated by solid arrows; the second one by dashed arrows. Each offspring inherits the first 4 elements from the first parent (the left legend explains the choice of the value 4 in this simplified setting); the remaining elements are inherited from the second parent in the order in which they appeared there.

The parent selection is performed as follows. Let $C_{\min}^{\text{tot}}(s)$ be the C^{tot}–best objective value of an explicit solution decoded from s, for any $s \in P_{\mathscr{S}}^{\texttt{it}}$. Let $C_{\text{worst}}^{\text{tot}}$ be the maximum value of $C_{\min}^{\text{tot}}(s)$ over all $s \in P_{\mathscr{S}}^{\texttt{it}}$, *i.e.*, $C_{\text{worst}}^{\text{tot}} = \max\left\{C_{\min}^{\text{tot}}(s) :\ s \in P_{\mathscr{S}}^{\texttt{it}}\right\}$. We would like to favor solutions s with a small $C_{\min}^{\text{tot}}(s)$ value, *i.e.*, s should be preferable to s' if $C_{\min}^{\text{tot}}(s) < C_{\min}^{\text{tot}}(s')$. We thus propose a parent selection based on a roulette wheel procedure that assigns to each $s \in P_{\mathscr{S}}^{\texttt{it}}$ a probability value proportional to $\left(C_{\text{worst}}^{\text{tot}} - C_{\min}^{\text{tot}}(s)\right)^2$. Notice that the solution s that has the worst $C_{\min}^{\text{tot}}(s)$ value (*i.e.*, $C_{\min}^{\text{tot}}(s) = C_{\text{worst}}^{\text{tot}}$) has zero chances of being selected.

3.4 Improving the population dynamics to avoid premature convergence

The above Algorithm 2 is actually the most general pseudo-code that can capture the main ideas of the overall method. To make `Decoder-EA` reach its full potential, one has to study (and improve) the general dynamics of the population, *i.e.*, to understand in more detail how it evolves over the generations. One can gain insight into such (sometimes tricky) aspects only after covering the previous more general algorithmic descriptions. This confirms that the devil is in the details: the current section *comes last but it nevertheless involves important or non-standard (research) material.* Without this material, we avow that our very first implementation of Algorithm 2 (with some naive calibration) was particularly prone to premature

[5] We did implement and try the following: a 2-point crossover, the "alternating edges" crossover, a grouping crossover popular in graph coloring. All these crossovers produced worse results; we think this is due to their stronger disruptive behaviour, *i.e.*, they can break too many links between consecutive edges constructed by previous evolution.

convergence, *i.e.*, all individuals could become almost identical in less than 100 iterations (generations). We tested many ideas to overcome such drawbacks; we finally implemented only the most effective ones that we present below.

1. To maintain a high-quality population in the long run, one has to be very careful in deciding what individuals may survive from one generation to the next; this has an important impact on selecting the gene patterns that survive over (many) generations. This (survival) selection is actually implemented inside the repeated call to `getFront(...)` in the inner **while** loop of Algorithm 2. This `getFront(...)` function first retrieves all individuals for the current rank and then it filters them by applying the following principles:

   (a) First and foremost, we need a protection against premature convergence, because otherwise a few high-quality individual refined over multiple generations could completely "shadow" all new offspring, *i.e.*, a few old individuals may achieve a quality level that is hardly ever reached by new individuals (which barely survive such selective pressure). To implement this protection, we never allow more than 30% of the current implicit population $P_{\mathscr{S}}$ to survive to the next generation; as such, at least 70% of the genetic material of each population has to come from recent offspring. Using $\texttt{pop}_{\texttt{size}} = 10$, we actually only allow the best 3 individuals to survive. Furthermore, `Decoder-EA` also checks that the surviving individuals are not always the same, which may happen if a few individuals reach such a high level of quality that no other individuals can compare to them. After five generations in which the surviving individuals are exactly the same, we select one of them to be artificially replaced (outlived) by a different random solution from the population. We can say that we implemented a policy that encourages young individuals instead of allowing a few old individuals to dominate the genetic material in the long run.

   (b) If the number of remaining positions in the new population (*i.e.*, $\texttt{pop}_{\texttt{size}} - \left|P_{\mathscr{S}}^{\texttt{it}+1}\right|$) becomes smaller than the number of individuals of rank `rank`, these individuals can not all survive. Recalling Remark 1 from Section 3.1, we formally say that all these individuals (solutions) s have same top-rank value $\texttt{rank}_{\texttt{top}}(s) = \texttt{rank}$, but they may have different top-rank sizes $\texttt{size}_{\texttt{top}}(s)$. In this case, `getFront(...)` performs a selection by applying a roulette wheel procedure in which the survival probability of each considered solution s is proportional to $\left(\texttt{size}_{\texttt{top}}(s)\right)^2$.

2. We also need a (re-)diversification operation that we use after many generations with no progress. `Decoder-EA` can detect such an undesirable state quite easily by monitoring the evolution of the Pareto frontier of the best objective values ever discovered. Recall that, at each generation, Line 8 attempts to improve this Pareto frontier by trying to insert the objective values of the new offspring. After 2000 generations with no new insertion at Line 8, `Decoder-EA` performs a (re-)diversification operation, so as to try to make the population escape from the basin of attraction of the current best implicit solutions. We first mark all implicit solutions $s \in P_{\mathscr{S}}^{\texttt{it}}$ such that the C^{tot}–best cost of an explicit solution decoded from s is within 110% of the C^{tot}–best cost ever reported since the last re-diversification. We then eliminate (replace with random solutions) the following: (i) all marked implicit solutions s, and (ii) all other solutions $\overline{s} \in P_{\mathscr{S}}^{\texttt{it}}$ that are very close to a marked implicit solution s.[6]

3. Finally, we need to monitor the potential appearance of duplicated individuals. Each time a new individual is generated (by crossover followed by LS), we check if the current population does not already contain an identical individual. We allow an individual to be duplicated once; this may be acceptable because it may lead the population towards a stronger basin of attraction where more high-quality gene patterns may be found. But if we detect that the last generated individual already exists at least twice in the population, we apply a perturbation before inserting it.[7]

[6] By "very close", we mean that the distance between s and $\overline{s}$ is less than $\frac{n}{10}$. The distance between two permutations is here given by the number of consecutive edges from the first permutation that do *not* arise in the second one. For example, consider permutations $[1, 2, 3, 4, 5]$ and $[5, 3, 4, 1, 2]$. The consecutive edges in the first permutation are: (1,2), (2,3), (3, 4), (4, 5) and (5,1). Two of these pairs arise as consecutive edges in the second permutations, *i.e.*, (3, 4) and (1,2). The remaining three pairs do not arise in the second permutation, hence the distance between the two permutations is 3.

[7] This perturbation scans all pairs of edges and swaps them with a 10% probability, followed by applying the LS from

4 Numerical Results on Arc-Routing

We now perform an evaluation of `Decoder-EA` over all CARP instances that we are aware of. They originate from six different benchmark sets, distinguished by their different prefix: `gdb`, `kshs`, `val`, `egl`, `C–F` or `g`. The average value of $n = |E_R|$ is around 80; there are 17 instances with $n > 100$ and 10 with $n > 300$. All these instances are publicly available on-line at `www.uv.es/~belengue/carp.html`, at `https://logistik.bwl.uni-mainz.de/forschung/benchmarks/` or at `cedric.cnam.fr/~porumbed/carpbest/`. We present below their characteristics in greater detail.

`gdb` These 23 instances have between 7 and 27 vertices and a number of edges $n \in [11, 55]$ all required. In fact, there is only one instance with $n = 11$, the rest having $n \geq 19$.

`kshs` These 6 small instances have between 6 and 10 vertices and exactly $n = 15$ edges, all required.

`val` These 34 moderate-size instances have between 24 and 50 vertices and they have a number edges $n \in [34, 97]$, all required.

`egl` These 24 larger instances were generated in connexion to a winter gritting application in Lancashire in the 1990s. Half of them have 77 vertices and half of them have 140 vertices; 18 instances have non-required edges. The number of required edges n ranges from 51 to 190.

`C–F` This is a large set of 100 instances that have between 26 and 97 vertices and the number of required edges n ranges from 28 to 121; non-required edges are present in all instances.

`g` This set contain 10 very large instances, all of them with 255 vertices. There are five `g1` instances with $n = 347$ and with 28 non-required edges; a second sub-set `g2` has $n = 375$ required edges and zero non-required edges. These 10 instances are also referred to as `egl-large`.

The code source was implemented in `C++` and compiled by `g++ -O3` under OpenSuse 15.1 Linux (kernel version 5.4.14-9). All reported CPU times have been obtained on an Intel Xeon Gold 5218 processor clocked at 2.30GHz.

4.1 Complete results over 100, 1000 and 10000 iterations on all instances

We generally allow `max_iters` = 10000 to solve each instance. For each run, we will actually present the results reported by `Decoder-EA` after 100, 1000 and 10000 iterations, roughly corresponding to a short, medium and resp. long term evaluation. In fact, the maximum number of iterations `max_iters` = 10000 can even be increased (a bit) in some exceptional cases. Following an idea from [Porumbel et al., 2017], if `Decoder-EA` discovers a new non-dominated solution during the last 10% iterations (*e.g.*, after iteration 9000), the value of `max_iters` is increased by 10% (*i.e.*, first from 10000 to 11000, than to 12100, etc). We never allow the number of iterations to exceed 15000 using this mechanism.

