Title: Arc-Routing via Column Generation and Iterated Local Search in a Permutation Set-Covering Framework

Corresponding Author: Daniel Porumbel
Institution:
CEDRIC Computer Science (CS) Laboratory
CS Department, CNAM (Conservatoire National des Arts et Métiers)
75003 Paris, France
Contact:
daniel.porumbel@cnam.fr
+331 40 27 24 33

Arc-Routing via Column Generation and Iterated Local Search in a Permutation Set-Covering Framework

Daniel Porumbel[a] Gilles Goncalves[b,c] Hamid Allaoui[b,c] Tienté Hsu[b,c]

[a] CEDRIC, Conservatoire National des Arts et Métiers, Paris, France
[b] Univ Lille Nord de France, F-59000 Lille, France
[c] U-Artois, LGI2A, F-62400 Béthune, France

Abstract

We propose a method that combines the paradigms of Column Generation (CG) and Iterated Local Search (ILS) to solve the `Capacitated Arc-Routing` problem. One of the goals is to integrate into the ILS (some of) the duality information that underpins the CG. We consider a space of permutations and sub-permutations (sequences) of the set of required edges $[1..m]$. This space is explored by an ILS process and a CG process that run in parallel and that can repeatedly exchange sub-permutations. The ILS uses an exact decoder that maps any permutation $s : [1..m] \to [1..m]$ to a list of sequences (routes) of minimum cost that services $[1..m]$ in the order $s(1), s(2), \dots, s(m)$. The first use of the CG paradigm in ILS is the following: all sequences discovered by the CG process are sent to the ILS process that can inject them into the current ILS solution. The second application of CG in ILS consists of a "CG improver" that starts from the current ILS solution and tries to improve it by running several CG iterations. The first half of the paper describes the proposed method in a general framework based on sequences, permutations and set covers. The second part is devoted to more specialized `Arc-Routing` techniques. For instance, the CG convergence could be accelerated by factors of tens or even hundreds by exploiting two ideas in the Dynamic Programming (DP) pricing: (i) avoid as much as possible to traverse edges without service before the end of the CG process, and (ii) detect and prune dominated DP states by recording them in a fast data structure that relies on an array and a red-black tree. Regarding the ILS, we show that the permutation-level search can be substantially improved if the exact decoder is reinforced with a deterministic post-decoding operator that acts on explicit routes. The overall results are competitive (reducing the best-known gap of five instances) and certain ideas could be potentially useful for other set-covering or permutation search problems.

Keywords Column Generation, Iterated Local Search, Arc Routing

1 Introduction

Given a graph (street network) and a set of required edges (streets) $E_R = [1..m]$, the `Arc-Routing Problem` (ARP) asks to find a set of routes of minimum cost that service each edge of E_R. The `Capacited ARP` (CARP) is a well-known variant in which the amount of service provided by each route is bounded by a maximum capacity Q. Historically, the first model related to `Arc-Routing` dates back to the well-known "Seven Bridges of Konigsberg" problem proposed by Euler in the 18^{th} century. This problem can be stated using the ARP terminology: can one travel along (service) the seven bridges (edges) of a city without traversing any bridge twice (without dead-heading)? However, modern applications of `Arc-Routing` go far beyond the scope of this question, spanning a variety of fields, *e.g.*, electric and rail line inspection, postal delivery, meter reading, road winter gritting, waste collection, street cleaning, cattle feeding logistics, etc. The reader whose curiosity is piqued can relate to ARP surveys [11, 34]. for more applications and references.

To ease the exposition, we will first describe our CARP method in a *Permutation Set-Covering* framework, *i.e.*, as a method for solving problems of the following form: minimize the cost needed to cover a ground set $[1..m]$ with sub-sets associated to sequences of $[1..m]$, *e.g.*, vehicle routes, crew schedules, commodity paths, etc.—see Definition 1 below. Regarding the exact methods, such problems are often tackled by Column Generation (CG) or branch-and-cut-and-price, because the number of feasible sequences can be prohibitively large. Regarding the (meta-)heuristic algorithms, they usually address such problems by searching through a space of permutations using permutation-specific operators [9].

While CARP received less attention than the `Vehicle Routing Problem` (VRP), the edge focus of CARP is very useful when the demands are located on edges rather than on vertices. A relatively straightforward approach for solving CARP on a graph G consists of applying a VRP algorithm on the line graph of G [20, 2, 23]. As already argued in the literature (see [6] or [19, §2.1]), this approach has significant drawbacks (*e.g.*, inherent symmetry, dense complementary graphs, huge number of vertices), and so, specialized CARP methods do merit serious consideration. Given the numerous applications of CARP and its particular structure, there has been an increasing interest and sophistication in pure CARP methods. We hereafter focus on such "VRP-free" algorithms. From a heuristic perspective, the following general methodologies have already been applied: tabu and scatter search [15, 14, 7, 24], variable neighborhood search [16, 26], iterated local search [29], guided local search [5], memetic and population-based methods [17, 18, 32, 12]. The exact algorithms often rely on solving integer programs using branch-and-cut or branch-and-bound combined with CG [4, 19, 3, 6, 21, 22, 27]. Important progress has been made during the last few years: recent algorithms [3, 6] could solve to optimality many instances with up to $m = 190$ clients (edges to service) and a capacity $Q \in [5, 600]$.

However, upper bounds for large-scale CARP ($m > 200$ and $Q \in [10000, 30000]$) have only been reported by Local Search (LS) [7, 23, 24]. For such instance sizes, a promising approach could be to combine a relatively lightweight LS with other meta-heuristics paradigms of well-acknowledged potential. For example, given a starting Iterated Local Search (ILS), it could be promising to extend it to a population-based memetic algorithm, *i.e.*, a genetic algorithm incorporating LS. The potential of population-based methods to improve stand-alone LS algorithms is long-acknowledged in `Arc-Routing` [17, 18, 32, 12], as well as in many other problems [28, 13, 25]. However, this paper combines an ILS with the paradigm of CG. One wonders if the advantages of population-based methods could also be realized (or surpassed) by CG techniques. A challenging aspect of this work was to accelerate the CG pricing, so as to make the CG process reach a speed comparable to that of the LS, especially for instances with $m > 200$ *and* $Q > 10000$.

We aim at enriching the ILS with sequences coming from CG, and so, to make the ILS take profit from the dual nature of the clients $[1..m]$. Technically, we can interpret this idea as follows. Consider the current ILS solution s as a concatenation of sequences (routes) with different costs. When these sequences are used as columns in the CG model, the dual variable y_i of any $i \in [1..m]$ actually represents a dual cost of client i, *i.e.*, its contribution to the cost of s. The dual variables y_i (with $i \in [1..m]$) actually guide the CG process: at each CG iteration, the CG searches for a new sequence (route) that services clients $I \subset [1..m]$ at a lower cost than at their current total dual cost $\sum_{i \in I} y_i$.

The resulting method, hereafter called `CG-P-ILS` (CG-Permutation-ILS), uses a CG process and a CG improver that interact with the ILS process by exchanging sequences of elements of $\big[1..m\big]$. More exactly, the CG process runs in *parallel* with the ILS process and acts upon it by repeatedly communicating sequences (routes) constructed by CG pricing. These sequences can be inserted in the current ILS solution during the ILS perturbation phase. In a general sense, these CG-constructed sequences play the role of the offspring solutions generated by crossover in memetic algorithms, *i.e.*, they enrich the local search with "external information" and increase diversity. The CG improver is a CG-based operator that can be applied on a primal ILS solution s to (try to) produce an improved primal solution. For this, it constructs an initial restricted master program from the columns (sequences) that compose s, generates new sequences by CG pricing, and progressively rounds master variables until all elements of $[1..m]$ are covered. After several CG iterations, this leads to a potentially improved primal solution that can replace s in the ILS process.

In the rapidly emerging math-heuristic literature, this CG-ILS hybridisation might seem to be a relatively classical approach at a first glance. However, it does not exactly fit in any of the widely-used math-heuristic typologies presented, for instance, in the recent vehicle routing survey [1]. The most related cited math-heuristics classes:[1] (a) the use of exact methods as operators inside meta-heuristics (local optimization), and

[1]Generally speaking, the other classes discussed in [1] are: *decompositions* of complex integrated problems into subproblems

(b) CG with a restricted set of columns constructed heuristically (restricted master heuristics). Our method does not really fit in the class (a), because the CG process is not merely an operator inside the ILS. The main CG process is intertwined with the ILS process: these processes run in parallel and they can continually communicate. On the other hand, the CG improver does clearly belong above class (a), because it *is* an operator inside the ILS. It can also be seen as "restricted master heuristic" of class (b), because it does not fully converge and it starts from columns constructed heuristically (by ILS).

The paper organization is incremental and all `CG-P-ILS` operators are gradually introduced according to their level of CARP specificity. As such, Section 2 starts out by presenting our CG and ILS algorithms in a *Permutation Set-Covering* framework. The CARP is formally presented in Section 3, where we slightly customize the set-covering CG model and the ILS process (Sections 3.2 and 3.3), essentially only using notions of edge sequences and permutations. The second half of the paper is devoted to more specific CARP notions. Section 4 recalls the sparsity-exploiting scheme from [19] and presents the new acceleration techniques, *e.g.*, avoid as much as possible to produce routes with edges traversed without service. Section 5 is devoted to the transformation of permutations into CARP solutions: we first use an exact decoder, followed by a deterministic post-decoding operator that applies faster route-level operations. We present numerical results and comparisons in Section 6, followed by conclusions in the last section. Additionally, we provide in appendix more details on: the numerical results (App. A), the ILS neighborhood and parameters (App. B), the CG cycling avoidance techniques (App. C); we finally give a small decoder example (App. D).

2 `CG-P-ILS`: Algorithmic Template for Permutation Set-Covering

We first present the main building blocks and definitions (Section 2.1), followed the `CG-P-ILS` pseudo-code (Section 2.2), finishing with a more detailed analysis of the CG-ILS interaction (Section 2.3).

2.1 General CG and ILS for Permutation Set-Covering

Definition 1. *(Permutation Set-Covering) Given a ground set* $[1..m]$*, a permutation set-covering problem requires selecting a number of sequences* $r = (r_1,\ r_2, \dots, r_{|r|})$ *with* $r_1, r_2, \dots, r_{|r|} \in [1..m]$ *such that each* $i \in [1..m]$ *arises in at least one sequence. The set of feasible sequences is typically described implicitly, e.g., as vehicle routes, crew schedules, paths of commodities, etc. Each sequence has a given cost that can represent a distance, a schedule delay, etc. The goal is to minimize the total cost of the selected sequences.*

To each feasible sequence r, we associate an incidence vector $\mathbf{a}$ (also noted $\mathbf{a}^r$) such that a_i is 1 if a_i arises in r or 0 otherwise, as well as a cost c_a (also noted c_a^r). Let us note $\mathcal{A}$ the set of (prohibitively-many) columns of the form $\begin{bmatrix} c_a \\ \mathbf{a} \end{bmatrix}$ associated to such sequences. We associate to each column a master variable x_a that indicates the selection status of the underlying sequence. In the initial ILP formulation, these decision variables need to be binary and, after linear relaxation, we obtain the following primal-dual model.

$$\mathbf{y}: \quad \begin{aligned} &\min \textstyle\sum c_a x_a \\ &\textstyle\sum a_i x_a \quad \geq 1,\ \forall i \in [1..m] \\ &x_a \geq 0 \qquad \forall \begin{bmatrix} c_a \\ \mathbf{a} \end{bmatrix} \in \mathcal{A} \end{aligned} \qquad (2.1a) \qquad\qquad \mathbf{x}: \quad \begin{aligned} &\max \mathbf{1}_m^\top \mathbf{y} \\ &\mathbf{a}^\top \mathbf{y} \leq c_a, \quad \forall \begin{bmatrix} c_a \\ \mathbf{a} \end{bmatrix} \in \mathcal{A} \\ &\mathbf{y} \geq \mathbf{0}_m \end{aligned} \quad , \qquad (2.1b)$$

where all the sums are carried out over all columns $\begin{bmatrix} c_a \\ \mathbf{a} \end{bmatrix} \in \mathcal{A}$.

This is actually the classical CG model for general *Set-Covering* with no ordering notion. It could be used even if the columns would not be associated to sequences, but with subsets of $[1..m]$. In many cases, there are prohibitively-many columns that are typically generated one-by-one (see below) by solving the pricing CG sub-problem. However, our CG process does not completely forget the permutation nature of the sequences generated by pricing, because it actually sends *full sequences* to the ILS process.

that can be solved exactly (*e.g.*, cluster-first route-second); *one-shot* applications of integer programs to improve solutions reported heuristically; *heuristic rounding* of relaxed solutions generated by CG, and CG heuristic branching.

We briefly recall the main CG ideas. A CG process starts out by optimizing a restricted master problem with relatively few initial columns. In the CARP-customized `CG-P-ILS`, the initial columns come from the pool $\mathcal{P}$ of the high-quality sequences generated by the first ILS iterations. However, for any given set of initial columns $\mathcal{A}' \subseteq \mathcal{A}$, the classical CG algorithm optimizes the above primal-dual programs (2.1a)-(2.1b) as follows: (i) find an optimal primal-dual solution considering only the current set of columns $\mathcal{A}' \subseteq \mathcal{A}$; (ii) given the current dual solution $\mathbf{y}$, solve the pricing sub-problem $\min_{[c_a\ \mathbf{a}]^\top \in \mathcal{A}}(c_a - \mathbf{a}^\top \mathbf{y})$ to search for a negative reduced cost column (violated dual constraint); (iii) if this reduced cost is strictly negative, add the corresponding column to $\mathcal{A}'$ and repeat from (i), or otherwise report optimality.

Regarding the heuristic component of `CG-P-ILS`, it relies on a classical Iterated Local Search framework acting in the space of permutations. To be specific, each ILS iteration consists of the following general steps: (i) generate and evaluate *one by one* the neighbors of the current solution s; (ii) replace the current permutation s with the best (or the first improving) neighbor; (iii) if a stagnation condition is met, apply a perturbation on the current solution. This last perturbation step is the main particularity of the ILS framework. Its goal is to make the search process move ("iterate") to a different search space area, so as to avoid stagnating or looping on local optima and plateaux. Our perturbation strategy relies on a pool $\mathcal{P}$ of high-quality sequences that can come either from primal solutions discovered via ILS or from sequences generated by CG pricing (see below). When a *stagnation condition* is detected, `CG-P-ILS` selects a sequence $r = (r_1, r_2, \ldots, r_{|r|})$ from $\mathcal{P}$ and simply perturbs the current solution s as follows: insert $r_1, r_2, \ldots, r_{|r|}$ at the beginning of s and remove all other occurrences of $r_1, r_2, \ldots, r_{|r|}$ from s.

Our ILS also uses a second (stronger) perturbation operator (see also Step 3 of the ILS pseudo-code in Algorithm 1). This is carried out by the CG Improver and it is devoted more difficult situations of recurrent looping. We describe both perturbation types in greater detail in Section 2.3.

