ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

A Fast and Scalable Heuristic for the Solution of Large-Scale Capacitated Vehicle Routing Problems

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:
Accorsi, L., Vigo, D. (2021). A Fast and Scalable Heuristic for the Solution of Large-Scale Capacitated Vehicle Routing Problems. TRANSPORTATION SCIENCE, 55(4), 832-856 [10.1287/trsc.2021.1059].

Availability:
This version is available at: https://hdl.handle.net/11585/846180 since: 2022-01-18

Published:
DOI: http://doi.org/10.1287/trsc.2021.1059

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

15 September 2026

Submitted to *Transportation Science*
manuscript (Please, provide the manuscript number!)

Authors are encouraged to submit new papers to INFORMS journals by means of a style file template, which includes the journal title. However, use of a template does not certify that the paper has been accepted for publication in the named journal. INFORMS journal templates are for the exclusive purpose of submitting to an INFORMS journal and should not be used to distribute the papers in print or online or to submit the papers to another publication.

A Fast and Scalable Heuristic for the Solution of Large-Scale Capacitated Vehicle Routing Problems

Luca Accorsi[1], Daniele Vigo[1,2]
[1]Department of Electrical, Electronic and Information Engineering "G. Marconi", University of Bologna,
Viale del Risorgimento 2, 40136 - Bologna, Italy,
[2]CIRI ICT, University of Bologna, via Quinto Bucci, 336, 47521 - Cesena, Italy
{luca.accorsi4, daniele.vigo}@unibo.it

In this paper, we propose a fast and scalable, yet effective, metaheuristic called FILO to solve large-scale instances of the Capacitated Vehicle Routing Problem. Our approach consists of a main iterative part, based on the Iterated Local Search paradigm, which employs a carefully designed combination of existing acceleration techniques, as well as novel strategies to keep the optimization localized, controlled and tailored to the current instance and solution. A Simulated Annealing-based neighbor acceptance criterion is used to obtain a continuous diversification, to ensure the exploration of different regions of the search space. Results on extensively studied benchmark instances from the literature, supported by a thorough analysis of the algorithm's main components, show the effectiveness of the proposed design choices, making FILO highly competitive with existing state-of-the-art algorithms, both in terms of computing time and solution quality. Finally, guidelines for possible efficient implementations, algorithm source code and a library of reusable components are open-sourced to allow reproduction of our results and promote further investigations.

Key words: Capacitated Vehicle Routing Problem, Metaheuristics, Acceleration Techniques, Large-Scale Instances

1. Introduction

The Capacitated Vehicle Routing Problem (CVRP) has been studied for several decades but still remains a challenging problem to solve in practice. Recently, several new benchmark instances having large and very large scale of this fundamental problem (see $\mathbb{X}$ dataset, Uchoa et al. (2017) and $\mathbb{B}$ dataset, Arnold, Gendreau, and Sörensen (2019)) have brought it into focus again in the vehicle routing scene. The careful design and implementation of solution algorithms becomes of primary importance when dealing with large instances. Failing to find the best tradeoff between effectiveness and efficiency has dramatic effects which are much more noticeable than when dealing with smaller instances. While computer memory capacity is a less pressing problem every year,

finding a solution of satisfactory quality within a reasonable computing time still remains the real challenge: algorithm designers cannot rely on continuous increments in the working frequency of future processors. Processing units have, in fact, almost hit their maximum physical speed. New chips are moving towards massive parallelization (see Etiemble (2018)), possibly initiating an age of new algorithms that make use of concurrent decomposition techniques.

The CVRP can be described by using an undirected graph $G = (V, E)$ where V is the vertex set and E is the edge set. The vertex set V is partitioned into $V = \{0\} \cup V_c$ where 0 is the depot and $V_c = \{1, \dots, N\}$ is a set of N customers. A cost c_{ij} is associated with each edge $(i, j) \in E$. Moreover, we assume that the cost matrix $\boldsymbol{c}$ satisfies the triangle inequality. For a vertex $i \in V$ and a subset of vertices $V' \in V$, we identify with $\mathcal{N}_i^k(V')$ the set of the k nearest neighbor vertices $j \in V'$ of i with respect to the cost matrix $\boldsymbol{c}$. The set $\mathcal{N}_i^k(V')$ is shortened to $\mathcal{N}_i(V')$ for the case $k = |V'|$. Each customer $i \in V_c$ requires an integer quantity $q_i > 0$ from the depot, and $q_0 = 0$. An unlimited fleet of homogeneous vehicles with capacity Q is located at the depot available to serve the customers. Recalling that a Hamiltonian circuit is a closed cycle visiting a set of customers exactly once, a CVRP solution S is composed of a number $|S|$ of Hamiltonian circuits, called *routes*, starting from the depot, visiting a subset of customers and coming back to the depot. We identify with r_i the route of load q_{r_i} serving customer $i \in V_c$. A solution is feasible if all customers are visited exactly once and none of the vehicles exceed its capacity. The cost of a solution S is given by the sum of the cost of the edges defining the routes of S. Finally, the CVRP goal is to find the feasible solution with the minimum cost.