Table 1 reports the results obtained by applying the following protocol on each instance: execute `Decoder-EA` ten times, rank the ten results (as described just next) and finally report the run of `median rank`. The ranks of the ten considered runs are defined by sorting the reported results according the left-most solution (the one of minimum C^{tot} value) of the Pareto returned in the end. Two runs that report the same left-most solution in the end are considered to have an equal (equivalent) rank. The rank in Column 7 is obtained from this sorting. The most frequent rank is 1-10/10 and it corresponds to a case in which all runs report equivalent results at the end of all allowed iterations. A rank of 3-5/10 indicates a tie for positions three through five. We say that all such runs have a median rank, because they cover the 5th place.

The columns of Table 1 are organized as follows.

– The first two columns report the instance name and the number of required edges $n = |E_R|$.

Section 2.2.2. If this still leads to a solution that already exists in the population, we repeat the perturbation by increasing the probability to 20%, than to 40%, 80% and eventually we generate a random individual if necessary.

– Columns 3 through 7 describe the results a full run at the end of all allowed iterations. Column 3 and 4 indicate the left-most and resp right-most solution of the final Pareto frontier in the format "$C^{\texttt{tot}}/C^{\texttt{max}}$". Column 5 represents the total CPU time in seconds. Column 6 reports a hyper-volume indicator.[8] Column 7 is the rank of the reported run with regards to the ten runs, as described above.

– Columns 8–9 indicate the left-most and resp right-most solution of the final Pareto frontier obtained after 1000 iterations.

– Columns 10–11 report the same results as in Columns 8-9, but obtained after only 100 iterations.

– The last column reports the best leftmost (C^{tot}-best) solution ever discovered by any of the ten runs. When (a cell in) this column is empty, this means that the best discovered solution is the one from the reported run in Column 3.

These results in Table 1 are rather self-explanatory. Notice that Column 7 frequently reports a rank of "1-10/10" which indicates that all ten runs returned the same left-most solution at the end of all allowed iterations. Such instances are not very sensitive to many changes in the design of `Decoder-EA`. The final `Decoder-EA` variant described throughout this paper was designed by performing various performance tests on more critical (or variation-inducing) instances; we will focus on such instances in Sections 4.2–4.3.

Remark 2. *Whenever a value is marked in boldface in the last column of Table1, this indicates we improved upon the best known C^{tot} upper bound ever reported in the (much) larger mono-objective CARP literature. We here report the previous and the new best-known upper bounds for the concerned nine instances:*

Instance	*Previous best upper bound*	*New upper bound*	*Instance*	*Previous best upper bound*	*New upper bound*
`egl-s2-A`	*9884*	*9875*	`egl-s2-B`	*13099*	*13065*
`egl-s4-B`	*16260*	*16198*			
`g1-A`	*998777*	*995012*	`g1-C`	*1241762*	*1240426*
`g1-D`	*1371443*	*1370958*	`g1-E`	*1512584*	*1512391*
`g2-C`	*1341519*	*1340303*	`g2-D`	*1481181*	*1480726*

The previous best-known upper bound for the first above instance (9884 for `egl-s2-A`*) was discovered in [Santos et al., 2010]. For the remaining instances, the best-known upper bounds were retrieved from (Table IX of) [Mei et al., 2014], where the authors stated that their bounds improved upon the previous best-known solutions. All these top results seem consistent when comparing to the results of the four heuristics (or resp. eight in the case of egl-s2-A, elg-s2-B and elg-s4-B) reported at https://logistik.bwl.uni-mainz.de/forschung/benchmarks/. All our new best solutions are publicly available on-line at cedric.cnam.fr/~porumbed/carpbest/ for any further improvement or research use.*

Besides these heuristic results, we could also use the decoder to exactly determine the optimal solutions for the smallest instance `gdb19` with $n = 11$. For this value of n, we simply called the decoder on all 11! permutations, *i.e.*, we evaluated the whole search space. The solutions of the optimal Pareto frontier are: $(83, 17)$, $(71, 19)$, $(63, 20)$ and $(55, 21)$, see also the figures in Appendix A. The total computation time was about 2 hours. Had we used a more classical "split with flips" decoder, the search space size would have been $2^{11} = 2048$ times larger, easily leading to a thousand-fold increase of the computation time. This may explain why we are the first to report the exact optimal Pareto frontier for a well-acknowledged instance.

[8] It was calculated with regards to the following reference point: $(1.05 \cdot C^{\text{tot}}_{\texttt{left}} + 1, 1.05 \cdot C^{\text{max}}_{\texttt{right}} + 1)$, where $C^{\text{tot}}_{\texttt{left}}$ is the C^{tot} cost of the solution from Column 3 and $C^{\text{max}}_{\texttt{right}}$ is the C^{max} cost (makespan) of the solution from Column 4.

Table 1: Detailed results over all instances

		10000 iterations					1000 iterations		100 iterations		best left sol
Instance	n	left	right	time[s]	hyper-vol	rank	left	right	left	right	ten runs
gdb01	22	316/74	337/63	25	451	1-10/10	316/74	337/63	316/74	337/63	/
gdb02	26	339/69	395/59	42	954	1-10/10	339/70	395/59	339/73	413/59	/
gdb03	22	275/65	339/59	29	688	1-10/10	275/65	339/59	275/65	346/59	/
gdb04	19	287/74	350/64	18	768	1-10/10	287/74	350/64	287/74	350/64	/
gdb05	26	377/78	447/64	35	1278	1-10/10	377/78	447/64	377/83	447/64	/
gdb06	22	298/75	351/64	27	760	1-10/10	298/75	351/64	298/75	351/64	/
gdb07	22	325/68	415/57	24	1095	1-10/10	325/68	415/57	325/69	415/57	/
gdb08	46	348/48	396/38	126	754	1-6/10	353/45	406/38	358/44	426/38	/
gdb09	51	303/43	333/37	357	339	1-10/10	303/49	335/37	306/43	335/37	/
gdb10	25	275/70	410/39	49	4272	1-10/10	275/71	410/39	275/76	410/39	/
gdb11	45	395/82	585/43	445	7767	1-10/10	395/87	585/43	399/93	591/43	/
gdb12	23	458/97	547/93	19	768	1-10/10	458/97	557/93	458/97	583/93	/
gdb13	28	544/128	544/128	27	196	2-10/10	544/128	544/128	544/128	544/128	536/140
gdb14	21	100/21	136/15	25	272	1-10/10	100/21	136/15	100/21	136/15	/
gdb15	21	58/15	68/8	29	78	1-10/10	58/15	68/8	58/16	70/8	/
gdb16	28	127/27	151/14	47	382	1-10/10	127/29	151/14	129/24	157/14	127/26
gdb17	28	91/14	101/9	49	68	1-7/10	91/14	103/9	91/15	103/9	/
gdb18	36	164/33	232/19	135	948	1-6/10	164/34	232/19	164/34	250/19	/
gdb19	11	55/21	63/17	6	40	1-10/10	55/21	63/17	55/21	63/17	/
gdb20	22	121/34	131/20	26	224	1-10/10	121/36	131/20	123/26	133/20	/
gdb21	33	156/28	194/15	73	570	1-8/10	156/29	194/15	158/27	198/15	/
gdb22	44	200/28	240/12	138	747	1-6/10	200/30	242/12	202/28	244/12	/
gdb23	55	235/25	279/13	520	627	2-7/10	235/28	281/13	235/28	287/13	233/26
kshs1	15	14661/4117	16825/3528	10	1716726	1-10/10	14661/4117	16825/3528	14661/4117	16825/3528	/
kshs2	15	9863/2646	13601/2109	11	1919208	1-10/10	9863/2646	13601/2109	9863/2646	13601/2109	/
kshs3	15	9320/2640	9666/2084	10	395567	1-10/10	9320/2640	9666/2084	9320/2670	9666/2084	/
kshs4	15	11498/3349	12964/2713	12	1007131	1-10/10	11498/3349	12964/2713	11498/3349	12964/2713	/
kshs5	15	10957/4195	15857/2865	14	6980854	1-10/10	10957/4195	15857/2865	10957/4746	15857/2865	/
kshs6	15	10197/4032	11177/3472	13	851510	1-10/10	10197/4032	11177/3472	10197/4032	11177/3472	/
val1A	39	173/58	216/40	276	865	1-10/10	173/58	216/40	173/61	230/40	/
val1B	39	173/60	216/40	194	929	5-7/10	173/59	216/40	179/52	216/40	173/59
val1C	39	245/41	248/40	84	61	1-10/10	245/41	248/40	245/42	248/40	/
val2A	34	227/114	395/71	224	6431	1-10/10	227/114	395/71	227/114	397/71	/
val2B	34	259/108	395/71	157	5338	1-10/10	259/108	401/71	260/101	407/71	/
val2C	34	457/71	457/71	54	92	1-10/10	457/71	457/71	462/71	462/71	/
val3A	35	81/41	106/27	252	375	1-10/10	81/41	114/27	81/41	120/27	/
val3B	35	87/32	106/27	130	109	1-10/10	87/32	106/27	87/32	106/27	/
val3C	35	138/27	138/27	52	14	1-10/10	138/27	138/27	138/27	138/27	/
val4A	69	400/134	514/80	7839	7493	1-10/10	400/134	578/80	400/134	578/80	/
val4B	69	412/104	530/80	3424	3126	1-10/10	412/106	562/80	412/108	573/80	/
val4C	69	428/99	496/80	1992	2224	1-10/10	428/99	534/80	428/100	555/80	/
val4D	69	530/82	538/80	1075	234	1-10/10	530/83	542/80	530/85	578/80	/
val5A	65	423/141	715/72	698 1	20759	1-10/10	423/141	722/72	423/142	785/72	/
val5B	65	446/112	718/72	4258	10933	1-10/10	446/113	762/72	446/115	774/72	/