2.2 The General `CG-P-ILS` Pseudo-code

Algorithm 1 Algorithmic template of `CG-P-ILS`

```
Main Column Generation process
    do
      1. optimize the current (restricted master)
         CG program and obtain dual vector $\mathbf{y}$
      2. $r, \mathbf{a}^r, c_a^r \leftarrow$ pricing($\mathbf{y}$)        ▷ find a new sequence $r$ (incidence vector $\mathbf{a}^r$, cost $c_a^r$)
      3. add column $\begin{bmatrix} c_a^r \\ \mathbf{a}^r \end{bmatrix}$ to the current CG program
      4. $\mathcal{P} \leftarrow$insert-sequence($\mathcal{P}, r$)        ▷ (try to) add sequence $r$ to pool $\mathcal{P}$
    while a negative reduced cost column is found
    return $lb$ =optimum determined at Step 1

Iterated Local Search process:        ▷ This process also launches the above CG process in parallel
    do
      1. repeat
           – $s \leftarrow$select-neighbor($s$)        ▷ $s$ is the current solution (permutation)
           – $ub \leftarrow \min(ub,$obj-value($s$))
           – $\mathcal{P} \leftarrow$insert-sequences($\mathcal{P}, s$)        ▷ it tries to insert the best sequences from $s$
         until a stagnation condition is detected
      2. $s \leftarrow$ perturb(s, $\mathcal{P}$)        ▷ insert a sequence from $\mathcal{P}$ at the beginning of $s$
      3. if recurrent looping detected,
           – $s \leftarrow$CG-Improver($s$)        ▷ (try to) improve $s$ via CG, see Section 2.3.2
    while a stopping condition is not met
    return $ub$
```

Algorithm 1 provides the algorithmic template of `CG-P-ILS`: it basically consists of a CG and an ILS process that run in parallel and that interact as follows. The sequences generated during the CG process can be inserted in a pool $\mathcal{P}$ of high-quality sequences shared with the ILS process (see Step 4 of the CG process in Algorithm 1). The ILS perturbation operator (Step 2 of the ILS process) retrieves such a sequence $r = \left(r_1,\ r_2, \ldots r_{|r|}\right)$ from $\mathcal{P}$ and inserts $r_1,\ r_2, \ldots r_{|r|}$ at the beginning of the current permutation s (removing all other existing occurrences of $r_1,\ r_2, \ldots r_{|r|}$ from s). While this operation might seem to have disruptive nature, it actually tries not to break the highest-quality blocks (sequences) that compose good solutions. Its design uses similar principles as crossover operators in genetic algorithms. For instance, the pool $\mathcal{P}$ tries to preserve the best sequences in the same way a population tries to preserve the best blocks or the best parts of solutions (*e.g.*, color classes in graph coloring [13]). To avoid inserting very similar elements into $\mathcal{P}$, we will use distances between sequences as in *spacing memetic algorithms* [28].

The second application of the CG paradigm in ILS is represented by the call $s \leftarrow$`CG-Improver`(s) in Line 3 of the ILS process (Algorithm 1). The goal of this "CG Improver" is to (try to) improve the current ILS primal solution s using a new shorter CG process that essentially performs the following steps. It takes as input the sequences that compose the primal solution and uses them as columns to build an initial CG program. This allows it to start out with a reasonable primal-dual solution with integer primal values. The dual value of an element $i \in \left[1..m\right]$ can be seen as the contribution of i to the cost of s. Guided by these dual values, the pricing constructs a new sequence and the CG improver performs several CG iterations. Gradually, the largest primal values of the new generated sequences are fixed to 1 after each iteration, enforcing their selection. When these selected sequences cover $[1..m]$, the CG improver concatenates them and returns a (potentially improved) primal solution—a description of greater detail is given in Section 2.3.2.

The interaction between the CG and ILS processes is also summarized in Figure 1 below.

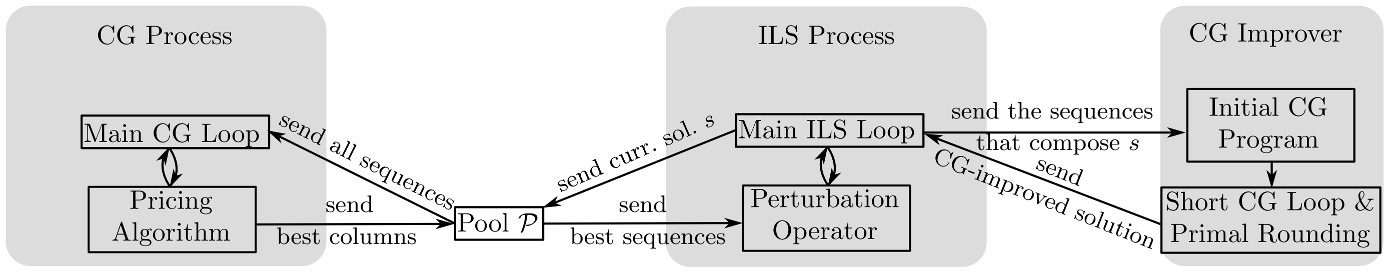


Figure 1: Summary of the interactions of the ILS process with the CG process and the CG improver.

2.3 The CG-ILS Interaction: Using CG information in ILS

We here describe in greater detail how: (i) the sequences inserted by the main CG process in $\mathcal{P}$ are used to perform standard perturbations (Section 2.3.1), and (ii) the CG improver executes several CG steps to perform stronger perturbations (Section 2.3.2).

2.3.1 Standard Perturbations Using CG-generated Sequences

As hinted above, the sequences generated by CG pricing can be inserted in the pool $\mathcal{P}$, so as to later inject them into the current ILS solution s (perturbation phase). In fact, this pool $\mathcal{P}$ is continually updated by adding sequences discovered both by CG or ILS. Each new sequence r is evaluated by a heuristic cost function that assesses its potential for contributing to high-quality full permutations. If a pool size limit is reached, an insertion of r would trigger the removal of a sequence with low heuristic cost. A reasonable heuristic cost function would favor a sequence r that contains some $i \in \left[1..m\right]$ covered by very few other $\mathcal{P}$ sequences. It could naturally take into account the similarity between r and the existing $\mathcal{P}$ elements; for instance, we forbid inserting the same sequence twice. We also forbid the complete removal of all CG-produced sequences. However, the CARP-customized heuristic cost function is described in greater detail in Section 3.3.3.2.

We still need to provide the technical stagnation condition: the perturbation is triggered after a number of iterations with no objective function variation. The value of this number can evolve depending on the status of the search (*e.g.*, the quality of the current solution compared to the best visited solution), but such customization aspects are provided in Section 3.3.3 (see also Appendix B.2 for the parameter setting).

2.3.2 The CG Improver: Stronger Perturbations

In certain cases, the standard perturbation operator is not strong-enough to make the ILS process evolve to a really new area of the search space. A few iterations after applying a classical perturbation, the ILS can actually return to previously visited area. Such situations indicate a *recurrent looping state* and we will later provide (Appendix B.2) CARP-customized conditions for detecting it. For now, it is enough to say that such a looping situation makes `CG-P-ILS` launch the CG improver on the current solution s. The goal is to exploit (some of) the dual information that underpins the *non-stagnant* nature of CG.

Starting from current permutation s, the CG improver performs several iterations to try to produce a higher quality solution s_{new}. The approach is inspired by the fixing-releasing CG heuristic from [8, §4]. More exactly, we first build an initial CG primal program (3.1.a) filled with the columns associated to the sequences that compose s. The new permutation s_{new} is initially empty and we construct it as follows:

(a) run a series of $nrPiv$ non-degenerate pivots, *i.e.*, run a CG process until it generates $nrPiv$ columns that do increase the objective value, where $nrPiv$ is a parameter (fixed in Section 3.3.3.1 for CARP).

(b) fix to 1 the highest primal value associated to a column in the current master program. The sequence $(r_1,\ r_2, \ldots r_{|r|})$ associated to this column is concatenated to the new permutation s_{new}. The next calls of point (a) above will only look up columns that service *no* element of $\{r_1,\ r_2, \ldots r_{|r|}\}$.

By iteratively applying the above steps, the fixed (integer-valued) columns progressively cover the set $[1..m]$. Since each new pricing sub-problem takes into account only the as-yet-uncovered elements of $[1..m]$, we gradually reduce the sub-problem size. If the whole set $[1..m]$ can be covered this way, the new permutation s_{new} is fully constructed. Otherwise, the CG improver stops when there is no negative reduced cost column only composed of *uncovered* elements of $\big[1..m\big]$; in this case, s_{new} is constructed by concatenating the generated columns, followed by the uncovered elements in their initial order in s.

3 `CG-P-ILS`: Arc-Routing Customization

3.1 Arc-Routing Definitions

We consider an input graph $G = (V, E)$ with $n = |V|$ and a set $E_R = \big[1..m\big] \subseteq E$ of required edges that need to be serviced. We use a function $d : V \times V \to \{R_+, \infty\}$ such that $d(u, v)$ is the length of edge $\{u, v\}$ if $\{u, v\} \in E$, or $d(u, v) = \infty$ if $\{u, v\} \notin E$; by slightly abusing notation, the length of edge $i \in E$ can also be noted d_i. If $i \in E_R$, then edge i also requires a service (supply) amount of q_i. A feasible route r is a walk in G that starts and ends at a special depot vertex v_{dep} and that provides a service (supply) of maximum Q to a sequence of required edges. The cost of a route is given by its total length, including both serviced serviced and non-serviced (dead-headed) edges. We consider we can only use a fleet size of k vehicles. The goal is to provide a service of q_i to each $i \in E_R$ using at maximum of k feasible routes such that the total cost (travelled distance) is minimized.

3.2 Customizing the CG Model

The CARP customization of the CG algorithm simply consists of incorporating a fleet size constraint in the CG formulation from Section 2.1. Specifically, we add a primal constraint limiting the number of selected routes to k, along with the associated dual variable μ; as such, the formulation (2.1a)-(2.1b) evolves to:

$$\begin{array}{rll} & \min \sum c_a x_a & \\ \mathbf{y}: & \sum a_i x_a \geq 1, \forall i \in [1..m] & \\ \mu: & \sum x_a \leq k & \\ & x_a \geq 0 \quad \forall \begin{bmatrix} c_a \\ \mathbf{a} \end{bmatrix} \in \mathcal{A} & \end{array} \qquad (3.1a) \qquad \begin{array}{rl} & \max \mathbf{1}_m^\top \mathbf{y} - k\mu \\ \mathbf{x}: & \mathbf{a}^\top \mathbf{y} - \mu \leq c_a, \quad \forall \begin{bmatrix} c_a \\ \mathbf{a} \end{bmatrix} \in \mathcal{A} \\ & \mathbf{y} \geq \mathbf{0}_m \\ & \mu \geq 0 \end{array} , \qquad (3.1b)$$

In the CARP-customized CG, the initial columns come from the routes inserted by the ILS process into the sequence pool $\mathcal{P}$. We start the main CG process when the size of this pool is sufficiently large. The CARP pricing sub-problem asks to find a route r such that the associated column $[c_a^r\ \mathbf{a}^r]^\top$ minimizes $c_a + \mu - \mathbf{a}^\top \mathbf{y}$ over all feasible columns $[c_a\ \mathbf{a}]^\top \in \mathcal{A}$. The sub-problem input consists of dual variables $\mathbf{y}$ and μ. If the resulting minimum value is strictly negative, column $[c_a^r\ \mathbf{a}^r]^\top$ violates a dual constraint, and so, we add it to the current set of columns. The Dynamic Programming (DP) pricing is presented in Section 4, along with new acceleration techniques. These techniques can accelerate the pricing by orders of tens, which is particularly relevant for (the Step 3 of) the ILS process in Algorithm 1, because the CG improver acts as a blocking operator in the ILS.

3.3 CARP Customization of the ILS

The CARP customization mainly concerns three aspects: the solution evaluation, the neighborhood structure and the perturbation operator. We first address the solution evaluation: since CARP is not an explicit permutation problem, we need to use tour-splitting decoding techniques to map permutations $s : [1..m] \to [1..m]$ into explicit solutions (Section 3.3.1). Secondly, we propose a CARP-specialized neighborhood constructed by pruning non-interesting neighbors from a very general *permutation neighborhood* (Section 3.3.2). Thirdly, we discuss in greater detail the perturbation operator and its relation to the CG, *i.e.*, the use of the CG improver and the construction of the pool $\mathcal{P}$ shared with the main CG process (Section 3.3.3)

3.3.1 Evaluating Permutations via Decoding and Post-decoding

Given a permutation noted $s = (s_1, s_2, \dots s_m)$, the decoder returns a set of routes of minimum total cost that services E_R in the order $s_1, s_2, \dots s_m$. We use a Dynamic Programming (DP) scheme that extends classical decoders for *permutations with orientations* (with a traversal sense for each edge). We will fully describe our DP scheme in Section 5.2, but for now it is enough to say that it uses $O(m \cdot \max_{r \in s}(|r|))$ DP states, where $\max_{r \in s}(|r|)$ is the maximum number of serviced edges in a potential route that can result from decoding s. Our decoder has the same complexity as other decoders of permutations with orientations, *i.e.*, $O(m \cdot \max_{r \in s}(|r|))$ if the number of vehicles is not taken into account or $O(m \cdot \max_{r \in s}(|r|) \cdot k)$ otherwise (see Section 5.2.3.

The resulting explicit solution can be further improved by a deterministic post-decoding operator. We will fully describe this operator in the CARP-customized Section 5.3, but its main idea is to scan all pairs of (sequences of) edges and to try to apply on each of them different route transformations that can be calculated locally, *e.g.*, swap two edges without modifying the rest of the route. Such route transformations can be calculated in $O(1)$, significantly more rapidly than by modifying s and decoding. If the input permutation is modified during this post-decoding, then current ILS solution takes the value of the modified permutation.

Finally, we mention a CARP-specific issue that should not be overlooked when one only has a limited fleet size k. For low-quality permutations s, it might not be possible to service all required edges in the order $s_1, s_2, \dots s_m$ only using k routes. In such cases, the objective value of s takes the value of an infeasibility penalty *plus* the minimum non-loadable quantity, *i.e.*, the minimum amount that can *not* be serviced. We say that such permutations belong to the penalized search space. The neighbor set is the same and the ILS iterations are performed in the same manner, but the decoder can be faster (Remark 3, Section 5.2.3). The post-decoding operator behaves as a heuristic with a bin-packing objective, *i.e.*, it swaps elements of s so as to make all routes reduce their unused capacity (Remark 4, Section 5.3).

3.3.2 A Specialized CARP Neighborhood from a Generic Permutation Neighborhood

We construct a *restricted* (CARP-focused) neighborhood from the most promising neighbors of a complete neighborhood N_C based on generic permutation operations. Given the current solution $s = (s_1,\ s_2, \dots s_m)$, $N_C(s)$ is defined as the set of all permutations s' that can be reached from s by applying any of the following:

consecutive sequence swap Given any $i \le j < \ell$, this operation consists of swapping sequence $(s_i, s_{i+1}, \dots s_j)$ with sequence $(s_{j+1}, s_{j+1}, \dots s_\ell)$. Considering all feasible values of $i, j, \ell \in [1..m]$, the complete neighborhood contains $O(m^3)$ *sequence neighbors.*

swap Given any $i < j$, swap s_i and s_j. These transformations can produce $O(m^2)$ *swap neighbors.*

This set of neighbors is actually generated in a *first-improvement* manner: the above transformations are tried iteratively *until* a first strictly improving neighbor is found. As such, the *swap* neighbors are always generated after the *sequence* neighbors—only if there is no *sequence* neighbor that strictly improves s. This order choice proved to have a certain importance on the practical effectiveness of the search. However, this complete neighborhood can be too large: it is not always computationally viable to iteratively generate $O(m^3)$ neighbors and to apply a decoder and a post-decoder on each one.

As hinted above, `CG-P-ILS` actually relies on a restricted version of this neighborhood and we use four reduction (pruning) techniques for this purpose. First, for $m > 100$, we only generate sequence neighbors with $\ell = m$, so as to reduce the neighborhood size to $O(m^2)$. We also pre-evaluate all permutations in $N_C(s)$ with a fast heuristic function and prune all neighbors with very distanced consecutive edges. All these techniques described in greater detail in Appendix B.1.

3.3.3 The ILS-CG Interaction: CG Improver and Perturbation Pool Updating

We recall the two tools for improving diversity in the ILS: the classical perturbation operator (for avoiding local optima) and the CG improver (for addressing more difficult recurrent looping). We first discuss the CG Improver (Section 3.3.3.1), followed by the classical perturbation (Section 3.3.3.2).