A rough analysis of the computing time, normalized to be comparable, of three of the current state-of-the-art CVRP algorithms having a termination criterion based on the number of iterations is shown in Figure 1. Algorithms using a time-based termination criterion are not included because they are not comparable. In fact, even if such an approach were preferable in practice, it may not be the best in comparison settings due to the high variability in the number of steps executed by the algorithm (even occurring within the same hardware configuration), which may harm the reproducibility of final solutions; see Johnson (1999). Moreover, as described in Chapter 4 of Toth and Vigo (2003), computing time is just one among many, seldom conflicting, dimensions characterizing heuristic solution approaches. The quality of the solutions is often another of the most obvious criteria used to assess algorithm quality. In addition, scalability with respect to the instance size is another very valuable quality, especially when moving to very large-scale instances. Figure 1 shows that these algorithms often exhibit a non-linear growth of the computing time that may, in their current state, undermine their applicability to large-scale instances within a reasonable computational effort. This growth is the main motivation of our research.

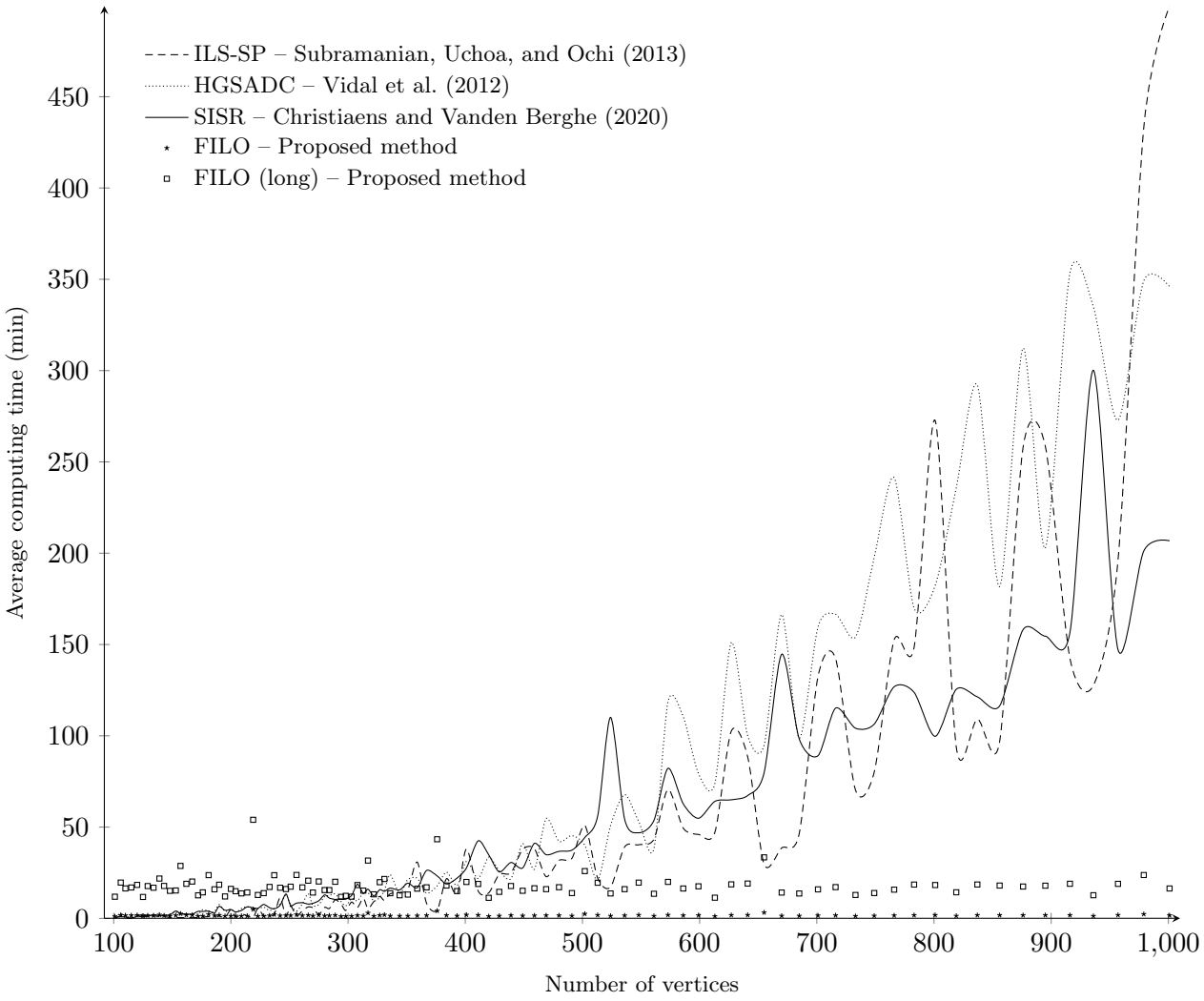


Figure 1 **Computing time growth trends for state-of-the-art CVRP algorithms proposed for the $\mathbb{X}$ dataset by Uchoa et al. (2017).**

The challenge we faced in this research was to design an effective and scalable heuristic approach to the CVRP that can solve, in reasonable computing times, very large-scale instances without an explicit instance decomposition. We reviewed and adapted existing local search acceleration techniques and introduced new strategies to keep the optimization localized, controlled, and tailored to the current instance and solution. The result is a well-defined and cohesive solution method.