Continued on next page

Table 1: Detailed results over all instances (continued)

		10000 iterations					1000 iterations		100 iterations		best left sol
Instance	n	left	right	time[s]	hyper-vol	rank	left	right	left	right	ten runs
val5C	65	474/95	725/72	2321	6094	1-10/10	474/97	737/72	474/99	774/72	/
val5D	65	575/90	697/72	995	3268	2-5/10	578/89	705/72	581/84	717/72	575/83
val6A	50	223/75	309/45	1717	2350	1-10/10	223/75	309/45	223/75	311/45	/
val6B	50	233/68	307/45	948	1856	1-10/10	233/68	315/45	233/68	319/45	/
val6C	50	317/54	323/45	297	242	1-10/10	317/55	329/45	317/55	343/45	/
val7A	66	279/85	385/39	4020	5443	1-10/10	279/85	391/39	279/85	401/39	/
val7B	66	283/58	385/39	2703	1890	1-10/10	283/58	385/39	283/61	403/39	/
val7C	66	334/50	387/39	946	839	1-10/10	334/50	405/39	334/50	411/39	/
val8A	63	386/129	635/67	4252	15741	1-10/10	386/129	657/67	386/129	695/67	/
val8B	63	395/99	623/67	4123	7401	1-5/10	395/101	655/67	395/105	680/67	/
val8C	63	521/85	635/67	849	2790	3-5/10	527/81	645/67	527/81	671/67	521/77
val9A	92	323/108	459/44	26676	9421	1-10/10	323/109	459/44	323/114	480/44	/
val9B	92	326/82	454/44	17292	5090	1-10/10	326/84	471/44	326/86	484/44	/
val9C	92	332/67	457/44	9374	2923	1-5/10	332/68	470/44	332/70	494/44	/
val9D	92	391/50	423/44	3361	306	1-10/10	391/51	439/44	393/52	455/44	391/49
val10A	97	428/143	742/47	37684	31348	1-10/10	428/143	781/47	428/145	787/47	/
val10B	97	436/109	751/47	24660	18955	1-10/10	436/110	783/47	436/111	789/47	/
val10C	97	446/90	755/47	14046	12663	1-10/10	446/91	768/47	446/95	790/47	/
val10D	97	526/64	751/47	4882	4310	1-10/10	526/74	769/47	528/69	772/47	526/63
egl-e1-A	51	3548/943	3889/820	633	77319	1-10/10	3548/943	4159/820	3548/943	4159/820	/
egl-e1-B	51	4498/899	4540/820	436	31161	1-10/10	4525/839	4615/820	4525/839	4973/820	/
egl-e1-C	51	5595/836	5659/820	345	19328	1-10/10	5595/836	5699/820	5595/836	5719/820	/
egl-e2-A	72	5018/953	6164/820	1525	225869	1-10/10	5018/953	6360/820	5018/953	6424/820	/
egl-e2-B	72	6321/870	7140/820	1410	105197	1-10/10	6344/871	7369/820	6352/871	7456/820	6317/878
egl-e2-C	72	8335/854	8583/820	846	45858	1-10/10	8354/854	8763/820	8354/854	8884/820	/
egl-e3-A	87	5898/929	7981/820	4385	317300	1-10/10	5898/929	8183/820	5898/929	8409/820	/
egl-e3-B	87	7777/872	9289/820	1754	149063	5-10/10	7777/872	9416/820	7801/872	9895/820	7775/872
egl-e3-C	87	10292/927	10495/820	1087	129013	1-6/10	10292/927	10746/820	10361/927	11222/820	/
egl-e4-A	98	6461/929	8079/820	3732	257491	1-10/10	6464/929	8597/820	6476/929	8597/820	/
egl-e4-B	98	9005/914	10126/820	2415	190411	5-6/10	9041/946	10264/820	9078/930	10633/820	8999/914
egl-e4-C	98	11614/872	11626/820	1541	56500	1-10/10	11624/872	12074/820	11688/872	12313/820	11594/872
egl-s1-A	75	5018/1023	7761/912	2248	395446	1-10/10	5018/1023	7849/912	5050/1023	7946/912	/
egl-s1-B	75	6388/984	8379/912	2567	233879	1-10/10	6435/984	8455/912	6435/984	8759/912	/
egl-s1-C	75	8518/1018	9571/912	863	235424	1-10/10	8518/1018	9824/912	8518/1018	10154/912	/
egl-s2-A	147	9915/1084	13017/979	13544	479242	4-6/10	10055/1075	13550/979	10055/1075	13715/979	**9875**/1083
egl-s2-B	147	13162/1060	15167/979	7893	327341	2-10/10	13283/1060	15468/979	13290/1060	15940/979	**13065**/1060
egl-s2-C	147	16547/1040	17549/979	3887	202535	5/10	16611/1040	17698/979	16633/1040	17780/979	16425/1040
egl-s3-A	159	10244/1099	13745/979	30651	621389	5/10	10314/1099	13747/979	10356/1192	13783/979	10233/1099
egl-s3-B	159	13704/1060	15768/979	11079	354833	5/10	13780/1060	15919/979	13822/1060	16088/979	13682/1060
egl-s3-C	159	17228/1040	18462/979	8319	208331	5/10	17341/1040	18472/979	17555/1040	18472/979	17223/1040
egl-s4-A	190	12351/1103	12793/1027	21318	129756	5/10	12442/1060	13017/1027	12476/1120	13464/1027	12315/1080
egl-s4-B	190	16323/1027	16322/1027	18055	42484	5/10	16430/1027	16430/1027	16430/1027	16430/1027	**16198**/1027
egl-s4-C	190	20648/1035	20696/1027	8826	62716	4-6/10	20725/1035	20731/1027	20890/1027	20890/1027	20614/1037
C01	79	4150/610	4230/585	1920	14752	1-10/10	4195/600	4230/585	4215/630	4240/585	/

Continued on next page

Table 1: Detailed results over all instances (continued)

		10000 iterations					1000 iterations		100 iterations		best left sol
Instance	n	left	right	time[s]	hyper-vol	rank	left	right	left	right	ten runs
C02	53	3135/515	3135/515	460	4082	1-10/10	3135/515	3135/515	3135/570	3140/515	/
C03	51	2575/575	2585/490	638	15135	1-10/10	2575/575	2585/490	2585/490	2585/490	/
C04	72	3510/620	3955/590	1484	26516	1-10/10	3510/620	3955/590	3510/620	3985/590	/
C05	65	5365/715	6510/595	829	194332	1-10/10	5365/755	6535/595	5370/705	6765/595	/
C06	51	2535/540	2935/460	584	46126	1-10/10	2535/540	2935/460	2550/505	2935/460	/
C07	52	4075/610	4075/610	380	6324	1-10/10	4075/610	4075/610	4075/660	4125/610	/
C08	63	4090/750	4495/600	607	111286	1-10/10	4100/670	4535/600	4100/670	4535/600	/
C09	97	5260/525	5650/505	3716	27056	1-10/10	5270/555	5650/505	5300/535	5800/505	/
C10	55	4700/710	4885/640	499	43355	1-10/10	4700/710	4915/640	4700/710	5005/640	/
C11	94	4640/590	5285/510	4953	71400	1-10/10	4640/590	5435/510	4640/605	5435/510	/
C12	72	4240/645	4260/575	952	22702	1-10/10	4240/645	4260/575	4240/645	4260/575	/
C13	52	2955/695	3750/520	667	176585	1-10/10	2955/695	3830/520	2955/695	3860/520	/
C14	57	4030/695	5535/520	600	305305	1-10/10	4030/695	5625/520	4060/695	5645/520	/
C15	107	4940/645	5200/640	4683	18663	1-10/10	4945/645	5215/640	4990/645	5260/640	/
C16	32	1475/500	1660/480	73	10324	1-10/10	1475/500	1660/480	1510/500	1660/480	/
C17	42	3555/665	5010/540	116	216704	1-10/10	3575/665	5010/540	3650/615	5010/540	/
C18	121	5620/650	5620/650	7560	9306	1-10/10	5625/655	5645/650	5625/655	5675/650	/
C19	61	3115/705	3645/560	1471	109714	1-10/10	3115/705	3710/560	3120/705	3975/560	/
C20	53	2120/515	2500/410	683	54236	1-10/10	2120/515	2640/410	2120/515	2670/410	/
C21	76	3970/685	3970/685	1526	6965	1-10/10	3970/685	3970/685	3970/685	3970/685	/
C22	43	2245/660	3440/550	262	155942	1-10/10	2245/660	3605/550	2245/660	3605/550	/
C23	92	4085/755	4210/695	4590	30028	1-10/10	4085/755	4275/695	4095/755	4410/695	/
C24	84	3400/655	4110/575	2680	97094	1-10/10	3400/655	4310/575	3410/685	4350/575	/
C25	38	2310/560	2490/475	120	25070	1-10/10	2310/560	2490/475	2345/560	2540/475	/
D01	79	3215/840	3870/585	5162	218802	1-10/10	3235/685	3870/585	3235/690	3915/585	/
D02	53	2520/695	3040/515	1301	105070	1-10/10	2520/695	3065/515	2520/695	3115/515	/
D03	51	2065/710	2605/490	1218	152630	1-10/10	2065/710	2695/490	2065/710	2815/490	/
D04	72	2785/750	3825/590	4996	176011	1-10/10	2785/750	3825/590	2785/750	3865/590	/
D05	65	3935/820	5795/595	2256	420500	1-10/10	3935/820	5795/595	3935/820	6115/595	/
D06	51	2125/785	3030/460	1640	318005	1-10/10	2125/785	3045/460	2125/785	3110/460	/
D07	52	3125/880	3775/610	1546	208335	2-10/10	3125/910	3815/610	3165/830	3875/610	3115/960
D08	63	3045/800	4285/600	1781	252655	1-10/10	3045/810	4285/600	3045/830	4285/600	/
D09	97	4120/695	5650/505	11578	304725	1-10/10	4120/700	5745/505	4120/720	5785/505	/
D10	55	3340/760	3880/640	1278	74765	1-10/10	3340/760	3880/640	3340/765	3880/640	/
D11	94	3745/820	5365/510	16496	522637	1-10/10	3760/790	5495/510	3760/790	5580/510	/
D12	72	3310/760	3845/575	4664	117522	1-10/10	3310/760	3905/575	3310/760	4105/575	/
D13	52	2535/825	3770/520	1825	390178	1-10/10	2535/840	3790/520	2540/780	3790/520	/
D14	57	3280/840	5505/520	1256	720511	1-10/10	3280/840	5605/520	3280/845	5635/520	/
D15	107	3990/845	4615/640	12473	190079	1-10/10	3990/920	4690/640	4000/755	4715/640	/
D16	32	1060/535	1480/480	270	28890	1-10/10	1060/535	1500/480	1060/535	1505/480	/
D17	42	2620/710	4920/540	252	378182	1-10/10	2620/710	4920/540	2620/710	4920/540	/
D18	121	4165/800	4840/650	25810	131418	1-10/10	4165/800	5140/650	4165/805	5235/650	/
D19	61	2400/875	3715/560	2325	440492	1-10/10	2400/875	3845/560	2400/875	3935/560	/
D20	53	1870/705	2530/410	2967	204497	1-10/10	1870/705	2670/410	1870/705	2700/410	/