3.3.3.1 Escaping Recurrent Looping Using CG `CG-P-ILS` triggers a standard perturbation after a number of iterations with no objective function variation. However, in certain cases, the standard perturbation can not really lead to a sustainable diversity. The ILS search can also enter a "recurrent looping" state that can be described by the following repetitive behaviour. After perturbing a solution s, the cost of the current solution increases for a few iterations but it rapidly decreases afterwords to a cost comparable to $f(s)$—see Appendix B.2 for more technical conditions and exact parameters.[2]

For now, it is enough to say that we try to avoid such looping by exploiting the non-stagnant nature of CG, *i.e.*, we apply on the current solution s the CG improver described in Section 2.3.2. The only CARP specialization of this permutation-level operator is the fact that we use $nrPiv = m/5$ CG steps to generate each route. The solution s_{new} returned by the CG improver replaces s only if $f(s_{\text{new}}) < 1.5 f(s)$. While this CG improver could be used more frequently, one should be aware it is a blocking operator in the ILS and it can slow down the search.

3.3.3.2 Perturbation and Pool Diversity To perturb the current permutation s, we extract a sequence r from the sequence pool $\mathcal{P}$, we inject r at the beginning of s and we remove from s any duplicate element of r. The pool $\mathcal{P}$ is continually updated throughout the search, by adding routes from primal ILS solutions or from routes constructed by CG pricing. To manage $\mathcal{P}$, we need to address two aspects:

deciding which routes to insert into $\mathcal{P}$ Each new route r discovered by CG or ILS is assigned a heuristic cost determined from the classical cost c^r and the similarity between r and other routes from $\mathcal{P}$. The similarity $\texttt{sim}(r, r')$ is calculated as the number of edges $i, j \in E_R$ that arise on consecutive positions in both sequences r and r'. The heuristic cost of r is given by the value c^r, plus a penalty below:

[2]Once such looping is detected, one can try many techniques to overcome it. As such, we also execute an iteration with a neighborhood size three times larger than usually (see Appendix B.1 for details on how to control the neighborhood size).

  - If $\mathtt{sim}(r, r') = |r| - 1$ for some $r' \in \mathcal{P}$, we add a prohibitively-large value to the heuristic cost of r (to ignore duplicates);
  - If $\mathtt{sim}(r, r') \geq 4/5 \cdot |r| - 1$ for some $r' \in \mathcal{P}$ and $|r| > 5$, a simple penalty term (100000) is added to the heuristic score (to discourage limited diversity routes);

  If the heuristic score of r is better than the heuristic score of some $r_{\text{out}} \in \mathcal{P}$, we remove r_{out} and insert r. This rule is slightly enriched with two principles: (i) never remove a unique route containing some $i \in [1..m]$ and (ii) always insert a route that contains some $i \in [1..m]$ not covered by other routes from $\mathcal{P}$. The size of $\mathcal{P}$ is fixed to $\min(100, m)$.

selecting from $\mathcal{P}$ the routes to inject into the current solution s The first extracted route is chosen as follows: randomly pick up four routes from $\mathcal{P}$ and choose the route r with the minimum heuristic cost such that the following similarity condition is verified: $\mathtt{sim}(r, r^s) < |r|/2$ for any route r^s in s. The goal is to avoid injecting into s routes that are too similar to existing routes of s. Afterwards, if we need to extract an additional route (recall that stronger perturbations require more routes), we select the route r_{nxt} that brings the maximum number of new elements into s, while still verifying the above similarity condition $\mathtt{sim}(r_{\text{nxt}}, r^s) < |r|/2$ for any r^s in s.

Finally, this perturbation strategy is applied in the same manner for feasible and *infeasible permutations*, *i.e.*, permutations for which there is no decoded solution with with k vehicles or less (see Remark 3, Section 5.2). However, after 3 perturbations that do not allow the search process to exit the penalized space area, our ILS applies a complete random restart.

4 Fast Arc-Routing Pricing

Our pricing routine mainly relies on the Dynamic Programming (DP) scheme from [19]. The advantage of this scheme is that its time complexity does *not* depend on any quadratic term such as $|V|^2$. More exactly, it uses an asymptotic running time of only $O(Q \cdot (|E| + |V|log(|V|)))$, by exploiting the sparsity of typical instances that usually verify $|E| \ll |V|^2$.

We first recall this scheme below (Section 4.1). Then, we present two acceleration techniques that can speed-up the convergence by factors of tens or even hundreds (Section 4.2). Finally, Appendix C presents two cycle prevention techniques, used for avoiding sending routes with cycles to the ILS process. We propose both an exact technique to avoid 2-cycles and a heuristic approach for cycles of length 20. While the main CG process needs an exact pricing, the CG improver can use a heuristic cycle avoidance, because it does not fully converge to the optimum of the initial CG model.

4.1 The Sparsity-Exploiting Dynamic Programming

We recall from Section 3.2 that the pricing sub-problem requires finding a route r (of cost c_a^r and incidence vector $\mathbf{a}^r$) such that the associated column $[c_a^r \ \mathbf{a}^r]^\top$ minimizes $c_a + \mu - \mathbf{a}^\top \mathbf{y}$ over all feasible columns $[c_a \ \mathbf{a}]^\top \in \mathcal{A}$. The dual variables are: μ for the fleet size constraint and $\mathbf{y}$ for the set-covering constraints (we will note y_e the dual variable of $e \in E_R$). Given any $v \in V$ and $q \in [0..Q]$, we define a DP state (v, q) for all open routes ending at v (the vehicle is not yet returned to depot) that delivered a total service (supply) of q. We note $f(v, q)$ the cumulative reduced cost of any open route in state (v, q); $g(v, q)$ is the cumulative reduced cost of any open route in state (v, q) such that the vehicle reached v by servicing some edge.

The main steps of the DP algorithm can be written as follows:

```
1. initialize $f(v,q)$ and $g(v,q)$ to a prohibitively-large value, $\forall v \in V, \forall q \in [0..Q]$
2. call Dijkstra's algorithm to compute $f(v,0)$ for all $v \in V$     $\triangleright$ simply set $f(v,0) = s_{\text{path}}(v_{\text{dep}}, v) + \mu$
3. for $q$ in $0, 1, \ldots, Q$
   (a) for each $e = \{v_a, v_b\} \in E_R$ such that $q + q_e \leq Q$     $\triangleright$ Perform service on edge $e$
         – if $f(v_a, q) + d_e - y_e < g(v_b, q + q_e)$     $\triangleright$ Traversal sense $v_a \to v_b$
               $g(v_b, q + q_e) \leftarrow f(v_a, q) + d_e - y_e$
         – if $f(v_b, q) + d_e - y_e < g(v_a, q + q_e)$     $\triangleright$ Traversal sense $v_b \to v_a$
```

```
        $g(v_a, q + q_e) \leftarrow f(v_b, q) + d_e - y_e$
    (b) call Dijkstra's algorithm to compute $f(v, q)$ for all $v \in V$      ▷ Perform dead-heading (see below)
4. **if** $\min_{q\in[1..Q]} f(v_{\text{dep}}, q) < 0$, return the corresponding column
```

We stress the fact that the above routine construct routes by intertwining a number of service and dead-heading (no-service) vehicles moves, *i.e.*, steps 3.(a) and 3.(b). The complexity factor $(|E| + |V|log(|V|))$ is due to calling Dijkstra's algorithm in Step 3.(b). This step actually works on an extended version of graph G constructed as follows: insert an artificial source vertex v_s and an edge $\{v_s, v\}$ weighted by $g(v, q)$ for each $v \in V$. By computing the shortest path from v_s to each other vertex v, one obtains a path $v_s \longrightarrow v' \xrightarrow{\text{dhead}} v$ composed of two segments that represent: (i) an open route that delivers a quantity of q and arrives in v' by servicing the last edge, and (ii) a dead-heading shortest path connexion from v' to v. The sum of the costs of these segments $g(v', q) + s_{\text{path}}(v', v)$ determines $f(v, q)$. For further detail, we refer the reader to [19, § 3.1].

4.2 New Pricing Acceleration Techniques

The first goal of this section is to minimize the asymptotic running time $O(Q\cdot(|E|+|V|log(|V|)))$ of the above pricing scheme, *as well as its practical* running time. This is particularly important for the CG improver, because it can slow down the ILS algorithm. First, by avoiding some dead-heading moves, we prove that one can obtain a heuristic pricing version (Section 4.2.1) that reduces the complexity factor $(|E| + |V|log(|V|))$. Secondly, we address the Q factor in Section 4.2.2: we reduce the practical running time by making the DP algorithm ignore some (dominated) states. For a fixed $v \in V$, one does not need to record and scan the states (v, q) for *all* $q \in [0..Q]$, because certain values of q yield dominated states.

4.2.1 Avoiding Deadheading

The first acceleration technique aims at reducing the number of calls to Dijkstra's algorithm in Step 3.(b), as this represents the most time consuming step. As hinted above, the goal of this step is to generate new states by linking states (v', q) to states (v, q) by a dead-heading move from v' to v. To reduce such computationally intensive calculations, we only execute Step 3.(b) with probability $1/p_{\text{skip}}$, where p_{skip} is a parameter (the initial value is $p_{\text{skip}} = m$). The number of potential dead-heading moves is thus divided by p_{skip} in average. This way, as long as $p_{\text{skip}} > 1$, the pricing algorithm actually returns a heuristic solution. However, p_{skip} is iteratively decremented each time this heuristic pricing finds no negative reduced cost column. At the last CG iteration, the p_{skip} value is always 1, so as to eventually remove any heuristic behaviour from our pricing. The CG can only finish when this exact pricing version (with $p_{\text{skip}} = 1$) finds no negative reduced cost column.

However, the exact pricing is only needed at the end of the search; most other CG iterations can use the heuristic pricing that reduces the complexity factor $(|E| + |V|log(|V|))$. For instance, if above Step 3.(b) is never called, the resulting pricing calculation require an asymptotic running time of only $O(Q \cdot |E_R|)$.

Despite its simplicity, this technique yielded the most important speed-up on the total CG convergence, *i.e.*, the total time can be reduced by factors of tens or even hundreds—see Table 2, Section 6.2. This comes from the fact that a nearly-optimal solution of the CG model could often be constructed from routes with limited dead-heading. This also has a stabilization effect in the way columns are generated: the complete pricing algorithm is only executed towards the end of the search, when the dual values are close to optimal.

4.2.2 A Mixed Tree-Array Data Structure for Fast Recording of Non-Dominated States

Certain states of this DP scheme are dominated by others and this can be used to reduce the above complexity factor Q. Consider two states (q_1, v) and (q_2, v). If $q_1 < q_2$ and $f(v, q_1) < f(v, q_2)$ then state (v, q_1) realizes a better performance and consumes fewer resources than (v, q_2). State (v, q_2) can be considered dominated (useless), because any transitions that can be triggered from (v, q_2) can also be triggered from (v, q_1) and they lead to better performances from (v_1, q_1). To avoid dominated states, we propose to consider for each

$v \in V$ only a list of states states $(v, q_1),\ (v, q_2), \ldots$ for which the following hold :

$$\begin{array}{ccccccc} q_1 & < & q_2 & < & q_3 & < \ldots & \text{and} \\ f(v,q_1) & > & f(v,q_2) & > & f(v,q_3) & > \ldots & \end{array} \tag{4.1}$$

Besides using an array of size Q to record all states of v, we use a logarithmic-time red-black tree that maintains a sorted list of states that verify above conditions. We recall that a red-black tree can record a list of sorted numbers such that any insertion or removal can be performed in logarithmic time [10]. In our case, the red-black tree maintains the quantities $q_1, q_2, q_3, \ldots$ sorted such that $q_1 < q_2 < q_3 \ldots$. The asymptotic running time of a read access to a state of the red-black tree is logarithmic in the number of states already recorded. However, by recording at the same time these states $(v, q_1), (v, q_2), \ldots$ in an array indexed by $[0..Q]$, we can access $f(v, q)$ in constant time. Other read access operations (*e.g.*, finding i such that $q_i < q < q_{i+1}$) are performed using the red-black tree.

Each newly discovered state (v, q) can actually be ignored (removed) if it is dominated by some other state. To obtain this behaviour, one can record (v, q) in the array and tentatively introduce (v, q) in the red-black tree at its appropriate position, *i.e.*, between some (v, q_i) and (v, q_{i+1}) such that $q_i < q < q_{i+1}$ (the special case $q_i = q$ is addressed below). If $f(v, q_i) \le f(v, q)$, then the new state (v, q) is actually dominated by state (v, q_i) and we immediately remove (v, q) from both data structures. Otherwise, the states (v, q_i) and (v, q) do not dominate one another and (v, q) is accepted.

Furthermore, if (v, q) dominates other existing states, these states need to be removed. Indeed, if $f(v, q) \le f(v, q_{i+1})$, the new state dominates (v, q_{i+1}) and we remove state (v, q_{i+1}). Furthermore, if state (v, q_{i+2}) also verifies $f(v, q) \le f(v, q_{i+2})$, we also remove (v, q_{i+2}). In fact, we iteratively remove all states $(v, q_{i+1}),\ (v, q_{i+2}),\ (v, q_{i+3})$ up to the first state non-dominated by q (or up the end of the list).

Finally, we address the case in which there is already some state (v, q_i) such that $q_i = q$. If $f(v, q_i) \le f(v, q)$, we remove the new state (v, q); otherwise, the new state dominates the old one and we remove as above all states $(v, q_i), (v, q_{i+1}),\ (v, q_{i+2}), \ldots$ dominated by (v, q).

We observe that each above operation requires individually a logarithmic (or constant) asymptotic time. In theory, the use of this structure should *increase* the total asymptotic running time by a logarithmic factor. However, despite this theoretical slowdown, the reduction of the number of states can actually make the practical running time *decrease*. Indeed, the fact that we can choose not to record a state can trigger a cascading effect and later prune many other states. The experiments from Section 6.2 show that the use of this red-black tree can actually reduce the total convergence time by factors larger than 2 (up to 8 for instances with $Q > 20000$). However, to draw firm conclusions on this speed-up, one would need more comprehensive experiments and this lies outside the scope of the current paper.

5 From Permutations to Routes: Decoding and Post-Decoding

5.1 Overview and Related Work

The Ulusoy's *split* algorithm [33], one of the earliest CARP heuristics, generates an explicit solution by splitting a *giant tour* into a set of feasible routes. A giant tour is a unique route that services *all* edges by (very probably) violating the capacity constraint. It can be represented as a permutation with orientations *i.e.*, a list of oriented edges $(s_1, t_1), (s_2, t_2), \ldots (s_m, t_m)$ that encode the following vehicle moves: $v_{\text{dep}} \xrightarrow{\text{dhead}} s_1 \xrightarrow{\text{srv.}} t_1 \xrightarrow{\text{dhead}} s_2 \xrightarrow{\text{srv.}} t_2 \ldots \xrightarrow{\text{dhead}} s_m \xrightarrow{\text{srv.}} t_m \xrightarrow{\text{dhead}} v_{\text{dep}}$, where "$\xrightarrow{\text{srv.}}$" stands for a serviced edge and "$\xrightarrow{\text{dhead}}$" represents a non-service shortest path. The use of *split* has become increasingly frequent over the last decades and the permutations with orientations has become one of the most successful indirect encodings in the CARP meta-heuristic literature [17, 18, 32, 12]. For more information on related splitting routines, we refer the reader to the recent survey [30].