Continued on next page

Table 1: Detailed results over all instances (continued)

		10000 iterations					1000 iterations		100 iterations		best left sol
Instance	n	left	right	time[s]	hyper-vol	rank	left	right	left	right	ten runs
D21	76	3055/825	3410/685	5323	77882	1-10/10	3055/855	3410/685	3060/860	3640/685	/
D22	43	1865/910	3420/550	690	597422	1-10/10	1865/910	3575/550	1865/910	3685/550	/
D23	92	3130/845	4035/695	8850	175004	1-10/10	3130/890	4115/695	3135/905	4115/695	/
D24	84	2710/855	4215/575	12870	452461	1-10/10	2710/855	4305/575	2710/855	4345/575	/
D25	38	1815/760	2490/475	223	181550	1-10/10	1815/760	2520/475	1815/760	2660/475	/
E01	85	4910/655	5150/585	2621	48827	1-6/10	4950/600	5190/585	4960/605	5260/585	/
E02	58	3990/605	4485/500	702	72846	1-10/10	3990/630	4560/500	4110/610	4655/500	/
E03	47	2015/620	2635/420	348	153164	1-5/10	2025/490	2635/420	2025/490	2645/420	/
E04	77	4155/625	4405/590	2159	27917	1-10/10	4165/625	4775/590	4205/665	4905/590	/
E05	61	4585/665	5765/595	1746	122621	1-5/10	4595/695	5860/595	4595/700	5870/595	/
E06	43	2055/500	2690/420	203	49670	1-10/10	2055/500	2820/420	2055/500	2880/420	/
E07	50	4155/660	4325/600	322	34852	1-10/10	4155/660	4415/600	4155/660	4465/600	/
E08	59	4710/670	5710/600	490	91528	1-10/10	4710/670	5730/600	4715/670	5760/600	/
E09	103	5820/590	7535/500	4943	177400	2-9/10	5830/595	7595/500	5955/635	7735/500	5810/590
E10	49	3605/680	4610/540	250	162775	1-10/10	3605/695	4730/540	3605/695	4730/540	/
E11	94	4670/570	4790/560	3836	12940	1-10/10	4720/570	4980/560	4720/590	5005/560	/
E12	67	4185/660	4685/570	1478	61165	3-10/10	4205/660	4685/570	4230/660	4895/570	4180/660
E13	52	3345/695	4215/520	428	181985	1-10/10	3345/695	4245/520	3345/695	4245/520	/
E14	55	4115/735	5650/520	482	405879	1-10/10	4115/735	5740/520	4135/685	5740/520	/
E15	107	4205/645	4375/640	7193	15696	1-10/10	4225/645	4435/640	4235/655	4540/640	/
E16	54	3775/640	3835/630	580	10386	1-10/10	3805/635	3835/630	3805/640	3860/630	/
E17	36	2740/700	4305/540	76	284226	1-10/10	2755/670	4305/540	2755/670	4395/540	/
E18	88	3835/635	4335/580	3272	37629	1-10/10	3835/635	4335/580	3835/635	4335/580	/
E19	66	3235/650	4115/530	1799	123583	1-10/10	3235/705	4145/530	3235/705	4155/530	/
E20	63	2825/465	3100/455	1429	12989	1-10/10	2825/465	3165/455	2825/465	3250/455	/
E21	72	3730/705	4680/570	3485	174481	1-10/10	3755/735	4790/570	3795/705	4880/570	/
E22	44	2470/685	3125/555	197	106605	1-10/10	2480/675	3125/555	2485/675	3230/555	/
E23	89	3710/635	4340/535	5270	92286	1-10/10	3710/640	4370/535	3710/640	4370/535	/
E24	86	4020/590	4400/580	4453	22265	1-10/10	4020/600	4425/580	4020/665	4540/580	/
E25	28	1615/565	2075/485	48	45775	1-10/10	1615/565	2085/485	1615/565	2115/485	/
F01	85	4040/825	4970/585	8260	263480	1-10/10	4050/865	5175/585	4060/865	5240/585	/
F02	58	3300/830	4505/500	2140	416968	1-10/10	3300/860	4505/500	3300/870	4520/500	/
F03	47	1665/685	2635/420	587	244000	1-10/10	1665/685	2635/420	1665/685	2645/420	/
F04	77	3485/855	4340/590	8528	280667	1-10/10	3485/875	4510/590	3485/900	4510/590	/
F05	61	3605/900	5465/595	2790	586035	1-10/10	3605/915	5830/595	3605/925	5840/595	/
F06	43	1875/690	2770/420	587	248520	1-10/10	1875/690	2880/420	1875/690	2880/420	/
F07	50	3335/880	4315/600	882	300075	1-10/10	3335/880	4325/600	3335/880	4370/600	/
F08	59	3705/1010	5530/600	2326	792722	1-10/10	3705/1010	5770/600	3705/1010	5835/600	/
F09	103	4730/820	7660/500	15894	967023	1-10/10	4730/870	7835/500	4770/865	7835/500	/
F10	49	2925/765	4610/540	414	371849	1-10/10	2925/765	4710/540	2925/770	4710/540	/
F11	94	3835/830	4645/560	19960	282936	1-10/10	3835/835	4875/560	3835/845	4995/560	/
F12	67	3395/830	4325/570	4878	249869	1-10/10	3395/830	4555/570	3405/855	4575/570	/
F13	52	2855/780	4255/520	1286	364325	1-10/10	2855/780	4315/520	2855/780	4315/520	/
F14	55	3330/915	5615/520	2247	907706	1-10/10	3330/915	5665/520	3330/950	5825/520	/

Continued on next page

Table 1: Detailed results over all instances (continued)

Instance	n	10000 iterations left	right	time[s]	hyper-vol	rank	1000 iterations left	right	100 iterations left	right	best left sol ten runs
F15	107	3560/825	4040/640	22403	143851	1-10/10	3560/830	4160/640	3560/845	4190/640	/
F16	54	2725/855	3495/630	1295	161810	1-10/10	2725/855	3495/630	2725/855	3515/630	/
F17	36	2055/825	4305/540	224	630432	1-10/10	2055/825	4325/540	2055/825	4405/540	/
F18	88	3075/825	4095/580	15324	259175	1-10/10	3075/825	4235/580	3075/830	4385/580	/
F19	66	2525/875	4125/530	6327	514673	1-10/10	2525/875	4130/530	2525/875	4205/530	/
F20	63	2445/760	3125/455	6513	241343	1-10/10	2445/770	3160/455	2445/805	3165/455	/
F21	72	2930/820	4670/570	4442	424018	1-10/10	2930/825	4680/570	2930/835	4850/570	/
F22	44	2075/790	3155/555	375	263275	1-10/10	2075/790	3235/555	2075/790	3235/555	/
F23	89	3005/820	4430/535	13843	424144	1-7/10	3010/820	4540/535	3010/820	4585/535	/
F24	86	3210/885	4355/580	10228	362225	1-10/10	3210/890	4550/580	3210/980	4550/580	/
F25	28	1390/695	2090/485	145	137350	1-10/10	1390/695	2120/485	1390/695	2165/485	/
g1-A	347	996314/79057	1106365/64602	247663	2772320599	4-5/10	1001014/79057	1109683/64602	1007721/79057	1127740/64602	**995012**/79057
g1-B	347	1119244/69401	1234420/64602	177761	1315176672	5/10	1127502/73666	1239506/64602	1128250/68281	1250141/64602	1118030/69401
g1-C	347	1247389/67327	1319112/64602	179723	790716736	5/10	1248841/67327	1322350/64602	1253038/70435	1322350/64602	**1240426**/65050
g1-D	347	1378623/69858	1461910/64602	142378	1263989332	5/10	1378623/69858	1482185/64602	1384613/69858	1488186/64602	**1370958**/69858
g1-E	347	1517332/65050	1581206/64602	105685	500386883	5/10	1523702/65050	1584236/64602	1541154/65050	1605028/64602	**1512391**/65050
g2-A	375	1099540/77143	1210964/64602	408421	2682206278	5/10	1106295/69858	1221964/64602	1108890/70190	1222205/64602	1097390/68655
g2-B	375	1212862/69576	1280285/64602	298460	1065013036	5/10	1222303/69858	1288482/64602	1225003/68909	1297422/64602	1209590/65050
g2-C	375	1342654/69858	1408032/64602	251254	1151418644	5/10	1342654/69858	1414252/64602	1350437/69858	1416940/64602	**1340303**/65050
g2-D	375	1483181/65050	1547055/64602	197390	494065575	5/10	1488894/65050	1552768/64602	1496833/65050	1560707/64602	**1480726**/65050
g2-E	375	1627294/65050	1687426/64602	157700	508708208	5/10	1629065/65050	1692939/64602	1642751/66004	1708399/64602	1621535/65050

4.2 The impact of three main components of `Decoder-EA`

Many (meta-)heuristic algorithms are designed by putting together a number of different components and `Decoder-EA` is no exception. A legitimate question might be asked: can certain of these components be disabled (or designed in a very different way) without substantially weakening the overall algorithm? To gain insight into this, we here compare the standard `Decoder-EA` to three other `Decoder-EA` variants obtained by performing the following changes:

A. Swap only blocks of equal length in the LS, *i.e.*, fix $\delta = \overline{\delta}$ in the neighborhood definition from Section 2.2.1. The resulting `Decoder-EA` variant is considerably easier to implement, because it is enough to record all routes using fixed-length array-like neighborhood data structures.

B. Perform a random survival selection instead of discriminating the implicit solutions using the non-dominated sorting from Section 3.1, and the roulette wheel from point 1.(b) of Section 3.4. We still keep the limitation from point 1.(a) of Section 3.4, *i.e.*, at maximum 30% of the current population (of implicit solutions) can survive to the next generation.

C. Perform a random parent selection instead of the roulette wheel from Section 3.3.

We selected the 5 instances for which `Decoder-EA` exhibits the most pronounced variation in performance when we change different parts of it. The results on many other instances are not not very sensible to (the considered) algorithm changes. To ensure fair comparative conditions, we do not impose a maximum number of iterations but a CPU time limit, *i.e.*, 200 seconds for `gdb08`, 800 seconds for `val5d` and `val8c`, 400 for `egl-e1-C` and 7000 for `egl-s2-C`.

Table 2 presents the comparison between the above `Decoder-EA` variants over 40 runs. For each variant, we reported the left-most solution (in the format "$C^{\text{tot}}/C^{\text{max}}$") for the best, the 10^{th}, the 20^{th}, the $30^{\text{th}}$and the worst run out of 40 (Column 2). The ranking criterion is the first objective (C^{tot}) breaking ties using the makespan. For the best and the worst run we also indicate the number of runs that tie for this position, *e.g.*, "3×348/44" means that 3 runs reached the same best objective values 348/44. The last five rows report the best rightmost solution ever reached in 40 runs. This table demonstrates that:

A. The use of unequal block swaps is clearly very useful for `Decoder-EA`, because the solution from Column 3 (standard `Decoder-EA`) is strictly better than the one from Column 4 (no unequal swaps) on roughly *three quarters of the rows.*

B. The non-dominated ranking used by the standard `Decoder-EA` to perform the survival selection is also very useful. The solution from Column 3 is strictly better than the one from Column 5 (random survival selection) for roughly *half of the rows.* The reverse happens much more rarely, only in 3 rows out of 25. The last five rows of Table 2 also show that using a random replacement selection degrades the quality of the best rightmost solution on all five instances. Preliminary experiments suggest that this may also degrade (the hyper-volume of) the optimal Pareto frontiers returned in the end.

C. Except for `egl-e1-C`, the comparison with the `Decoder-EA` variant from Column 6 (random parent selection) shows that is generally important to use the parent selection included by default in `Decoder-EA` (Section 3.3). Interestingly, a random parent selection may sometimes improve the quality of the best rightmost solution (see last five rows). This may come from the fact that the parent selection from Section 3.3 imposes a selective pressure that only relies on the first C^{tot} criterion while ignoring the second one.

If we did not include in Table 2 an algorithm version with no Local Search at all, it is because such a `Decoder-EA` variant would report very low quality results. Preliminary experiments clearly show that such an algorithm variant can not compete with the ones considered in this section. It may have difficulties to reach even the worst solution returned (in the end) by any algorithm from Table 2.

Instance	Rank	Standard `Decoder-EA`	Use only equal block swaps in LS	Random survival selection	Random parent selection
gdb08	best	3×348/44	4×348/44	3×348/44	5×348/44
	10th	348/51	348/48	348/48	348/62
	20th	350/44	350/44	350/44	350/44
	30th	350/44	350/44	350/44	350/44
	worst	1×352/44	2×350/50	1×350/46	1×353/44
val5D	best	8×575/83	2×577/83	1×575/83	3×575/90
	10th	577/82	581/80	577/83	577/83
	20th	577/83	583/83	577/84	578/83
	30th	577/83	585/80	578/82	579/83
	worst	3×579/83	1×586/79	1×581/81	1×581/82
val8C	best	2×521/77	3×521/83	1×521/80	1×521/85
	10th	521/83	525/77	523/80	525/80
	20th	523/77	527/76	525/79	527/77
	30th	523/79	527/79	527/77	527/79
	worst	1×525/79	1×531/77	1×527/81	1×529/78
egl-e1-C	best	38×5595/836	7×5595/836	35×5595/836	39×5595/836
	10th	5595/836	5615/859	5595/836	5595/836
	20th	5595/836	5615/859	5595/836	5595/836
	30th	5595/836	5625/839	5595/836	5595/836
	worst	1×5615/859	1×5675/836	5×5613/859	1×5613/859
egl-s2-C	best	3×16425/1040	1×16551/1040	4×16425/1040	1×16496/1040
	10th	16430/1040	16613/1040	16431/1040	16609/1040
	20th	16442/1040	16648/1040	16451/1040	16618/1040
	30th	16479/1040	16696/1040	16538/1040	16625/1040
	worst	1×16619/1040	1×16789/994	1×16669/1040	1×16672/1040
gdb08	best right	388/38	26×396/38	25×396/38	394/38
val5D	best right	689/72	671/72	689/72	683/72
val8C	best right	2×621/67	2×627/67	3×627/67	2×627/67
egl-e1-C	best right	7×5659/820	2×5677/820	4×5659/820	30×5659/820
egl-s2-C	best right	17046/979	17075/979	17058/979	16976/979

Table 2: Comparison of the standard `Decoder-EA` with three different variants obtained by changing certain components.

Finally, it is worth mentioning that we also launched more massive runs on `g1-B` and `egl-s2-B` because we had access to a more powerful cluster towards the end of the project.[9] Despite the fact that we launched hundreds of runs in parallel, the best we were able to achieve was to improve the best solution of `g1-B` from 1118030/69401 to 1117796/68328 and that of `egl-s2-B` from 13065/1060 to 13058/1060. This suggest that performing 100 runs instead of 10 runs does not necessarily lead to revolutionary better solutions.

4.3 Comparison with existing literature on the most critical instances

We here compare the results of the standard `Decoder-EA` to the best-known results from the bi-objective CARP literature. Specifically, we will refer to the following algorithms already discussed in the introduction:

1. the Multi-Objective Genetic Algorithm (MOGA) [Lacomme et al., 2006];

[9] We thank the director of the CRIStAL laboratory [more details after potential publication].