To interpret `Arc-Routing` as a *permutation set-covering* problem (see Definition 1, Section 2), we need to encode any CARP solution as a classical (non-oriented) permutation $s : [1..m] \to [1..m]$, also written $s = (s_1, s_2, \ldots s_m)$ To turn such s into an explicit CARP solution, we apply two steps. First, we use an exact decoder (Section 5.2) that finds the best explicit CARP solution that services all edges in the order $s_1, s_2, \dots s_m$. Secondly, this explicit solution is further improved by a deterministic post-decoding operator (Section 5.3). This way, any given permutation s is always mapped to a unique CARP solution.

The exact decoder relies on a Dynamic Programming (DP) routine that builds a state graph G_S on which the best explicit solution is calculated as a constrained shortest path. Each vertex of G_S represents a vehicle state such as : "the vehicle returned to depot after servicing edge i" or "edge i has just been serviced in a given direction". Our DP routine has the same complexity as *split*, considering, for instance, the implementation discussed in [29, §2.2].

The general idea of transforming non-oriented lists of edges into routes is not new. The guided local search from [5] describes a move in which a sequence of edges is relocated such that the best traversal direction is determined afterwords. A very related technique is *split with flips* [29] that generalizes the classical (oriented) *split* by determining the best edge directions. This *split with flips* decoder was already used in Grasp and ILS algorithms, but tested with very short computing times. Transformations of edge flows into routes also arise in the exact algorithm [6], although their approach is devoted to individual routes. However, to our knowledge, this is the first study that provides a full pseudocode, a detailed complexity discussion and an investigation of exceptional cases for this transformation of permutations into CARP solutions.

Regarding the deterministic post-decoder, its goal is to improve the explicit solution returned by the above decoder by applying faster modifications on explicit routes. For instance, a simple route operation consists of swapping two edges: this only requires disconnecting the entering and the exit ends of the swapped edges and re-connecting them to new positions. We will also discuss more refined route operations, such as the famous `2-Opt` and the `Cross-Exchange` operator used for both VRP [31] and CARP [26].

5.2 The Exact Decoder

Given input permutation $s = (s_1, s_2, \dots s_m)$, the decoder determines a set of routes of minimum total cost that service all required edges E_R in the order $s_1, s_2, \dots s_m$. To lighten the notation, we use the following convention: we consider that the required edges $E_R = [1..m]$ are indexed in such a way that the input permutation is $(s_1, s_2, \dots s_m) = (1, 2, \dots m)$. This does not restrict the generality: a simple *relabeling* of E_R can always make any input permutation equal to $(1, 2, \dots m)$. The decoder first constructs a state graph G_S (Section 5.2.1); the best explicit solution is calculated as a constrained shortest path on G_S (Section 5.2.2).

5.2.1 Constructing a State Graph

We need an edge-indexed vertex notation. We consider that each required edge $i \in [1..m]$ has two end vertices $[i:0]$ and $[i:1]$. As such, any vertex can be denoted as an end point $[i:o]$ of some edge i (with $o \in \{0,1\}$). We define a *state graph* G_S with two types of vertices (states):

$2m$ ***open*** **states** $[i:o]$ that represent an open route in which the vehicle has just arrived at vertex $[i:o]$ and has just serviced edge i, by traversing it from $[i:1-o]$ to $[i:o]$ (with $o \in \{0,1\}$).

$m+1$ ***depot*** **states** $[i]$ indicating route endings and possible beginnings of new routes. In state $[i]$, the vehicle has just returned to the depot after servicing edges $[1..i]$ and is ready to start servicing edge $i+1$. The *depot* states $[0]$ and $[m]$ are the start and end states (vertices) in the constrained shortest path calculations.

One can observe several types of weighted arcs in G_S:

1. arcs $[i:o_1] \to [i+1:o_2]$ between *open* states with a weight of $d_{i+1} + s_{\text{path}}\big([i:o_1], [i+1:1-o_2]\big)$, where d_{i+1} is the length of edge $i+1$ (recall the notations from Section 3.1) and notation $s_{\text{path}}(u,v)$ indicates the shortest path between vertices u, v $(\forall u, v \in V)$.
2. arcs linking *open* states and *depot* states:
   - "return to depot" arcs $[i:o] \to [i]$ with a weight of $s_{\text{path}}\big([i:o], v_{\text{dep}}\big)$;
   - "leaving depot" arcs $[i-1] \to [i:o]$, weighted by $s_{\text{path}}\big(v_{\text{dep}}, [i:1-o]\big) + d_i$.

3. path arcs $[i] \to [j]$ between *depot* states (with $i, j \in [0..m]$ and $i < j$). Such an arc represents the shortest G_S path between $[i]$ and $[j]$, calculated using the above two arc types in G_S. This calculation can be carried out using $O(j - i)$ operations (see an example in Appendix D).

The complexity and computational aspects are discussed in Section 5.2.2 below. For the moment, we point out that the above G_S construction requires (pre-)computing the shortest path $s_{\text{path}}(u, v)$ between any two vertices u and v of G. This can be done by applying (only once in advance) an all-pairs shortest path algorithm, *e.g.*, Roy-Floyd-Warshall algorithm of complexity $O(n^3)$ or Johnson's algorithm of complexity $O(nm+n^2 log(n))$ [10]. In fact, these pre-computed shortest paths are also used by the CG pricing algorithm.

5.2.2 Constrained Shortest-Path on the State Graph G_S

We observe that any arc $[i] \to [j]$ between depot states $[i]$ and $[j]$ of G_S ($\forall i, j \in \{0, 1, \dots m\}$) can be seen as a unique route in G. As such, the final decoded solution is a set of routes that we determine as a constrained shortest path from $[0]$ to $[m]$ in G_S; the constraints are the following:

(i) each arc of the form $[i] \to [j]$ has to respect the capacity constraint $q_{i+1} + q_{i+2} + \dots q_j \le Q$. We will enforce this constraint by computing the maximum number of clients `maxClients`$[i]$ that can be serviced by a route that starts at edge $i + 1$ (see Line 2 in Algorithm 2, next page).

(ii) the number of depot states in the constrained shortest path is at maximum k (excluding the initial state $[0]$). This constraint bounds the fleet size to k.

The pseudo-code of the decoding routine is provided by Algorithm 2 next page. It consists of three stages:

stage 1 initialize several array data structures for the arcs of G_S (the comments in Line 3 are self-explanatory);

stage 2 compute the weights of the arcs of G_S, *i.e.*, both the arcs from depot states to open states (structure `depToOpn`) and arcs from depot to depot states (structure `depToDep`);

stage 3 compute the constrained shortest path from $[0]$ to $[m]$ in G_S. This last stage relies on a Dynamic Programming (DP) routine that can actually generate up to k (sub-)states for each depot state $[i]$ of G_S. Each such DP state encodes the minimum cost `constrShPth`$[i][\kappa]$ necessary to service edges $[1..i]$ using exactly κ routes. The DP recursion formula is provided by Line 19 of Algorithm 2 and it is rather self-explanatory. One can remove the second index to lift the fleet size constraint.

5.2.3 Speed-up and Complexity Remarks

The complexity of Algorithm 2 is $O(m \cdot k \cdot \max_{r \in s}(|r|))$, as directly resulting from the three nested `for` loops of the last stage (Lines 16-18), where $\max_{r \in s}(|r|)$ is the maximum number of clients (edges) that can be serviced by a potential route associated to input permutation s. The factor k can actually be reduced both theoretically and practically. In theory, one can reduce it to $log(k)$ as follows: write k in binary as $k = \sum_{\kappa=0}^{\kappa=\lfloor \log(k) \rfloor} b_\kappa 2^\kappa$ and compute all values of the form `constrShPth`$[i][2^\kappa]$ from `constrShPth`$[i][2^{\kappa-1}]$ values, $\forall i \in [1..m]$. However, one can achieve a more practical speed-up using the following two remarks.

Remark 1. *We first run a $O(m \cdot \max_{r \in s}(|r|))$ simplified decoder version with no fleet size constraint. This simplified version is constructed from Algorithm 2 as follows: remove the* `for` *loop from Line 17 and transform* ***constrShPth*** *into a one-dimensional array (remove the second κ index). This way, Algorithm 2 uses an unlimited number of vehicles and returns an explicit CARP solution of any fleet size. If this solution uses more than k routes, then the complete $O(m \cdot k \cdot \max_{r \in s}(|r|))$ decoder is still needed. Otherwise, the simplified decoder alone is sufficient.*

Experiments suggest that this simplified decoder returns a solution with no more than k routes in roughly half of the cases (see Appendix A) . However, for certain instances, the fleet size constraint is very difficult to satisfy, and so, the simplified decoder can actually fail too often. For such instances, `CG-P-ILS` can very frequently end up calling both decoders. As such, if `CG-P-ILS` detects that this unwanted situation arises in more than $\frac{1}{k}$ of cases, it disables the simplified decoder. □

Algorithm 2 The Decoding Routine

```
Require: a service order given by permutation $s = (1, 2, 3, \dots m)$    ▷ without restricting generality
Ensure: minimum cost needed to service all required edges in the order $1, 2, \dots, m$
   Stage 1: Initialize data
 1: for $i \leftarrow 1$ to $m$ do
 2:     $\texttt{maxClients}[i] \leftarrow \max\{\delta\colon\ q_{i+1} + q_{i+2} + \cdots + q_{i+\delta} \le Q\}$ ▷ nr edges that can be serviced from $i+1$ on
 3:     Initialize the following with a very large value $M$
          – $\texttt{depToDep}[i][j] \leftarrow M$, for all $j \in \big[i+1..i+\texttt{maxClients}[i]\big]$    ▷ weights of $G_S$ arcs $[i] \to [j]$
          – $\texttt{depToOpn}[i][j,o] \leftarrow M$ for all $j \in \big[i+1..i+\texttt{maxClients}[i]\big], o \in \{0,1\}$    ▷ arcs $[i] \to [j:o]$
          – $\texttt{constrShPth}[i][\kappa] \leftarrow M$ for all $\kappa \in \big[0..k\big]$   ▷ constrained shortest path $[0] \to [i]$ using $\kappa$ routes
 4: end for
 5: $\texttt{constrShPth}[0][0] \leftarrow 0$    ▷ initial state: serviced 0 edges with 0 vehicles at 0 cost
   Stage 2: Compute depToDep and depToOpn (the arcs of $G_S$)
 6: for $i = 0$ to $m-1$ do    ▷ $i = 0$ indicates that nothing was serviced yet
 7:     $\texttt{depToOpn}[i][i+1,0] \leftarrow s_{\text{path}}(v_{\text{dep}}, [i+1,1]) + d_{i+1}$    ▷ $s_{\text{path}}(u,v) =$ shortest path from $u$ to $v \in V$
 8:     $\texttt{depToOpn}[i][i+1,1] \leftarrow s_{\text{path}}(v_{\text{dep}}, [i+1,0]) + d_{i+1}$    ▷ $d_{i+1} =$ length of edge $i+1$
 9:     $\texttt{depToDep}[i][i+1] \leftarrow \min\big(\texttt{depToOpn}[i][i+1,0] + s_{\text{path}}([i+1,0], v_{\text{dep}}),$    ▷ depot-to-depot arcs in $G_S$
                $\texttt{depToOpn}[i][i+1,1] + s_{\text{path}}([i+1,1], v_{\text{dep}})\big)$
10:     for $\delta = 2$ to $\texttt{maxClients}[i]$ do
11:         $\texttt{depToOpn}[i][i+\delta,0] \leftarrow \min\big(\texttt{depToOpn}[i][i+\delta-1,0] + s_{\text{path}}([i+\delta-1,0],[i+\delta,1]) + d_{i+\delta}),$
                $\texttt{depToOpn}[i][i+\delta-1,1] + s_{\text{path}}([i+\delta-1,1],[i+\delta,1]) + d_{i+\delta})\big)$
12:         $\texttt{depToOpn}[i][i+\delta,1] \leftarrow \min\big(\texttt{depToOpn}[i][i+\delta-1,0] + s_{\text{path}}([i+\delta-1,0],[i+\delta,0]) + d_{i+\delta},$
                $\texttt{depToOpn}[i][i+\delta-1,1] + s_{\text{path}}([i+\delta-1,1],[i+\delta,0]) + d_{i+\delta})\big)$
13:         $\texttt{depToDep}[i][i+\delta] \leftarrow \min\big(\texttt{depToOpn}[i][i+\delta,0] + s_{\text{path}}([i+\delta,0], v_{\text{dep}}),$
                $\texttt{depToOpn}[i][i+\delta,1] + s_{\text{path}}([i+\delta,1], v_{\text{dep}})\big)$
14:     end for
15: end for
   Stage 3: Calculate by Dynamic Programming the constrained shortest path from [0] to $[m]$
16: for $i = 0$ to $m-1$ do
17:     for $\kappa = \texttt{minVeh}[i]$ to $\texttt{maxVeh}[i]$ do    ▷ see Remark 2 for setting the values $\texttt{minVeh}[i]$ and $\texttt{maxVeh}[i]$
18:         for $\delta = 1$ to $\texttt{maxClients}[i]$ do
19:             $\texttt{constrShPth}[i+\delta][\kappa+1] \leftarrow \min(\texttt{constrShPth}[i+\delta][\kappa+1],$    ▷ Main DP recursion
                    $\texttt{constrShPth}[i][\kappa] + \texttt{depToDep}[i][i+\delta])$
20:         end for
21:     end for
22: end for
23: return $\texttt{constrShPth}[m][k]$    ▷ see Remarks 2 and 3 for the case "no solution with $k$ routes"
```

Remark 2. *While the fleet size constraint accounts for k-fold increase of the asymptotic running time, it has a much lower impact on the practical decoding time, because the interval $\big[\texttt{minVeh}[i], \texttt{maxVeh}[i]\big]$ can be very small in Line 17.*

Indeed, observe that Line 17 of Algorithm 2 does not actually go from 1 to k but from $\texttt{minVeh}[i]$ to $\texttt{maxVeh}[i]$, where $\texttt{minVeh}[i]$ and $\texttt{maxVeh}[i]$ represent the minimum and (resp.) the maximum fleet sizes that are relevant for state $[i]$. A fleet size κ is relevant at state $[i]$ if there exists a path with exactly κ arcs linking [0] to $[i]$ in G_S. In the beginning of the decoding process, most states $[i]$ are associated to very short intervals $[\texttt{minVeh}[i]..\texttt{maxVeh}[i]]$. It is only after an update of $\texttt{constrShPth}[i+\delta][\kappa+1]$ in Line 19 that a fleet size of $\kappa+1$ can be seen as relevant for state $[i+\delta]$, enlarging $[\texttt{minVeh}[i+\delta], \texttt{maxVeh}[i+\delta]]$.