Instance	Decoder-EA		MOGA		D-MAENS		**ϵ-constraint**		ID-MAENS		IRDG-MAENS		DE-ICA	
	$best_1$	$best_2$	$best_1$	$best_2$	$best_1$	$best_2$	$best_1$	$best_2$	$best_1$	$best_2$	$best_1$	$best_2$	$best_1$	$best_2$
gdb08[a]	*348*	38	350	38	348	38	348	38	350		348	44		
gdb09[a]	*303*	37	309	37	304	37	303	37	303		303	43		
gdb13	*536*	128	544	128	536	128	536	128	536		536	128		
gdb23	*233*	13	235	20	233	20	233	25	233		233	22		
egl-e1-B	*4498*	820	4525	820	4525	820	4498	820	4525		4525	836	4525	
egl-e1-C	*5595*	820	5687	820	5595	820	5595	820	5595		5595	838	5595	
egl-e2-B	*6317*	820	6411	820	6347	820	6317	820	6340		6317	852	6321	
egl-e2-C	*8335*	820	8440	820	8339	820	8335	820	8414		8343	854	8335	
egl-e3-A	*5898*	820	5956	820	5926	820	5898	820	5898		5898	916	5898	
egl-e3-B	***7775***	820	7911	820	7801	820	7777	820	7789		7801	872	7787	
egl-e3-C	*10292*	820	10349	820	10340	820	10305	929	10307		10305	864	10311	
egl-e4-A	6461	820	6548	820	6476	820	6456	820	6472		6464	914	6473	
egl-e4-B	***8999***	820	9116	820	9069	820	9000	820	9004		9037	843	9031	
egl-e4-C	***11594***	820	11802	820	11774	820	11601	820	11618		11618	820	11634	
egl-s1-A	*5018*	912	5102	924	5068	912	5018	1023	5018		5018	1032	5018	
egl-s1-B	*6388*	912	6500	912	6435	912	6388	984	6422		6422	981	6435	
egl-s1-C	*8518*	912	8694	912	8518	912	8518	946	8518		8518	966	8518	
egl-s2-A	***9875***	979	10207	979	10117	979	9956	1051	10122		10122	1023	10040	
egl-s2-B	***13065***	979	13548	979	13459	979	13165	1060	13345		13331	1040	13283	
egl-s2-C	***16425***	979	16932	979	16832	979	16524	979	16682		16674	1040	16691	
egl-s3-A	***10233***	979	10456	979	10469	979	10260	1051	10347		10436	1053	10402	
egl-s3-B	***13682***	979	14004	979	14082	979	13807	979	13918		14020	998	13841	
egl-s3-C	***17223***	979	17825	979	17650	979	17234	979	17363		17342	1040	17324	
egl-s4-A	***12315***	1027	12730	1027	12602	1027		1040	12442		12654	994	12422	
egl-s4-B	***16198***	1027	16792	1027	16686	1027	16442	1027	16443		16600	1023	16430	
egl-s4-C	20614	1027	21309	1027	21213	1027	20591	1034	21195		20933	1027	20964	
val03A	*81*	27	81	31	81	27	81	34	81		82	39		
val04D	*530*	80	539	80	536	80	530	82	536		536	88		
val05D	***575***	72	595	72	595	72			586		579	81		
val08C	*521*	67	545	67	532	67	521	67	523		527	78		
val09A	*323*	44	326	68	324	47	323	97	323		325	107		
val09D	*391*	44	399	44	392	44	391	49	391		391	51		
val10A	*428*	47	428	91	428	62	428	125	429		430	139		
val10B	*436*	47	436	77	436	60	436	98	437		437	97		
val10C	*446*	47	448	66	446	58	446	79	446		447	39		
val10D	***526***	47	537	54	533	51	528	54	533		535	60		
g1-A	***995012***	64002									1027498	65367	1012078	
g1-B	***1118030***	64002									1174216	65752	1138229	
g1-C	***1240426***	64002									1289544	63833	1258065	
g1-D	***1370958***	64002									1441236	64473	1426809	
g1-E	***1512391***	64002									1587913	63365	1584395	
g2-A	***1097390***	64002									1137134	65050	1125827	
g2-B	***1209590***	64002									1256059	67738	1240357	
g2-C	***1340303***	64002									1428880	67296	1424978	
g2-D	***1480726***	64002									1568942	64602	1535056	
g2-E	***1621535***	64002									1687900	64602	1679487	

Table 3: Decoder-EA compared to six other algorithms (in chronological order) on the selected critical instances. We omit the Beullens instances (C01–F25) because no cited paper provides full results on them.[b] An underlined ***best***$_1$ value in Column 2 signals that the corresponding solution improves upon the best C^{tot} upper bound ever reported before in the (considerably) larger mono-objective CARP literature (as described in Remark 2, p. 16).

[a] In certain papers, gdb8 and gdb9 are called gdb10, resp gdb11; actually, all instances gdb$x$ above with $x \geq 8$ are called gdb$x+2$.

[b] IRDG-Maens provides only statistical/simulation information and DE-ICA only reports results for 16 instances out of 100.

2. the Decomposition-Based Memetic Algorithm (D-MAENS) [Mei et al., 2011];
3. The ϵ-**constraint** method [Grandinetti et al., 2012];
4. The Improved D-MAENS (ID-MAENS) [Shang et al., 2014];
5. The IRDG-MAENS algorithm [Shang et al., 2016a];
6. The Directed Evolution Immune Clonal Algorithm (DE-ICA) [Shang et al., 2016b].

We restrict to a set of (very) critical instances on which one can observe a higher variation in the performance of the compared algorithms. Recall from (Column 7 of) Table 1 that, on many instances, all ten `Decoder-EA` runs may report the same solutions in the end. On such instances, most algorithms from the literature also report the same C^{tot}–best and $C^{\max}$–best solutions.

Table 3 compares the best solutions reported by `Decoder-EA` to the one reported by the above algorithms; this is a simple comparison for indicative purposes only, because the experimental conditions and the running times of these algorithms can be very different. The columns $best_1$ and $best_2$ simply provide the C^{tot}–best and (resp.) the $C^{\max}$–best solution (ever) reported by each of the seven considered algorithms. For `Decoder-EA`, the contents of the columns $best_1$ and $best_2$ were simply imported from (Columns 3-4 or Column 12) of Table 1. A $best_1$ value in italics (in Column 2) indicates that `Decoder-EA` reached the best (C^{tot}) upper bound ever reported in the bi-objective CARP literature. A $\boldsymbol{best_1}$ value in both italics and boldface signals that corresponding solution strictly dominates all solutions reported by the other bi-objective algorithms shown here. An underlined $\underline{\boldsymbol{best_1}}$ value in italics and boldface indicates that `Decoder-EA` discovered an C^{tot} upper bound that has never been reported before in the (considerably) larger mono-objective CARP literature (as described in Remark 2, p. 16).

5 Exploring a Traveling Salesman Problem bi-objective variant

We here study the application of the decoder-based framework from Section 3 on a different problem. Most algorithmic descriptions from Sections 3.1–3.2 may actually apply to a new problem (most often without changing a word) by simply plugging-in a different decoder.

We consider the well-known Traveling Salesman Problem (TSP), but let us formulate it a manner that better fits the given CARP instances. As thus, the n required edges E_R become the vertices of a directed graph G_{TSP}. In the new TSP problem, we ask to traverse each edge $e = \{v_a, v_b\} \in E_R$ (*i.e.*, each TSP vertex) along the direction $v_a \to v_b$, where $v_a < v_b$. We say that v_a is the low end of e and that v_b is the high end of e. The salesman has to first visit the low end and then the high end; all required edges E_R have to be travelled this way. The TSP length of an arc (e, e') of G_{TSP} is simply given by the shortest path (in G) between the high end of e and the low end of e'. This means that after finishing travelling e at its high end, the salesman has to move to the low end of the next required edge (G_{TSP} vertex). The most straightforward mono-objective goal is to find the minimum cost tour in the directed graph G_{TSP}; the resulting problem can be easily reduced to TSP and vice-versa.

We next propose the bi-objective variant. First, let us assign a weight (price) to each vertex $e \in E_R$; this price is simply given by the demand q_e of e in the considered CARP instance.[10] Secondly, we allow the feasible tours to skip visiting a vertex by paying the associated price as penalty; this way, the number of vertices visited by a feasible tour becomes $|E_R| - 1$. Finally, a candidate solution is defined by a permutation of E_R and a skipped edge $e \in E_R$. We ask to minimize the following objective functions:

obj1 the length of the tour that visits all TSP vertices except the chosen skipped $e \in E_R$ in the order indicated by the permutation

obj2 the price (penalty) q_e associated to the chosen e.

[10] One may use a price given by the edge length and we can obtain a very similar problem.

The decoder is significantly simpler than the arc-routing one. Given a permutation of E_R as input, it is enough to scan all elements e of this permutation one by one and to calculate the above objective values for for each skipped edge e. Since the permutation has $n = |E_R|$ elements, this may generate up to n explicit solutions per permutation. In practice, however, most of these n solutions may be dominated; the decoder filters them and may eventually return a Pareto frontier with (far) less non-dominated solutions, in many cases only a few. Figure 6 below illustrates two tours that could be determined by our decoder on a simple instance.

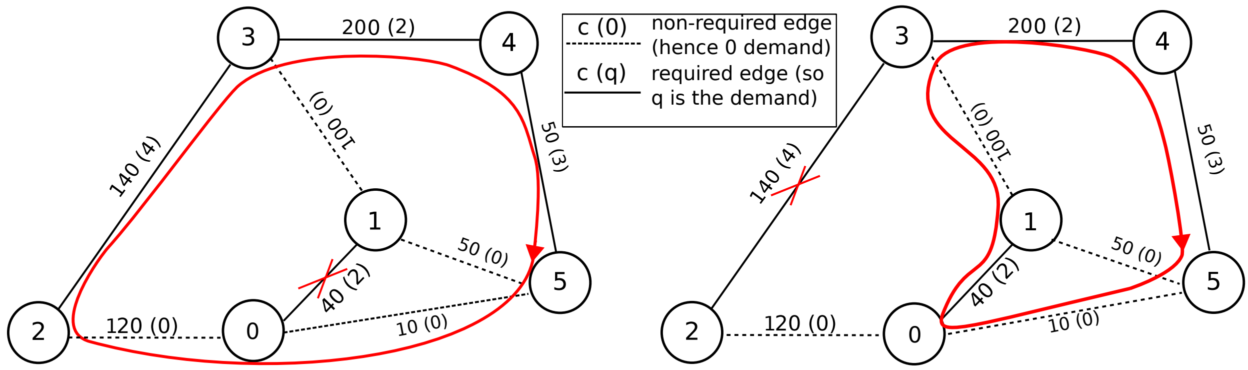


Figure 6: Two TSP tours for the same instance (originally used for Arc-Routing). Each edge has a label "c_{ij} (q_{ij})", where c_{ij} is the traversal cost and q_{ij} is the demand. The left solution skips edge $\{0, 1\}$ with a demand of 2, and so, $obj = 2$; its $obj1$ value is the total cost $120 + 140 + 200 + 50 + 10 = 520$. The right solution skips edge $\{2, 3\}$ with a demand of 4, and so, $obj2 = 4$; its $obj1$ value is the total cost $40+100+200+50+10 = 400$. Considering input permutation permutation $(\{0, 1\}, \{2, 3\}, \{3, 4\}, \{4, 5\})$, our decoder may determine (among others) these two solutions with objective values $(520, 2)$ and resp. $(400, 4)$ (and notice all edges $e = \{v_a, v_b\}$ are traversed in the sense $v_a \to v_b$ with $v_a < v_b$).

Table 4 presents the results of `Decoder-EA` on the `gdb` instances. Let us mention that we did not change a single line of code in the `C++` software module that implements the EA component; we only needed to link it to a separately compiled file (`tsp_decoder.cpp`) that implements the decoder. Table 4 reports the instance in Column 1, the number of vertices of G_{TSP} (equal to $n = |E_R|$) in Column 2, the leftmost solution in Column 3, the rightmost solution in Column 4 and the CPU time in seconds in Column 5. The stopping condition for the results in Columns 3-5 is to reach 100.000 iterations. The last two columns also report the leftmost and rightmost solutions reached after 1.000 iterations (usually calculated in a time of milliseconds).

Table 4: Results of `Decoder-EA` on the proposed TSP bi-objective variant

		100000 iterations			1000 iterations	
`Instance`	$n = \lvert E_R \rvert$	left	right	time[s]	left	right
`gdb01`	22	371/1	371/1	4	412/1	412/1
`gdb02`	26	439/1	439/1	5	462/1	462/1
`gdb03`	22	377/1	377/1	4	409/1	409/1
`gdb04`	19	363/1	363/1	4	377/1	377/1
`gdb05`	26	506/1	506/1	5	542/1	542/1
`gdb06`	22	369/1	369/1	4	396/1	396/1
`gdb07`	22	377/1	377/1	4	406/1	406/1
`gdb08`	46	531/9	543/1	18	570/4	591/1
`gdb09`	51	480/6	493/1	13	480/6	493/1
`gdb10`	25	384/2	391/1	6	405/2	411/1
`gdb11`	45	685/6	704/1	16	743/6	752/1
`gdb12`	23	558/3	578/1	9	598/16	622/1
`gdb13`	28	563/8	639/2	12	575/8	655/2

Continued on next page

Table 4: Detailed TSP results over all `egl` instances (continued)

		100000 iterations			1000 iterations	
`Instance`	$n = \|E_R\|$	left	right	time[s]	left	right
`gdb14`	21	131/5	142/1	10	136/5	145/1
`gdb15`	21	68/7	74/2	14	70/5	76/2
`gdb16`	28	145/2	147/1	10	154/7	155/1
`gdb17`	28	102/8	109/2	20	106/8	115/2
`gdb18`	36	218/8	221/1	11	227/3	230/1
`gdb19`	11	55/5	63/1	8	55/5	63/1
`gdb20`	22	145/1	145/1	7	149/4	151/1
`gdb21`	33	197/2	199/1	10	207/2	208/1
`gdb22`	44	234/8	236/1	12	240/8	243/1
`gdb23`	55	292/5	294/1	13	306/4	309/1

Future work may focus on problems that are not naturally expressed as permutation problems. In fact, any problem for which we can establish an order of the decision variables can be seen as a sequencing or permutation problem [Campos et al., 2005, Porumbel et al., 2017, van Hoorn, 2016]. If one can encode a candidate solution as an order of the decision variables, a decoder can assign values to these variables in the considered order.

To discuss only one example, consider the graph coloring problem. Graph coloring is usually seen as a partition problem in the sense that the candidate solutions represent partitions of the vertex set. But we could also interpret it as a permutation problem: consider an order of the vertices and color them step by step in this order by choosing, for each vertex, a color that minimizes the number of edges with their end vertices of the same color. Actually, there already exists a well-known coloring heuristic (DSatur) that uses such an order in a similar manner and that could (be extended to) be used as a decoder inside `Decoder-EA`.

6 Conclusions

We proposed a bi-objective Evolutionary Algorithm (EA) framework to which we plugged in an exact decoder to solve the bi-objective Capacitated Arc Routing Problem (CARP). Although we were initially motivated only by CARP, the proposed EA framework can be seen as an algorithmic "backbone" to which one can also attach a different decoder to solve a different problem. We could this way also solve a bi-objective variant of the Traveling Salesman Problem (TSP), without changing a single line of code in the EA software module. For both problems, the role of the decoder is to turn an implicit solution (permutation) into a Pareto frontier of non-dominated explicit solutions. The CARP decoder is significantly more complex than the TSP one because it integrates a dynamic programming scheme.

The decoder enabled us to decompose the problem, shifting the focus from the given problem to the space of implicit solutions. The EA framework could thus be designed (Section 3) without taking into account any particular CARP feature. This framework builds upon the non-dominated sorting concept of NSGA2; we could not directly use NSGA2 because the (fitness) evaluation of the implicit solutions is more complex than in NSGA2. This is due to the fact that each implicit solution is associated to a *set of* 2D points in the objective space and not to a unique point as in NSGA2. The proposed EA incorporates multiple mechanisms to maintain diversity and to encourage young individuals (recent offspring); we noticed there is a serious need to prevent older individuals from monopolizing the genetic material in the population.

Yet, we do not think that working only in the (more abstract) permutation space is enough to obtain the most competitive results. Recall that the final full algorithm was able to improve upon the best-known total-cost upper bound for nine instances, with regards to the (larger) mono-objective CARP literature. To achieve this, we had to reinforce the exact decoder using a CARP-specific Local Search (LS). More exactly, the explicit solution of minimum total-cost returned by the decoder is improved using an LS operator that manipulates explicit routes. Since the decoder is exact subject to the service order indicated by the input permutation (Theorem 1), the LS can only find better solutions by changing the input permutation. We can thus see the LS as a (mutation) operator that moves from one permutation to the other in the encoded space, with no negative interaction between the LS and the decoder.

References

[Campos et al., 2005] Campos, V., Laguna, M., and Martí, R. (2005). Context-independent scatter and tabu search for permutation problems. INFORMS Journal on Computing, 17(1):111–122.

[Corberán et al., 2021] Corberán, Á., Eglese, R., Hasle, G., Plana, I., and Sanchis, J. M. (2021). Arc routing problems: A review of the past, present, and future. Networks, 77(1):88–115.

[Deb et al., 2002] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. Evolutionary Computation, IEEE Transactions on, 6(2):182–197.

[Grandinetti et al., 2012] Grandinetti, L., Guerriero, F., Laganà, D., and Pisacane, O. (2012). An optimization-based heuristic for the multi-objective undirected capacitated arc routing problem. Computers & Operations Research, 39(10):2300–2309.

[Lacomme et al., 2006] Lacomme, P., Prins, C., and Sevaux, M. (2006). A genetic algorithm for a bi-objective capacitated arc routing problem. Computers & Operations Research, 33(12):3473–3493.

[Mei et al., 2014] Mei, Y., Li, X., and Yao, X. (2014). Cooperative coevolution with route distance grouping for large-scale capacitated arc routing problems. IEEE Transactions on Evolutionary Computation, 18(3):435–449.

[Mei et al., 2011] Mei, Y., Tang, K., and Yao, X. (2011). Decomposition-Based Memetic Algorithm for Multiobjective Capacitated Arc Routing Problem. IEEE Transactions on Evolutionary Computation, 15(2):151–165.

[Porumbel et al., 2017] Porumbel, D., Hsu, T., Allaoui, H., and Goncalves, G. (2017). Arc-routing via column generation and iterated local search in a permutation set-covering framework. European Journal of Operational Research, 256:349–367.

[Prins et al., 2014] Prins, C., Lacomme, P., and Prodhon, C. (2014). Order-first split-second methods for vehicle routing problems: A review. Transportation Research Part C: Emerging Technologies, 40:179 – 200.