Furthermore, a new DP state with a fleet size of $\kappa+1$ is only useful if it reduces the cost compared to a fleet size of κ, *i.e.*, only if $\texttt{constrShPth}[i][\kappa+1] < \texttt{constrShPth}[i][\kappa]$. This property can be used to further reduce the interval $\big[\texttt{minVeh}[i]..\texttt{maxVeh}[i]\big]$. It is enough to record only a set $\kappa_1, \kappa_2, \kappa_3, \dots$ of relevant fleet sizes, *i.e.*, for which the following hold :

$$\begin{array}{ccccccl} \kappa_1 & < & \kappa_2 & < & \kappa_3 & < \dots & \text{and} \\ \texttt{constrShPth}[i][\kappa_1] & > & \texttt{constrShPth}[i][\kappa_2] & > & \texttt{constrShPth}[i][\kappa_3] & > \dots & \end{array} \tag{5.1}$$

Such sets can be implemented using the mixed tree/array data structure as for (4.1) in Section 4.2.2. □

Remark 3. *For low-quality permutations s, one can not provide all service in the order $s_1, s_2, \dots s_m$ with only k vehicles of capacity Q. Algorithm 2 detects this case by finding there is no fleet size value relevant at state $[m]$, i.e., no value $\texttt{minVeh}[m] < k$. In this case, Algorithm 2 simply returns a very large infeasibility penalty plus the minimum unserviceable quantity, i.e., the minimum amount that can not be serviced using the order imposed by the current permutation.*

This unserviceable quantity can be (heuristically) computed in $O(m)$ time: restrict the values of δ in Lines 10 and 18 to $\delta = \texttt{maxClients}[i]$, *i.e.*, we obtain a version of Algorithm 2 that forces each route to "stretch" to cover as many serviced edges as possible. The service amount remaining unserviced at the end of Algorithm 2 constitutes an upper bound for the unserviceable quantity. In practice, when the ILS process traverses a penalized search space area, the unserviceable quantity is computed by the above heuristic, running the full decoder only if the returned unserviceable quantity is 0. While this heuristic might fail detecting the feasibility of some permutations, the general speed gain seems more important. □

5.3 Deterministic Post-Decoding Operator

While the ILS process (Section 3.3) evolves in the space of permutations, the fastest way to improve a solution consists of performing local modifications on decoded routes. After decoding permutation s, `CG-P-ILS` tries to improve the returned decoded solution using three types of route operations (see below). For this, the post-decoding operator scans all (pairs of) positions on which these route operations can be executed: any modification that can bring a strict improvement is effectively implemented right away. These operations are tried on a fixed order of indices (positions) until no improvement can be done, as in a (deterministic) `buble-sort` loop.

`route rotation` A route r can be seen as a closed walk $v_{\text{dep}} \to v_1 \to v_2 \to \dots v_{|r|} \to v_{\text{dep}}$ containing both serviced and non-serviced edges. Using a similar approach as in [33, §3.3], one can compute in constant time the improvement that can be obtained by re-locating the depot just before any index $i \in [2..|r|]$. More exactly, for any such i, this operation constructs route $v_{\text{dep}} \to v_i \to v_{i+1} \to \dots v_{|r|} \to v_1 \to v_2 \dots \to v_{i-1} \to v_{\text{dep}}$. The cost variation resulting from this modification can be computed in constant time, because one only has to evaluate the cost of re-locating the "connexion" points v_1, v_{i-1}, v_i and $v_{|r|}$. The route rotation can be applied on a linear number $O(m)$ of positions i.

`cross-exchange` Given routes $v_{\text{dep}} \to v_1 \to v_2 \to \dots \to v_{\text{dep}}$ and $v_{\text{dep}} \to v'_1 \to v'_2 \to \dots \to v_{\text{dep}}$, `cross-exchange` takes a segment $v_i \to \dots \to v_{i+\delta}$ of the first route and swaps it with a segment $v'_{i'} \to \dots \to v'_{i'+\delta}$ of the second route. We also consider reversing one of these segments when this improves the cost. For given i, j and δ, the objective function evolution can be calculated in constant time, because one only has to evaluate the cost variation resulting from disconnecting the two segments at their end points and re-connecting them at their new places. `Cross-exchange` can be applied on a total number of $O\left(m^2 \max_{r \in s}(|r|)\right)$ positions, considering all possible choices of the two routes. This is because we only consider connexion points ($v_i, v'_{i'}$, etc.) that represent start vertices of serviced edges. If $\delta = 1$, `cross-exchange` becomes a simple edge swap. This route-level operation was already used and described in other CARP papers [26, Fig. 2].

`2-Opt` Given routes $v_{\text{dep}} \to v_1 \to v_2 \to \dots \to v_{\text{dep}}$ and $v_{\text{dep}} \to v'_1 \to v'_2 \to \dots \to v_{\text{dep}}$, `2-Opt` replaces them with the following two routes: (i) a first route that links a segment $v_{\text{dep}} \to v_1 \to v_2 \to \dots v_i$ to a segment $v'_{i'+1} \to v'_{i'+2} \to \dots \to v_{\text{dep}}$, and (ii) a second route that links $v_{\text{dep}} \to v'_1 \to v'_2 \to \dots v'_{i'}$ to $v_{i+1} \to v_{i+2} \to \dots \to v_{\text{dep}}$. The cost increase induced by this modification can be easily calculated as

$s_{\text{path}}(v_i, v'_{i'+1}) + s_{\text{path}}(v'_{i'}, v_{i+1}) - s_{\text{path}}(v_i, v_{i+1}) - s_{\text{path}}(v'_{i'}, v'_{i'+1})$. The number of potential choices of v_i is proportional to the number of serviced edges in the first route, because v_i needs to be the exit vertex of required edge $\{v_{i-1}, v_i\}$. The same applies to $v'_{i'}$. By reversing one of the routes, we obtain a second version of `2-Opt`, see also [18, Fig. 4] for examples and further `2-Opt` discussions.

After trying all these route operations, the initial permutation s can be transformed into a new permutation s'. In this case, `CG-P-ILS` calls again the decoder on s', because the decoder can better split s' and further improve the cost ; if this happens, the post-decoder operator is also called again. As such, the decoder and the post-decoder are called iteratively until they can no longer strictly improve the cost. In the end, the initial service order can evolve to a new order $\bar{s}_1, \bar{s}_2, \ldots \bar{s}_m$. We say that s and $\bar{s}$ are decoded into the same CARP explicit solution; however, `CG-P-ILS` continues by replacing s with $\bar{s}$.

Finally, observe that numerous other route-level operations could have been applied during this post-decoding, such as the repair operator from [24], or *Shorten* from [15].

Remark 4. *(the case of infeasible permutations) If the input permutation s makes the decoder return an explicit solution with more than k routes (see Remark 3, previous page), then this explicit solution is actually infeasible. In this case, we do not use the post-decoder described above, but a post-decoder variant with a "bin-packing" goal. More exactly, this post-decoder variant tries to make all existing routes reduce their slack (unused capacity), so as to get closer to a feasible CARP solution. For this, we scan the edges from 1 to m and, for each last edge i of a route r^i, we perform the following: find some edge j situated after i in the input permutation such that q_j is smaller or equal to the slack (unused capacity) of r^i and move j after i in r^i. Such edges $j > i$ are thus advanced to earlier service positions, allowing earlier routes r^i to include them in their service. By iteratively repeating this, the slack of earlier routes decreases and the number of later (potentially) unserviceable edges can also decrease.*

6 Numerical Experiments

6.1 Summarized Results and General Trends

Our implementation first launches the ILS process. We fixed the total time limit at $TM_{\max} = m \cdot k \cdot \lceil m/70 \rceil \cdot \lceil m/100 \rceil$, based on the following principles: (i) $m \cdot k$ seconds are usually enough for small graphs ($m \leq 50$), (ii) we observed it can be useful to allow twice as much time for medium-sized graphs ($m \in [70, 100]$), and (iii) larger graphs with $m > 100$ might need quadratically more time. If the best solution if found at some moment between $0.9TM_{\max}$ and $TM_{\max}$, we allow an additional time of $0.1TM_{\max}$. Finally, the main CG process is launched in parallel and we always wait for its full convergence.

Table 1 presents a summary of the results of two `CG-P-ILS` versions on all CARP instances that we are aware of.[3] The first group of 5 rows (besides the heading) is devoted to the standard `CG-P-ILS` and it reports: (i) the average deviation from the best-known solution, (ii) the number of instances for which the optimum solution was found, and (iii) the average CPU time (in seconds) of both the ILS and CG components. The second group (last 3 rows) corresponds to an `CG-P-ILS` version with no CG component at all.

When comparing summarized average gaps, one should be aware that, on certain easier instances, all `CG-P-ILS` variants always report a gap of 0. This has a implicit smoothing effect on the reported *average* gaps, making certain differences less visible at a first glance. For example, 22 of the 23 `gdb` instances are solved to optimality by all `CG-P-ILS` variants, and so, the *average* gap on column `gdb` is generally equivalent to $\frac{1}{23}$ of the gap reported on the 23th instance. Observe that the pure ILS alone can reach a gap of 0 on 105 out of 197 instances (see last "#Hits opt" row).

Table 1 shows that `CG-P-ILS` reports an upper bound UB_{ILS} within 101%`opt` and a lower bound $\text{LB}_{\text{CG}} \geq$ 90%`opt`, where `opt` is either the optimum value or the best known (upper or resp. lower) bound. Furthermore, the average gap between UB_{ILS} and LB_{CG} is at most 10%UB_{ILS}, even for the very large `egl-large` instances

[3]To the best of our knowledge, we use all benchmark sets available in the literature. They are publicly available on-line, see `www.uv.es/~belengue/carp.html` or `logistik.bwl.uni-mainz.de/benchmarks.php`. The later web-site also provides a collection with the best bounds reported by other papers; we used these bounds to determine the average gaps of `CG-P-ILS`.

Alg ver.	Instance set / Number of instances	`kshs` 6	`gdb` 23	`val` 34	`beullens` 100	`egl` 24	`egl-large` 10
standard full `CG-P-ILS`	Avg gap $UB_{ILS}/UB_{bst} - 1$ (in ‰)	0	0	2.5	2.4	2.9	3.0
	Avg ILS time [s.]	0.17	13.00	121.88	184.87	5043	158638
	#Hits optimum or best-known	6	23	28	64	8	0
	Avg gap $LB_{CG}/LB_{bst} - 1$ (in ‰)	-71.4	-48	-95.8	-64.9	-34.7	-39.2
	Avg CG time (incl. CG improver)	0.04	0.11	2.46	18.16	1350	230324
	Avg gap $LB_{CG}/UB_{ILS} - 1$ (in ‰)	-71.4	-48	-98.1	-67.6	-40.1	-79.2
pure ILS (no CG)	Avg gap $UB_{ILS}/UB_{bst} - 1$ (in ‰)	0	0.2	14.8	6.4	7.5	6.4
	Avg ILS time [s.]	0.17	38.96	103.76	149.56	5553	109589
	#Hits optimum or best-known	6	22	26	45	5	0

Table 1: Average optimality gaps and CPU times of two `CG-P-ILS` versions (with or without CG). Let us insist we provide *averages over dozens of instances*: since both versions can reach the best-known bound on half of instances (see rows #Hits), the average gap variation might only be due to the remaining half of instances. See also Appendix A for instance by instance results or Section 6.2 for statistics over several runs.

(see the last of the "standard full `CG-P-ILS`" rows). The comparison between the standard `CG-P-ILS` and the pure ILS confirms the interest in using CG paradigm: the use of CG can reduce the ILS gap by half.

Finally, we discuss more technical implementation details as they can also influence the performance of optimization methods. The ILS process is written in Java and all CG programs are written in C++. The main CG process is launched as an external program in a separate thread of the ILS Java program. This thread executes the associated C++ program, retrieves its standard output and can insert new sequences into the pool $\mathcal{P}$. Since this pool is shared with the main ILS Java thread, all read and write operations on $\mathcal{P}$ are guarded by a mutex. This solves any synchronisation issue between the main CG process and the ILS process, because there is no other interaction between these two processes. However, when a CG improver is launched, the ILS process waits for its completion, because the CG improver can modify the current ILS solution. The CPU times are obtained on an Intel Xeon processor clocked at 2.33GHz under Debian Linux.

6.2 Evaluating the Impact of CG over Several Runs

We now evaluate: (i) the speed gain realized by the acceleration techniques in CG, and (ii) the impact of the CG mechanisms and of the post-decoder in the ILS.

For this, we perform for each instance several runs of different `CG-P-ILS` variants. So far, we only presented summarized results, because the whole CARP benchmark set is relatively large and we are not interested in the precise bounds of particular instances, but rather in trends that show up across many instances. However, since we here perform *several runs of several CG-P-ILS variants per instance*, we prefer to report instance-by-instance results, using a smaller benchmark set. We voluntarily selected seven instances on which `CG-P-ILS` exhibits a relatively large performance variation. We consider the same maximum running time as above, but we used a slightly faster machine than in Table 1 (*i.e.*, an Intel Core I7-950 processor clocked at 3.07GHz under Suse Linux).

We first analyze (Table 2) the impact of the CG acceleration techniques. It is due to these techniques that the CG process could be fast enough to be effectively integrated within the ILS for the larges instances. Table 2 reports a comparison of the running time of several CG versions. Since the main CG process always reports the same optimum value of the CG model, we only compare the running times. We observe that:

- the dead-heading avoidance technique (Section 4.2.1) substantially improves the convergence speed. Without this component, the search can be hundreds of times slower.
- the use of the data structure from Section 4.2.2 can double the speed, as it rapidly prunes dominated states in the DP pricing scheme. Other experiments on the `egl-large` instances with $Q > 20000$ showed that the use of this structure can make the speed increase by a factor of up to 8.

Instance/LB_{CG}	Complete CG version with all proposed options	No deadheading avoidance (§4.2.1)	No deadheading avoidance (§4.2.1) No red-black tree (§4.2.2)
gdb8/330	0.081	1.02	1.1
val9D/357	1.16	153.0	249.0
C09/5003	2.42	273.0	419.0
D23/2893	61.1	2652.85	3495.36
egl-e3-C/9954	2.95	430.0	815.0
egl-e4-C/11137	3.21	735.0	1092.67
egl-s1-C/8267	2.31	120.0	260.0

Table 2: The convergence time needed by three CG variants (averages over 10 runs, in seconds). For comparison, the original implementation of sparsity-exploiting DP (on which our approach is based) required around 400 seconds in average on the `egl` instances [19].

Table 3 below compares the final cost values reported by ten runs of the following algorithms: the standard full `CG-P-ILS` (Columns 2-4), `CG-P-ILS` without CG improver (Columns 5-7), `CG-P-ILS` with no CG component at all (Columns 8-10) and `CG-P-ILS` with no deterministic post-decoder (last 3 columns). Several conclusions can be drawn from this table.

- the last 3 columns show that the deterministic post-decoder operator is clearly extremely useful. Corroborating information from other CARP papers, it seems that it is difficult to reach very competitive results by only working in a high-level permutation space.
- the CG improver can be evaluated by comparing the version "Standard `CG-P-ILS`" with the version "No CG Improver". This shows that the CG improver brings an improvement on the average reported cost (compare Column 4 to Column 7, in boldface) and also on the general success rate (compare Column 2 to Column 5).
- the impact of the CG process can be evaluated by observing that the version "No CG Improver" reports lower average costs than "No CG at all" (compare Columns 7 to Column 10, in boldface).

Instance/optimum or	Standard `CG-P-ILS`			No CG Improver			No CG at all			No Post-Decoding		
best upper bound	bst(#bst)	max	avg	bst(#bst)	max	avg	bst(#bst)	max	avg	bst(#bst)	max	avg
gdb8/348	0 (10)	0	**0.0**	0 (6)	2	**0.8**	0 (6)	2	**0.8**	0 (1)	2	**2.2**
val9D/388	3 (10)	3	**3.0**	3(4)	5	**3.8**	3(5)	6	**3.7**	16(1)	25	**18.4**
C09/5260	0 (1)	40	**21.5**	5 (1)	60	**26.5**	5 (1)	60	**36.5**	100 (2)	120	**111.0**
D23/3130	0 (4)	10	**6.0**	0 (1)	25	**10.0**	10 (2)	25	**15.0**	45 (1)	440	**264.0**
egl-e3-C/10292	13 (3)	47	**33.8**	17 (3)	66	**39.0**	17 (1)	73	**45.3**	46 (1)	110	**84.6**
egl-e4-C/11550	8 (1)	77	**44.5**	25 (1)	122	**69.8**	33 (1)	136	**86.5**	169 (1)	444	**289.0**
egl-s1-C/8518	0 (10)	0	**0.0**	0 (8)	23	**20.2**	0 (6)	28	**23.5**	42 (1)	120	94.4

Table 3: Deviation from the optimum (or the best known bound) of the upper bounds reported by four `CG-P-ILS` variants over ten runs on different instances. For each variant, we report best (smallest) deviation from the best-known solution (bst), the number of runs reaching this deviation (#bst), the average result over ten runs (avg), as well as the worst deviation ever reported at the end of a run (max).