[Santos et al., 2010] Santos, L., Coutinho-Rodrigues, J., and Current, J. R. (2010). An improved ant colony optimization based algorithm for the capacitated arc routing problem. Transportation Research Part B: Methodological, 44(2):246–266.

[Shang et al., 2016a] Shang, R., Dai, K., Jiao, L., and Stolkin, R. (2016a). Improved memetic algorithm based on route distance grouping for multiobjective large scale capacitated arc routing problems. IEEE transactions on cybernetics, 46(4):1000–1013.

[Shang et al., 2016b] Shang, R., Du, B., Ma, H., Jiao, L., Xue, Y., and Stolkin, R. (2016b). Immune clonal algorithm based on directed evolution for multi-objective capacitated arc routing problem. Applied soft computing, 49:748–758.

[Shang et al., 2014] Shang, R., Wang, J., Jiao, L., and Wang, Y. (2014). An improved decomposition-based memetic algorithm for multi-objective capacitated arc routing problem. Appl. Soft Comput., 19:343–361.

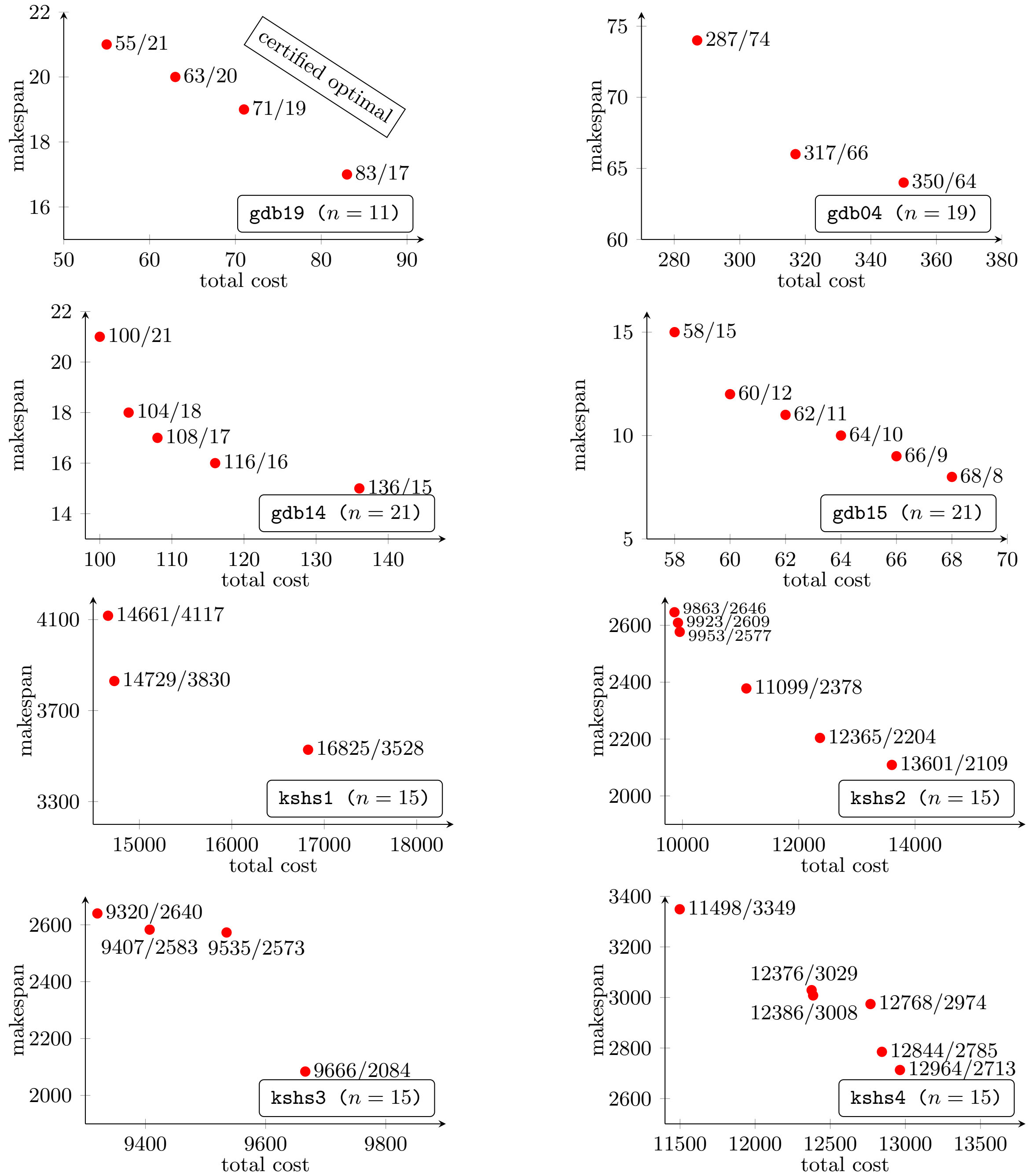
[Talbi, 2009] Talbi, E.-G. (2009). Metaheuristics: From Design to Implementation. Wiley Publishing.

[Ulusoy, 1985] Ulusoy, G. (1985). The fleet size and mix problem for capacitated arc routing. European Journal of Operational Research, 22(3):329–337.

[van Hoorn, 2016] van Hoorn, J. (2016). Dynamic Programming for Routing and Scheduling: Optimizing Sequences of Decisions. PhD thesis, Vrije Universiteit Amsterdam.

A The best Pareto frontiers obtained on the smallest instances

Here, we graphically depict the best Pareto frontiers reported by `Decoder-EA` on all instances with $n = |E_R| \leq 21$. For the smallest graph (`gdb19`) with $n = 11$, we could even calculate the optimal Pareto frontier by running the decoder on all $n!$ permutations.



B A decoder execution example

We here follow the construction of the (partial) solutions `sol`$[k]$ step by step for each $k \in [0..n]$ in Algorithm 1. The full solutions servicing all clients are constructed in the last step when $k = n$, *i.e.*, the final `sol`$[n]$ contains the Pareto frontier to be eventually returned.

The (partial) solutions determined at Step 3 of Algorithm 1 (p. 7) are generated as follows:

- Solution 1 ($k = 0$) represents a null artificial solution initialized at Line 14. The **for** loop at line 17 extends this solution by inserting all feasible routes starting at $e_{k+1} = e_1$. There are two such routes: (e_1) and (e_1, e_2) which generate transitions to Solutions 2 and 3.
- Solution 2 ($k = 1$) is expanded with all the routes starting at $e_{k+1} = e_2$, *i.e.*, all the routes recorded in $R(e_{k+1}, \ell)$ for all $\ell \in [1..\texttt{len}(e_{k+1})]$, see lines 18-19. This leads to Solutions 4, 6 and 8, *i.e.*, all solutions with two routes, the first of which is (e_1).
- Solution 3 ($k = 2$) is extended to Solutions 5 and 9. But Solution 5 is discarded at Line 20 because it is dominated by the already-computed Solution 6.
- Solution 4 ($k = 2$) leads to Solutions 7 and 12;
- Solution 5 was discarded above;
- Solution 6 ($k = 3$) and resp. Solution 7 ($k = 3$) generate Solutions 10 and resp. 11, both dominated by existing Solution 12.

solution ID	$C^{\max}$-ordered routes in $\texttt{sol}[k]$ in the form:$(C^{\text{tot}}, C^{\max})$	k	fleet size
1	$\emptyset$:(0,0)	0	0
2	$\{\underbrace{(e_1)}_{6}\}$:$(6,6)$	1	1
3	$\{\underbrace{(e_1,e_2)}_{18}\}$:$(18,18)$	2	1
4	$\{\underbrace{(e_1)}_{6},\underbrace{(e_2)}_{14}\}$:$(20,14)$		2
5	$\{\underbrace{(e_1,e_2)}_{18},\underbrace{(e_3)}_{8}\}$:$(26,18)^*$	3	2
6	$\{\underbrace{(e_1)}_{6},\underbrace{(e_2,e_3)}_{16}\}$:$(22,16)$		2
7	$\{\underbrace{(e_1)}_{6},\underbrace{(e_2)}_{14},\underbrace{(e_3)}_{8}\}$:$(28,14)$		3
8	$\{\underbrace{(e_1)}_{6},\underbrace{(e_2,e_3,e_4)}_{20}\}$:$(26,20)$	4	2
9	$\{\underbrace{(e_1,e_2)}_{18},\underbrace{(e_3,e_4)}_{12}\}$:$(30,18)$		2
10	$\{\underbrace{(e_1)}_{6},\underbrace{(e_2,e_3)}_{16},\underbrace{(e_4)}_{10}\}$:$(32,16)^*$		3
11	$\{\underbrace{(e_1)}_{6},\underbrace{(e_2)}_{14},\underbrace{(e_3)}_{8},\underbrace{(e_4)}_{10}\}$:$(38,14)^*$		4
12	$\{\underbrace{(e_1)}_{6},\underbrace{(e_2)}_{14},\underbrace{(e_3,e_4)}_{12}\}$:$(32,14)$		3

* indicates a dominated solution (never returned).

Table 5: Final solutions $\texttt{sol}[k]$ for $k \in [0..n]$. Algorithm 1 constructs this table and returns the non-marked (non-dominated) solutions listed for $k = 4$, i.e., $\{(26, 20), (30, 18), (32, 14)\}$.