6.3 Comparisons with the CG and LS Literature

The CARP generated a large amount of work in the last couple of decades: important progress has been made and a dozen of new algorithms have been recently proposed. Since the goal of this paper is not to perform a comprehensive survey of such a vast literature, we only compare `CG-P-ILS` with the most related CG or ILS work. More exactly, Table 4 below compares our ILS bounds with those of CARPET [15], the two Variable Neighborhood Search algorithms VNS1 [16] and VNS2 [26], the two Tabu Search (TS) methods TS1 and TS2 from [7], the TS enhanced with a repair operator (TS-RepOp) from [24], the ILS acting on a transformation of CARP into VRP (ILS-VRP) from [23] and the ILS with short running times from [29]. This latter algorithm is the most similar to our ILS, because it uses a decoder "split with shifts" that acts on permutations (although this decoder is not used on certain instances such as `egl`).

Regarding the lower bounds, we compare with the following CG work: the sparsity-exploiting CG with elementary routes (CG-ElemR) from [19], the CG with cuts generated by a dual ascent heuristic (CG-Cuts) from [21], the results at the root node of the Branch-and-Cut-and-Price (BCP) from [22], and the Integer Ray Method (IRM) for CG optimization and lower bounding from [27].

	Algorithm	`kshs`	`gdb`	`val`	`beullens`	`egl`	`egl`-large
$\frac{UB_{LS}-UB_{bst}}{UB_{bst}}$	`CG-P-ILS`	0.00%	0.00%	0.25%	0.24%	0.29%	0.30%
	CARPET [15]	–	0.35%	0.51%	–	–	–
	VNS1 [16]	–	0.3%	0.85%	–	–	–
	VNS2 [26]	–	–	0.00%	–	0.09%	–
	GLS [5]	–	–	–	0.03%	–	–
	TS1 [7]	–	0.46%	0.74%	1.71%	1.50%	2.17%
	TS2 [7]	–	0.07%	0.13%	0.15%	0.72%	–
	TS-RepOp [24]	–	0.00%	0.12%	0.07%	0.47%	0.99%
	ILS-VRP [23]	–	–	–	–	–	0.08%[a]
	ILS-ShortTime [29]	–	0.24%	0.47%	–	1.23%	–
$\frac{LB_{CG}-LB_{bst}}{LB_{bst}}$	`CG-P-ILS`	-7.14%	-4.80%	-9.58%	-6.49%	-3.47%	-3.86%
	CG-ElemR [19]	-6.85%	-4.76%	-1.71%	–	-2.14%	–
	CG-Cuts [21]	-0.00%	-0.06%	-0.61%	–	-0.89%	–
	BCP-root node [22]	–	–	–	–	-0.5%	–
	IRM [27]	-7.54%	-9.07%	-14.39%	–	-37.45%	–

[a]This deviation is calculated with regards to the best upper bounds that do include the new values discovered by `CG-P-ILS`.

Table 4: The average gap of LS and CG algorithms from the best-known (or optimum) solution.

Given the large number of CARP algorithms developed over the last decade, one can say that `CG-P-ILS` does reach quality results compared to other LS or CG methods. We do not claim that it systematically outperforms on medium-sized graphs all best recent exact methods [3, 6]. However, it is worthwhile mentioning that `CG-P-ILS` did find new upper bounds for the instances `egl-g1-C`, `egl-g1-E`, `egl-g2-A`, `egl-g2-C` and `egl-g2-E`, see Appendix A for exact figures.[4] On several widely-used `egl` instances, the quality of the `CG-P-ILS` upper bounds has only been reached before by 2 or 3 algorithms out of a dozen of papers (compare the results on `egl-s2-A`, `egl-s4-A` and `egl-s4-B` from Appendix A with those at `logistik.bwl.uni-mainz.de/benchmarks.php`). If we consider a "VRP-free" comparison (*i.e.*, excluding the algorithm [23] based on a CARP-to-VRP transformation), one can say that `CG-P-ILS` improves the best-known CARP upper bound on almost all `egl-large` instances (all except `egl-g2-D`).

7 Conclusions

We described two techniques for combining Iterated Local Search (ILS) with Column Generation (CG) for permutation problems and `Arc-Routing`:

– the ILS process is intertwined with a CG process: the two processes run in parallel and the best sequences (routes) discovered by CG can be inserted into the current ILS solution during the perturbation

[4]See also `cedric.cnam.fr/~porumbed/carp/` for different explicit CARP solutions for individual instances.

phase. This aims at making the ILS take profit from (some of) the dual nature of the CG models.

– we use a CG-Improver operator that starts from the current ILS solution and tries to improve it by performing several CG iterations. The goal is to increase the diversity of the solutions visited by the ILS and to avoid to repeatedly loop on the same plateaux and local optima.

Mixing ILS and CG this way can be very useful for two reasons : (i) the use of CG sequences in the ILS can improve the quality of the (CG perturbed) solutions visited by the ILS *and* (ii) the CG process naturally reports a valid lower bound at the end of its convergence.

The first half of the paper presented the main ideas in a *permutation set-covering* context, *i.e.*, essentially using notions of sequences, permutations and set-covering. The second half is devoted to more specific CARP-customized operators. For instance, we presented several pricing acceleration ideas that could reduce the CG convergence time by factors of tens or even hundreds, *e.g.*, the most important is to avoid generating routes with dead-heading as long as possible before the final CG iterations. Based on such acceleration ideas, we could apply CG techniques for the first time on the largest CARP instances. Since the CG improver is used as a blocking operator inside the ILS process, the CG speed can strongly influence the speed of the ILS process as well.

Acknowledgements We thank the two referees for the work they performed on our paper. We thank Pierre Cubaud for providing us access to a machine on which we carried out several preliminary tests.

References

[1] C. Archetti and M. Speranza. A survey on matheuristics for routing problems. EURO Journal on Computational Optimization, 2(4):223–246, 2007.

[2] R. Baldacci and V. Maniezzo. Exact methods based on node-routing formulations for undirected arc-routing problems. Networks, 47(1):52–60, 2006.

[3] E. Bartolini, J.-F. Cordeau, and G. Laporte. Improved lower bounds and exact algorithm for the capacitated arc routing problem. Mathematical Programming, 137(1-2):409–452, 2013.

[4] J. M. Belenguer and E. Benavent. A cutting plane algorithm for the capacitated arc routing problem. Computers & Operations Research, 30(5):705 – 728, 2003.

[5] P. Beullens, L. Muyldermans, D. Cattrysse, and D. Van Oudheusden. A guided local search heuristic for the capacitated arc routing problem. European Journal of Operational Research, 147(3):629–643, 2003.

[6] C. Bode and S. Irnich. Cut-first branch-and-price-second for the capacitated arc-routing problem. Operations Research, 60(5):1167–1182, 2012.

[7] J. Brando and R. Eglese. A deterministic tabu search algorithm for the capacitated arc routing problem. Computers & Operations Research, 35(4):1112 – 1126, 2008.

[8] V. Cacchiani, V. Hemmelmayr, and F. Tricoire. A set-covering based heuristic algorithm for the periodic vehicle routing problem. Discrete Applied Mathematics, 163(0):53 – 64, 2014.

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

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms. MIT press Cambridge, 2001.

[11] M. Dror. Arc routing: theory, solutions, and applications. Kluwer Academic Pub, 2000.

[12] H. Fu, Y. Mei, K. Tang, and Y. Zhu. Memetic algorithm with heuristic candidate list strategy for capacitated arc routing problem. In IEEE Congress on Evolutionary Computation, pages 1–8, 2010.

[13] P. Galinier and J. Hao. Hybrid evolutionary algorithms for graph coloring. Journal of Combinatorial Optimization, 3(4):379–397, 1999.

[14] P. Greistorfer. A tabu scatter search metaheuristic for the arc routing problem. Computers & Industrial Engineering, 44(2):249–266, 2003.

[15] A. Hertz, G. Laporte, and M. Mittaz. A tabu search heuristic for the capacitated arc routing problem. Operations Research, 48(1):129–135, 2000.

[16] A. Hertz and M. Mittaz. A variable neighborhood descent algorithm for the undirected capacitated arc routing problem. Transportation Science, 35(4):425–434, 2001.

[17] P. Lacomme, C. Prins, and W. Ramdane-Chérif. A genetic algorithm for the capacitated arc routing problem and its extensions. In E. Boers, editor, Applications of Evolutionary Computing, volume 2037 of Lecture Notes in Computer Science, pages 473–483. Springer Berlin Heidelberg, 2001.

[18] P. Lacomme, C. Prins, and W. Ramdane-Chérif. Competitive memetic algorithms for arc routing problems. Annals of Operations Research, 131(1-4):159–185, 2004.

[19] A. Letchford and A. Oukil. Exploiting sparsity in pricing routines for the capacitated arc routing problem. Computers & Operations Research, 36:2320–2327, 2009.

[20] H. Longo, M. P. de Aragão, and E. Uchoa. Solving capacitated arc routing problems using a transformation to the CVRP. Computers & Operations Research, 33(6):1823–1837, 2006.

[21] R. Martinelli, D. Pecin, M. Poggi, and H. Longo. Column generation bounds for the capacitated arc routing problem. In XLII SBPO, 2010.

[22] R. Martinelli, D. Pecin, M. Poggi, and H. Longo. A branch-cut-and-price algorithm for the capacitated arc routing problem. In Proceedings of the 10th International Symposium of Experimental Algorithms (SEA), volume 6630 of LNCS, pages 315–326. Springer, 2011.

[23] R. Martinelli, M. Poggi, and A. Subramanian. Improved bounds for large scale capacitated arc routing problem. Computers & Operations Research, 40(8):2145 – 2160, 2013.

[24] Y. Mei, K. Tang, and X. Yao. A global repair operator for capacitated arc routing problem. IEEE Transactions on Systems, Man, and Cybernetics, Part B, 39(3):723–734, 2009.

[25] F. Neri, C. Cotta, and P. Moscato. Handbook of memetic algorithms, volume 379 of Studies in Computational Intelligence. Springer, 2012.

[26] M. Polacek, K. F. Doerner, R. F. Hartl, and V. Maniezzo. A variable neighborhood search for the capacitated arc routing problem with intermediate facilities. Journal of Heuristics, 14(5):405–423, 2008.

[27] D. Porumbel. Ray projection for optimizing polytopes with prohibitively many constraints in set-covering column generation. *Mathematical Programming*, in press (2014).

[28] D. C. Porumbel, J.-K. Hao, and P. Kuntz. Spacing memetic algorithms. In the 13th Annual Conference on Genetic and Evolutionary Computation, GECCO 11 (GA track), pages 1061–1068, 2011.

[29] C. Prins, N. Labadi, and M. Reghioui. Tour splitting algorithms for vehicle routing problems. International Journal of Production Research, 47(2):507–535, 2009.

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

[31] É. Taillard, P. Badeau, M. Gendreau, F. Guertin, and J.-Y. Potvin. A tabu search heuristic for the vehicle routing problem with soft time windows. Transportation science, 31(2):170–186, 1997.

[32] K. Tang, Y. Mei, and X. Yao. Memetic algorithm with extended neighborhood search for capacitated arc routing problems. IEEE Transactions on Evolutionary Computation, 13(5):1151–1166, 2009.

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

[34] S. Wøhlk. A decade of capacitated arc routing. In B. Golden, S. Raghavan, and E. Wasil, editors, The Vehicle Routing Problem: Latest Advances and New Challenges, volume 43 of Operations Research/Computer Science Interfaces, pages 29–48. Springer, 2008.

A Detailed Instance by Instance Results

The table below provides the instance-by-instance results that generated the summaries from Table 1 (Section 6.1). The column headings are rather self-explanatory. We point out that Column 4 actually reports the best bounds retrieved from `logistik.bwl.uni-mainz.de/benchmarks.php` at the beginning of this study, but we provide a footnote for the graphs for which we improved the best-known upper bound. Column 6 and 8 provide the time and the number of iterations at the moment when the best solution was found, while Column 7 indicates the number of decoder calls at the end of the search.

Instance	m	k	LB_{bst}–UB_{bst} (or optimum)	UB_{ILS}	Time [s]	Total Decoders (simplified ver.)	ILS iters	LB_{CG}	Time: CG Improver+CG Proc	CG iters
`kshs1`	15	4	14661	14661	0.28	98 (74)	4	13553	0+0.03	32
`kshs2`	15	4	9863	9863	0.04	336 (280)	1	8693	0+0.02	30
`kshs3`	15	4	9320	9320	0.14	23 (12)	5	8538	0+0.02	35
`kshs4`	15	4	11498	11498	0.92	1917 (1014)	47	11498	0.14+0.04	33
`kshs5`	15	3	10957	10957	1.15	3949 (1976)	28	10370	0+0.06	54
`kshs6`	15	3	10197	10197	0.24	70 (51)	4	9213	0+0.04	44
`gdb1`	22	5	316	316	0.57	386 (298)	4	284	0+0.03	45
`gdb2`	26	6	339	339	0.56	232 (214)	6	313	0+0.05	45
`gdb3`	22	5	275	275	0.18	21 (16)	1	250	0+0.02	45
`gdb4`	19	4	287	287	0.23	48 (27)	3	270	0+0.02	27
`gdb5`	26	6	377	377	3.13	5855 (4710)	11	359	0+0.02	37
`gdb6`	22	5	298	298	0.41	162 (145)	4	283	0+0.02	41
`gdb7`	22	5	325	325	0.70	530 (399)	2	291	0+0.03	37
`gdb8`	46	10	348	348	125.28	244764 (110071)	198	330	1.21+0.09	114
`gdb9`	51	10	303	303	83.93	173203 (55726)	134	294	0.52+0.16	116
`gdb10`	25	4	275	275	0.72	317 (197)	4	254	0+0.05	60
`gdb11`	45	5	395	395	3.35	2235 (1297)	9	364	0+0.19	116
`gdb12`	23	7	458	458	5.89	29472 (17507)	65	445	0.09+0.02	14
`gdb13`	28	6	536	536	2.05	4223 (1479)	38	526	0+0.08	69
`gdb14`	21	5	100	100	0.31	60 (54)	2	98	0+0.03	43
`gdb15`	21	4	58	58	0.02	2605 (2464)	1	57	0+0.02	39
`gdb16`	28	5	127	127	0.78	1028 (337)	9	122	0+0.06	70
`gdb17`	28	5	91	91	0.22	18 (18)	1	85	0+0.05	46
`gdb18`	36	5	164	164	0.47	54 (51)	2	159	0+0.11	89
`gdb19`	11	3	55	55	0.19	106 (88)	2	54	0+0.02	19
`gdb20`	22	4	121	121	7.85	58511 (4036)	67	114	0.07+0.04	60
`gdb21`	33	6	156	156	0.63	242 (111)	9	152	0+0.07	72
`gdb22`	44	8	200	200	8.52	17063 (6139)	12	197	0+0.11	85
`gdb23`	55	10	233	233	63.50	190357 (1551)	44	233	0.41+0.17	109
`val1A`	39	2	173	173	0.77	153 (122)	3	146	0+0.12	81
`val1B`	39	3	173	173	8.53	16438 (5132)	54	149	0.17+0.32	149
`val1C`	39	8	245	253	44.02	1011207 (379179)	1543	235	14.02+0.15	121
`val2A`	34	2	227	227	0.52	125 (118)	2	200	0+0.36	183
`val2B`	34	3	259	259	0.30	18 (14)	2	231	0+0.31	213
`val2C`	34	8	457	457	7.09	28572 (9922)	120	457	0.23+0.067	94
`val3A`	35	2	81	81	0.50	56 (54)	2	69	0+0.18	124

Instance	m	k	LB_{bst}–UB_{bst} (or optimum)	UB_{ILS}	Time [s]	Total Decoders (simplified ver.)	ILS iters	LB_{CG}	Time: CG Improver+CG Proc	CG iters
val3B	35	3	87	87	1.11	536 (400)	5	77	0+0.29	248
val3C	35	7	138	138	3.92	10963 (4350)	30	131	0+0.067	90
val4A	69	3	400	400	2.44	384 (312)	6	357	0+1.9	278
val4B	69	4	412	412	2.12	319 (229)	5	369	0+1.4	305
val4C	69	5	428	428	18.44	8225 (5254)	21	393	0+1.2	296
val4D	69	9	530	536	135.68	330648 (221911)	58	497	3.07+0.65	269
val5A	65	3	423	423	14.24	3640 (2922)	14	382	0+1.3	218
val5B	65	4	446	446	1.64	302 (246)	4	404	0+1.1	265
val5C	65	5	474	474	57.58	21822 (15797)	32	434	0+0.73	198
val5D	65	9	577	585	241.11	313451 (221259)	96	544	2.48+0.62	231
val6A	50	3	223	223	1.47	387 (313)	5	194	0+0.63	216
val6B	50	4	233	233	40.03	27017 (20766)	63	203	0.53+0.57	254
val6C	50	10	317	317	2.88	2179 (1411)	12	298	0+0.13	131
val7A	66	3	279	279	1.50	225 (153)	7	249	0+0.55	193
val7B	66	4	283	283	3.45	978 (607)	12	252	0+0.74	232
val7C	66	9	334	334	12.80	7393 (3903)	23	301	0+0.34	222
val8A	63	3	386	386	12.02	3263 (2605)	8	353	0+0.92	176
val8B	63	4	395	395	31.01	11468 (8497)	13	365	0+0.74	192
val8C	63	9	521	527	325.00	687417 (214864)	410	501	6.30+0.37	196
val9A	92	3	323	323	76.63	64702 (50357)	18	280	2.03+3.4	297
val9B	92	4	326	326	331.54	47798 (36402)	43	286	1.52+2.5	301
val9C	92	5	332	332	313.75	56237 (40638)	44	292	0+2.1	325
val9D	92	10	388	391	845.07	497204 (338620)	146	357	5.64+1.5	423
val10A	97	3	428	428	277.08	25878 (21844)	45	382	4.75+3.1	259
val10B	97	4	436	436	631.19	74481 (61059)	103	389	5.69+2.8	282
val10C	97	5	446	446	257.66	37969 (29535)	65	400	1.89+2.1	271
val10D	97	10	525	532	455.65	424943 (300107)	51	485	5.19+2.3	487
C01	79	9	4150	4160	80.19	566459 (393986)	44	3870	9.43+1.6	490
C02	53	7	3135	3135	7.28	8975 (3932)	32	2982	0+0.48	244
C03	51	6	2575	2585	30.02	285221 (212109)	44	2428	4.69+0.88	333
C04	72	8	3510	3510	14.44	5936 (4262)	16	3286	0+1.3	496
C05	65	10	5365	5370	418.29	895803 (299769)	764	5118	11.62+0.85	356
C06	51	6	2535	2535	44.90	38824 (29379)	55	2354	0.45+0.61	318
C07	52	8	4075	4075	1.31	665 (452)	18	3810	0+0.42	263
C08	63	8	4090	4090	74.77	107351 (34289)	210	3865	1.90+0.55	256
C09	97	12	5245-5260	5300	1669.94	1159289 (527514)	499	5003	27.75+3.1	678
C10	55	9	4700	4730	99.64	453234 (317378)	95	4367	6.21+0.43	251
C11	94	10	4615-4630	4645	901.39	598765 (406690)	217	4397	9.05+4.3	688
C12	72	9	4240	4240	181.19	193031 (68447)	257	4006	3.87+1.2	419
C13	52	7	2955	2955	58.88	47729 (38788)	65	2774	0+0.72	329
C14	57	8	4030	4030	99.85	175211 (2491)	349	3885	6.94+0.72	321
C15	107	11	4920-4940	4985	684.59	520927 (391992)	94	4741	7.94+10	1220
C16	32	3	1475	1475	2.48	4210 (2644)	23	1394	0+0.68	232
C17	42	7	3555	3555	33.06	58071 (36026)	122	3314	0.48+0.23	174
C18	121	11	5580-5620	5645	1818.30	709067 (493603)	875	5368	54.91+35	1495
C19	61	6	3115	3120	4.38	260670 (154614)	13	2933	5.74+1.2	413
C20	53	5	2120	2120	5.89	5603 (2742)	26	2019	0+1.1	417
C21	76	8	3970	3970	15.93	5915 (4082)	29	3774	0+2.7	637
C22	43	4	2245	2245	24.04	37009 (20642)	125	2164	0.55+0.74	322
C23	92	8	4075-4085	4130	683.31	1203496 (1932)	932	3761	57.53+8.4	937
C24	84	7	3400	3400	369.05	166937 (94213)	202	3283	3.26+4.1	853
C25	38	5	2310	2310	8.55	13676 (8621)	41	2221	0+0.29	208
D01	79	5	3215	3235	8.42	186175 (173920)	14	3000	21.85+5.6	699
D02	53	4	2520	2520	2.36	926 (778)	7	2358	0+3.6	560
D03	51	3	2065	2065	4.72	2234 (1944)	11	1913	0+5.4	638
D04	72	4	2785	2785	4.93	981 (797)	8	2505	0+5.1	806
D05	65	5	3935	3935	7.60	3650 (2276)	16	3614	0+2.1	479
D06	51	3	2125	2125	33.82	20958 (16802)	53	1899	5.95+1.7	539
D07	52	4	3115	3165	6.24	161667 (105252)	20	2873	9.69+2.2	378
D08	63	4	3045	3045	2.78	1075 (704)	14	2782	0+2.7	529
D09	97	6	4120	4120	134.23	33329 (23564)	48	3834	0+9.9	1116
D10	55	5	3340	3340	1.84	567 (554)	10	3163	0+2.2	442
D11	94	5	3745	3760	36.12	175035 (152308)	21	3374	10.87+37	1233

Instance	m	k	LB_{bst}–UB_{bst} (or optimum)	UB_{ILS}	Time [s]	Total Decoders (simplified ver.)	ILS iters	LB_{CG}	Time: CG Improver+CG Proc	CG iters
D12	72	5	3310	3310	10.40	3200 (2583)	13	2961	0+5.4	720
D13	52	4	2535	2535	125.25	77249 (74724)	209	2226	5.17+2	447
D14	57	4	3280	3280	3.87	2088 (1517)	15	2971	0+4.7	663
D15	107	6	3990	4000	53.18	177834 (159872)	21	3778	14.45+22	1598
D16	32	2	1060	1060	0.59	86 (82)	5	1090	0+3.8	253
D17	42	4	2620	2620	0.72	113 (101)	3	2375	0+0.44	217
D18	121	6	4165	4165	116.32	13134 (11829)	43	3855	0+1.62	2880
D19	61	3	2400	2400	0.89	91 (78)	4	2248	0+13	981
D20	53	3	1870	1870	11.90	4472 (4101)	21	1778	0+1.6	452
D21	76	4	3005-3050	3055	58.14	173296 (145604)	46	2791	22.93+37	1317
D22	43	2	1865	1865	2.32	1415 (1073)	14	1730	0+5.5	517
D23	92	4	3130	3135	271.50	319189 (2247)	250	2893	167.76+122	1667
D24	84	4	2710	2710	10.91	1779 (1527)	11	2495	0+42	1443
D25	38	3	1815	1815	0.90	264 (248)	4	1670	0+1.2	301
E01	85	10	4900-4910	4940	1185.24	1902721 (1988)	2274	4719	36.35+2.7	617
E02	58	8	3990	3990	168.28	318500 (31512)	963	3866	3.64+0.63	308
E03	47	5	2015	2015	2.73	1835 (1563)	9	1854	0+0.95	414
E04	77	9	4155	4220	147.24	754484 (442993)	97	3983	13.65+2.6	589
E05	61	9	4585	4675	129.03	362630 (270372)	84	4374	2.96+0.92	351
E06	43	5	2055	2055	1.16	434 (325)	6	2008	0+0.48	258
E07	50	8	4155	4155	1.81	1157 (746)	8	3980	0+0.6	311
E08	59	9	4710	4710	19.68	22727 (9743)	41	4461	0+0.81	306
E09	103	12	5805-5820	5910	1358.29	1787277 (750223)	1989	5563	75.72+5.4	841
E10	49	7	3605	3605	54.94	53152 (37059)	92	3488	0.83+0.57	288
E11	94	10	4650	4745	1242.26	570177 (396834)	249	4381	8.14+4	758
E12	67	9	4180	4245	291.19	375353 (234886)	206	4019	6.51+1.2	435
E13	52	7	3345	3345	12.09	13763 (7109)	58	3201	0+0.61	348
E14	55	8	4115	4135	51.95	692596 (325737)	144	3902	12.67+0.8	316
E15	107	9	4205	4245	96.02	470829 (311398)	35	3904	9.17+10	1113
E16	54	7	3775	3775	174.71	182444 (108856)	266	3648	2.90+0.77	417
E17	36	5	2740	2740	7.60	17789 (7689)	57	2556	0.31+0.22	173
E18	88	8	3835	3835	304.56	86326 (62437)	50	3602	0+4.7	898
E19	66	6	3235	3240	250.03	511012 (3078)	1193	3053	35.79+3.7	636
E20	63	7	2825	2825	150.60	79451 (62631)	103	2656	0.84+1.3	436
E21	72	7	3730	3735	751.43	728889 (325557)	739	3561	15.12+3.1	754
E22	44	5	2470	2470	12.52	11523 (8934)	31	2343	0+0.92	350
E23	89	8	3710	3730	878.00	446519 (291832)	249	3406	8.01+4.4	824
E24	86	8	4020	4040	361.99	420112 (319133)	129	3743	10.37+4.9	915
E25	28	4	1615	1615	1.17	702 (549)	5	1558	0+0.21	161
F01	85	5	4040	4045	321.02	464620 (5221)	298	3565	52.99+23	1478
F02	58	4	3300	3300	70.05	76695 (4277)	198	3115	5.23+3.2	541
F03	47	3	1665	1665	1.85	666 (653)	7	1532	0+8.2	623
F04	77	5	3485	3520	5.97	205626 (184284)	12	3109	4.76+12	751
F05	61	5	3605	3605	5.61	2367 (1990)	10	3295	0+4.3	835
F06	43	3	1875	1875	0.66	75 (73)	5	1790	0+2.1	459
F07	50	4	3335	3335	2.03	888 (615)	14	3121	0+1.6	421
F08	59	5	3705	3705	8.27	3575 (2941)	9	3311	0+2.4	557
F09	103	6	4730	4770	192.04	443480 (282953)	316	4310	169.79+43	1502
F10	49	4	2925	2925	1.91	656 (585)	4	2708	0+2.9	550
F11	94	5	3835	3865	19.80	176054 (144014)	14	3400	13.98+29	1403
F12	67	5	3395	3395	372.12	136637 (117032)	214	3097	5.31+5.2	642
F13	52	4	2855	2855	1.40	437 (371)	12	2584	0+3.9	766
F14	55	4	3330	3350	144.05	169874 (117560)	252	3021	8.73+3.6	577
F15	107	5	3560	3560	688.87	79514 (70945)	105	3202	2.09+43	1825
F16	54	4	2725	2725	1.41	478 (425)	4	2633	0+27	767
F17	36	3	2055	2055	0.91	279 (251)	8	1935	0+3.1	363
F18	88	4	3065-3075	3075	35.29	133803 (106839)	23	2848	43.52+36	1942
F19	66	3	2515-2525	2525	4.05	151846 (5245)	20	2317	32.63+54	1353
F20	63	4	2445	2450	3.28	87218 (81782)	14	2230	2.93+15	917
F21	72	4	2930	2930	6.21	1412 (1219)	17	2696	0+28	1318
F22	44	3	2075	2075	15.83	10647 (10433)	34	1851	0.40+13	726
F23	89	4	3005	3010	82.91	134450 (102764)	29	2754	6.71+18	1041
F24	86	4	3210	3215	652.45	150020 (131166)	255	2924	20.53+27	1464

Instance	m	k	$\mathrm{LB_{bst}}$–$\mathrm{UB_{bst}}$ (or optimum)	$\mathrm{UB_{ILS}}$	Time [s]	Total Decoders (simplified ver.)	ILS iters	$\mathrm{LB_{CG}}$	Time: CG Improver+CG Proc	CG iters
`F25`	28	2	1390	1390	0.28	17 (15)	2	1356	0+0.29	168
`egl-e1-A`	51	5	3548	3548	1.51	779 (502)	23	3395	0+20	498
`egl-e1-B`	51	7	4498	4516	309.71	553520 (255829)	878	4252	26.43+4.1	370
`egl-e1-C`	51	10	5595	5613	460.72	755568 (435483)	771	5412	38.68+2.1	277
`egl-e2-A`	72	7	5018	5018	40.47	24918 (14193)	43	4714	0+59	817
`egl-e2-B`	72	10	6317	6343	1320.87	1044208 (590194)	835	6017	48.20+14	635
`egl-e2-C`	72	14	8335	8335	98.75	157114 (48303)	178	8112	2.22+3.3	450
`egl-e3-A`	87	8	5898	5898	49.48	33026 (14741)	53	5614	0+48	1030
`egl-e3-B`	87	12	7744-7775	7787	1791.06	1512201 (643071)	941	7467	53.18+8.5	650
`egl-e3-C`	87	17	10244-10292	10309	220.10	3163038 (1186823)	154	9954	533.32+4.1	501
`egl-e4-A`	98	9	6408-6444	6464	649.87	840474 (371448)	357	6125	55.42+67	1082
`egl-e4-B`	98	14	8935-8961	9042	2158.97	2555719 (738112)	1443	8553	108.83+9.5	692
`egl-e4-C`	98	19	11512-11550	11626	347.69	2943840 (1285837)	614	11137	1424+3.3	469
`egl-s1-A`	75	7	5018	5018	257.24	111127 (68084)	150	4832	8.00+36	687
`egl-s1-B`	75	10	6388	6388	294.64	133135 (87867)	116	6109	9.53+9	475
`egl-s1-C`	75	14	8518	8518	61.02	78751 (21612)	110	8267	6.14+2.5	317
`egl-s2-A`	147	14	9825-9884	9890	39764.5	22756387 (14651276)	11443	9517	271+48	1620
`egl-s2-B`	147	20	13017-13100	13212	4368.76	14562843 (6234602)	3724	12718	836+15	1104
`egl-s2-C`	147	27	16425	16445	20089.1	27865308 (7674563)	8550	16183	8642+6.7	872
`egl-s3-A`	159	15	10165-10220	10341	5455.05	3032408 (2147235)	512	9751	100.44+60	1700
`egl-s3-B`	159	22	13648-13682	13738	7210.94	10176509 (4612659)	1042	13297	711.34+12	1075
`egl-s3-C`	159	29	17188	17209	16293.4	24677478 (7751558)	3453	16931	6335+6.5	815
`egl-s4-A`	190	19	12153-12268	12317	12290.2	4631560 (2689085)	835	11787	91.53+71	1526
`egl-s4-B`	190	27	16113-16321	16321	6810.30	13898323 (5278254)	933	15671	3451+13	1212
`egl-s4-C`	190	35	20430-20481	20590	705.071	19497691 (13457194)	657	20090	9114+7	1000
`egl-g1-A`	347	20	976907-1004864	1013682	202868	8031791 (4819108)	30348	916216	39684+803865	8554
`egl-g1-B`	347	25	1093884-1129937	1135065	138417	10422233 (4896198)	40218	1041850	9603+277327	6991
`egl-g1-C`[a]	347	30	1212151-1262888	1265692	80908	14761301 (5716637)	34499	1169088	8083+102932	5854
`egl-g1-D`	347	35	1341918-1398958	1400828	144100	20394529 (5714699)	57792	1306688	9815+70382	5315
`egl-g1-E`[a]	347	40	1482176-1543804	1542822	104758	38168743 (7573006)	169525	1451259	4953+36688	4821
`egl-g2-A`[a]	375	22	1069536-1115339	1120467	166418	9685568 (3724590)	38565	1008530	100403+402740	8055
`egl-g2-B`	375	27	1185221-1226645	1236449	69470	11214854 (5317883)	9802	1128118	30263+257932	6773
`egl-g2-C`[a]	375	32	1311339-1371004	1365825	56810	21190804 (7467003)	20692	1261382	6910+73086	5652
`egl-g2-D`	375	37	1446680-1509990	1518633	169060	45321225 (11640880)	96080	1400291	10506+22865	5463
`egl-g2-E`[a]	375	42	1581459-1659217	1657134	453570	55355616 (12058694)	325854	1543043	11762+23447	4682

[a] For these instances, the values $\mathrm{UB_{bst}}$ in Column 4 represent the best upper bounds known at the *beginning* of this study. Unless indicated otherwise, all gaps reported in Section 6 are calculated with regard to these bounds. However, `CG-P-ILS` has found the following new upper bounds: 1260433, 1542822, 1115207, 1365198, and 1657134 for `egl-g1-C`, `egl-g1-E`, `egl-g2-A`, `egl-g2-C`, and (resp.) `egl-g2-E`.

Finally, one can use this table to evaluate the faster simplified decoder. Recall (from Remark 1, Section 5.2.3) that that this simplified decoder ignores the fleet size constraint, and so, it uses k times less asymptotic running time. Column 7 provides both the total number of decoder calls and the number of simplified decoder calls (in parentheses). If the simplified decoder reports an explicit solution with more than k vehicles, the full decoder still needs to be run, *i.e.*, `CG-P-ILS` ends up calling both decoders. To see how many times this latter (unfortunate) situation arises, it is enough to compute the difference between the number of decoder calls and the number of simplified decoder calls. By counting all rows, this simplified decoder is sufficient for roughly half of the decoded permutations.

B CARP-Focused Neighbors and Stagnation Parameters in ILS

B.1 Identifying a Promising Sub-Neighborhood

We described in Section 3.3.2 a neighborhood structure of size $O(m^3)$ and we argued that it might be necessary to prune certain neighbors and focus on a smaller CARP-specialized neighborhood. We propose the following pruning and reduction techniques:

1. First, we observe that for the case $m > 100$, a neighborhood size of $O(m^3)$ is certainly prohibitively large. To avoid this, we simply reduce the set of sequence neighbors by fixing $\ell = m$, *i.e.*, we generate *sequence* neighbors only by moving sequences $(s_i, s_{i+1}, \dots s_j)$ to the end. This leads to a $O(m^2)$ neighborhood version. The next reductions below apply for both the $O(m^2)$ and the $O(m^3)$ neighborhoods.
2. The main pruning technique relies on *pre-evaluating* all neighbors using a heuristic cost function that can be calculated more rapidly. Before performing any decoding, the ILS process first computes for each neighbor $s \in N_C(s)$ the following *non-service linking cost* $\texttt{lnk}(s')$

$$\texttt{lnk}(s') = \overline{d}(v_{\text{dep}}, s'_1) + \overline{d}(s'_1, s'_2) + \overline{d}(s'_2, s'_3) + \dots \overline{d}(s'_{m-1}, s'_m) + \overline{d}(s'_m, v_{\text{dep}})$$

   where $\overline{d}(i, j)$ is basically an underestimator of the distance from either end of i to either end of j *or* to v_{dep} (see below). The *non-service linking cost* $\texttt{lnk}(s')$ underestimates the cost of leaving the depot, consecutively linking the edges in the order $s'_1, s'_2, \dots, s'_m$ (without servicing them) and returning to depot. After this pre-evaluation, we restrict $N_C(s)$ to set $N(s) \subseteq N_C(s)$ with the lowest *nsize* values of the above linking cost, where *nsize* is a parameter that allows one to change the neighborhood to any desired size.[5]
3. Another technique relies on pruning potential neighbors with very far consecutive edges. We prune any neighbor s' for which there is some $i \in [1..m-1]$ such that $\overline{d}(s'_i, s'_{i+1}) > 2 \cdot \frac{\texttt{lnk}(s_{\text{best}})}{m+1}$, where $\texttt{lnk}(s_{\text{best}})$ is the non-service linking cost of the best visited solution s_{best}. We mention a special case: if the average linking cost $\frac{\texttt{lnk}(s_{\text{best}})}{m+1}$ of s_{best} is very low (less than 1), this reduction would be very imprecise and it is skipped.
4. The last reduction technique simply prunes all neighbors that swap two different routes (sequences) of the current solution. Such neighbors would only artificially increase the neutrality of the search space.

We still need to formally define the above edge-to-edge distance function $\overline{d}$. Given any $e_1, e_2 \in [1..m]$, $\overline{d}(e_1, e_2)$ should quantify how far edge e_1 is from edge e_2. We observed that typical optimal solutions s^* verify the following property: any consecutive edges s^*_i and s^*_{i+1} are either relatively close to each other or they belong to different routes. Based on this, $\overline{d}(e_1, e_2)$ is the minimum of: (i) the shortest path between either ends of e_1 and e_2 and (ii) the shortest path between any end of either edge and the depot. We observe that point (ii) is motivated by the possibility of finishing a route with edge e_1 and starting a new one with edge e_2. For instance, if e_1 is close to the depot, then we consider that e_1 could be easily followed by any distant edge e_2, because e_1 and e_2 can easily belong to different routes.

B.2 Parameters for Detecting Stagnation and Recurrent Looping

As hinted in Section 2.1, the perturbation is automatically triggered after a number of *maxNeutralIt* iterations with no variation of the objective value. This represents the condition for detecting a stagnation, *i.e.*, a situation in which the search process is stuck looping on a plateau. The value of *maxNeutralIt* depends on the cost $f(s)$ of the current solution. As long as the current $f(s)$ does not deviate by more than 3% from the cost $f(s_{\text{best}})$ of the best visited solution s_{best}, we are on a "base" case. We set *maxNeutralIt* at a "base" value of $maxNeutralIt = 10 + 20 \cdot \lfloor \frac{m}{200} \rfloor$. The last term comes from the fact that the largest instances can have larger plateaux. As soon as $f(s) > 103\% f(s_{\text{best}})$, we consider the search process explores a lower quality area and *maxNeutralIt* is divided by 2. More exactly, in this case, we update $maxNeutralIt \leftarrow \max\left(\frac{maxNeutralIt}{2} - f_{\text{dev}}(s)/3, 1\right)$, where $f_{\text{dev}}(s) = \left\lceil \frac{f(s) - f(s_{\text{best}})}{f(s_{\text{best}})} \cdot 100 \right\rceil$. We observe that this formula allows the perturbation to be triggered immediately if $f(s) \gg f(s_{\text{best}})$. Furthermore, `CG-P-ILS` can apply stronger perturbations on lower quality solutions, by injecting in such solutions more than one route from $\mathcal{P}$: the exact number of sequences is set to $1 + \lceil f_{\text{dev}}(s)/5 \rceil$.

[5]After preliminary experiments, we we set $nsize = m \cdot k$ (corresponding to a neighborhood linear in the number of edges and the number of vehicles) for $m < 100$, or $msize = 0.001|N_C(s)|$ otherwise.

Regarding the recurrent looping state (recall Section 3.3.3.1), we detect it by the following approach. We consider a reference iteration `itrf` and, as long as the solution $s^{\texttt{it}}$ at current iteration `it` verifies $f(s^{\texttt{it}}) \in [0.9f(s^{\texttt{itrf}}), 1.1f(s^{\texttt{itrf}})]$, we keep `itrf` as reference iteration. As soon as this condition is not verified at some iteration `it`, we update $\texttt{itrf} \leftarrow \texttt{it}$. The recurrent looping state is declared at any iteration `it` such that $\texttt{it} - \texttt{itrf} > 2 \cdot maxNeutralIt$.

Recall that the detection of such looping state triggers the CG improver. Since the CG improver is a blocking operator in ILS, the above condition was designed so as to avoid calling the CG improver too frequently (this would slow down the search). However, to be certain that such problems do not arise, we limit the total time spent on the CG improver to 100000s; this limit was reached only on the instance `egl-g2-A`. It is not difficult to make the algorithm adapt the above parameters so as to keep a good balance between the time spent on the ILS and the time spent on the CG improver.

C Exact and Heuristic Cycle Avoidance in the CG Pricing

Cycling is a recurrent issue in CG pricing algorithms based on Dynamic Programming (DP). The problem comes from the fact that a DP state S can be reached via a transition $S' \xrightarrow{e} S$ by servicing an edge e already serviced during the transitions that led to state S'. A simplest cycle example consists of a sequence of transitions such as $S \xrightarrow{e} S' \xrightarrow{e} S$. Using the notations from Section 4.1, observe that a state $S = (v_b, q+q_e)$ can indeed be constructed by servicing edge $e = (v_a, v_b)$ from state $S' = (q, v_a)$ without checking if edge e was not already used to construct state (q, v_a).

Let us interpret the states as vertices of a directed graph whose arcs encode precedence relations between states, *i.e.*, an arc $S' \xrightarrow{e} S$ indicates that state S can be reached from S' by servicing e. As such, if Step 3.(a) (recall the Algorithm from Section 4.1) finds a new transition to an existing state $S = (v_b, q+q_e)$, *e.g.*, by finding $f(v_a, q) + d_e - y_e = g(v_b, q + q_e)$, there can be more than one arc that leads to S. To service e from state S, one should ensure there is at least one path towards S that does not use e.

We first present a simpler *inexact technique* for avoiding cycles with length up to 20. It is only used by the CG improver that does not need to converge, but that should avoid sending routes with cycles to the ILS process. The main idea is to construct a simplified version of the above graph by keeping only one arc entering each state S, *i.e.*, the discovery of a second arc leading to S is ignored. As such, this simplified graph has a unique path towards each state S. We then simply record the last 20 edges serviced by this path to S and we forbid servicing on S any of these 20 edges, *i.e.*, these edges are considered forbidden in S. The implementation of such routine involves a constant factor per iteration, and so, the theoretical complexity of the pricing algorithm does not increase.

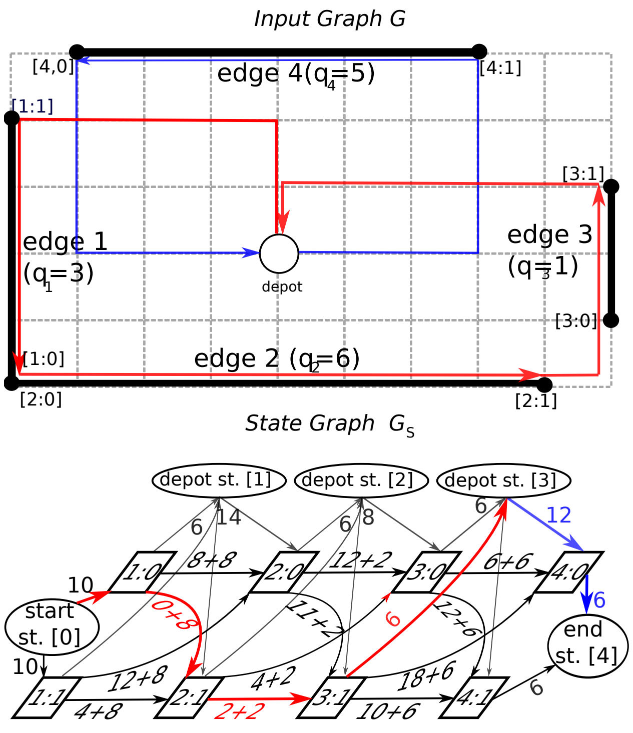
We now present the *exact 2-cycle reduction* used by the main CG process. To guarantee the correctness of the lower bound returned at the end of the CG convergence, the pricing algorithm needs to remain exact. For this, we need to consider that an edge e is forbidden (not to be serviced) in S only if it arises on every path of length 2 leading to S. For instance, if the only path towards S is $S_2 \xrightarrow{e_2} S_1 \xrightarrow{e_1} S$, then e_1 and e_2 are forbidden in S; if there exists an alternative path $S_2' \xrightarrow{e_2'} S_1 \xrightarrow{e_1} S$, then only e_1 is forbidden in S.

This approach has a computational and memory cost: it might produce more sub-states for the same $v \in V$ and $q \in [1..Q]$. Indeed, if two sub-states S' and S for the same (v, q) have different forbidden edges, one should consider recording both of them (even if $f(S') > f(S)$). By this duplication process, we obtain a larger graph but with a *unique entering arc per sub-state*. Under these conditions, we can then simply associate to each sub-state S a list of edges that are forbidden (can not be serviced) in S: they are exactly the edges of the unique path of length 2 leading to S.

However, the number of sub-states that have to be created for the same (v, q) is at maximum 3. We show this by observing the following. If a new sub-state S_{new} has a lower quality than an existing state S for the same (v, q), then S_{new} can be useful *only if* there exists some edge forbidden in S and *not* forbidden in S_{new}. More generally, a lower quality state S_{new} is only useful if it reduces the size of the set of edges forbidden in all sub-states already created on (v, q). This size reduction can happen at maximum 3 times, because the set of edges forbidden in (v, q) can have at most of 2 elements, *i.e.*, at most the edges of the unique 2-length path leading to S. After at most three size reductions for the same (v,q), the set of forbidden edges in (v,q) becomes empty. In such case, there is always a sub-state in (v,q) that allows Step 3.(a) to service any new edge.

D Example of a State Graph and Decoder

For the sake of clarity, let us exemplify the construction of a state graph G_S used in Section 5.2.1. The graph G in the right figure has four required edges of total demand $q_1 + q_2 + q_3 + q_4 = 15$ and $Q = 10$. For any $u, v \in V$, the length of edge $\{u, v\}$ is given by the Manhattan distance, *i.e.*, $|x_u - x_v| + |y_u - y_v|$, where (x_u, y_u) and (x_v, y_v) are the coordinates of u and v. Observe the depot is at distance 6 from any other vertex. The state graph G_S is provided below: the *open* states are depicted as squares in perspective and the *depot* states in ovals. Let us exemplify some arcs of G_S:



1. the very first route needs to start with edge 1: traverse a distance of 6 to reach either end of edge 1 and a distance of $d_1 = 4$ to service it, ending up either in $[1:0]$ or in $[1:1]$, hence two arcs of weight 10 exiting vertex ``‘‘start st. [0]’’``;
2. The arc $[1:0] \to [2:0]$ represents a vehicle move from $[1:0]$ to $[2:1]$, followed by a service of edge 2, thus a cost of $s_{\text{path}}([1:0],[2:1]) + d_2 = 8+8$;
3. The red path from "`start st. [0]`" to "`depot st.` [3]" represents the shortest route that starts at the depot, services $\{1, 2, 3\}$ and returns to the depot (total cost $10 + 8 + 4 + 6 = 28$).

The red and the blues paths in G_S represent arcs between *depot* states. For instance, we can say there is a (red) arc between [0] and [3] of weight 28 and a (blue) arc between [3] and [4] of weight 18. Both these arcs encode complete routes: the first one services edges $\{1, 2, 3\}$ and the second one services $\{4\}$. Based on them, one can compute the shortest path from [0] and [4], and find an optimal solution of cost $28 + 18 = 46$ that services the required edges in the order $1, 2, 3, 4$.