In fact, local search acceleration techniques represent a very promising direction in the design of scalable algorithms that are efficient but still retain their effectiveness. The local search component, for a local search-based solution method, is typically one of the most time-consuming. Naive implementations, e.g., those built on full neighbor enumeration, fail to be competitive even for medium-sized instances. Among the most popular acceleration techniques, Granular Neighborhoods

(GNs), proposed by Toth and Vigo (2003), define a heuristic filtering of less promising neighbors. This approach has been experimentally shown to provide an excellent compromise between computing time and solution quality; see, e.g., Toth and Vigo (2003), Zachariadis and Kiranoudis (2010), Schneider, Schwahn, and Vigo (2017), Accorsi and Vigo (2020). The Sequential Search proposed by Irnich, Funke, and Grünert (2006) breaks a local search move into basic blocks called partial moves. The execution of those partial moves can be aborted if certain conditions are met, thus pruning in advance a non-promising local search move. Finally, Static Move Descriptors (SMD), introduced by Zachariadis and Kiranoudis (2010) and later improved by Beek et al. (2018), provide a data-oriented approach to the local search execution that avoids unnecessary evaluations by exploiting the locality of a local search move application.

Successful CVRP algorithms for large-scale instances typically make use of acceleration techniques and ad-hoc data structures to support their optimization process. Kytöjoki et al. (2007) have devised a Variable Neighborhood Search (VNS, see Mladenović and Hansen (1997)) algorithm combined with the Guided Local Search metaheuristic (GLS, see Voudouris and Tsang (1999)) to escape from local optima by accepting moves that worsen the solution value according to certain solution features. Their method is able to solve problems with up to twenty thousand customers in short computing times by using a number of implementation techniques to reduce memory utilization (e.g., storing compact representations for the cost matrix) and speeding up the computation with appropriate data structures and smart pre-processing. Zachariadis and Kiranoudis (2010) propose a Tabu Search metaheuristic (TS, see Glover (1989)) based on the SMD concepts in which a penalization strategy is used to diversify the search process. The method is able to solve problems with up to three thousand customers by exploiting the acceleration role of the SMD and a neighborhood reduction policy similar to the GN concept. Finally, Arnold, Gendreau, and Sörensen (2019) propose an adaptation for very large instances, having up to thirty thousand customers, of the algorithm introduced in Arnold and Sörensen (2019), consisting of a local search-based approach using a GLS metaheuristic enhanced by knowledge extracted from previous data mining analyses. The authors reduce the computing time and space requirements by limiting the amount of information stored and using pruning and sequential search techniques.

The algorithm described in this paper, called FILO, consists of a main iterative part based on the Iterated Local Search paradigm, coupled with a Simulated Annealing-based neighbor acceptance criterion to obtain a continuous diversification aimed at exploring diversified regions of the search space. Our approach makes use of GNs and SMDs to speed up the local search executions, together with other newly introduced concepts to keep the optimization localized, controlled, and tailored to the current instance and solution. The main design contributions, embedded into the proposed solution algorithm, are the following:

- We extend the move generator concept introduced in Schneider, Schwahn, and Vigo (2017) to support a dynamic, fine-grained management to intensify the local search only in areas that are more likely to require a more accurate optimization process, such as parts of the solution that were not improved after several attempts.
- We introduce a selective caching of vertices to identify solution parts that are more likely to be relevant for forthcoming decisions (e.g., because they were changed more recently). This technique is used to selectively optimize restricted solution areas.
- We develop a semi-structured organization of local search operators to achieve the best compromise between diversification and intensification, likelihood of escaping from local optima, and execution time.
- We refine the integration of GNs and SMDs in light of the above concepts. We also provide guidelines on the implementations of local search operators that (to the best of our knowledge) were never encoded into the SMD framework, such as the general CROSS-exchange (see Taillard et al. (1997)) and the ejection-chain (see Glover (1996)) operators.
- We detail a strategy to iteratively adapt the strength of a compatible shaking procedure based on the quality and structure of instances and solutions.

Finally, as the result of a thorough analysis we are able to combine the above defined concepts to obtain a fast, scalable, and effective CVRP solution algorithm.

The paper is structured as follows. Section 2 describes the details of our solution approach. Section 3 provides the experimental results, and Section 4 offers an experimental analysis of the algorithm components. Finally, the Appendix contains supplemental material covering implementation details.

2. Solution Approach

The metaheuristic we propose, whose high-level pseudocode is shown in Algorithm 1, consists of a *construction* phase (Line 3), which builds an initial feasible solution using a restricted version of the savings algorithm (see Clarke and Wright (1964)). Then, it follows an *improvement* phase (Lines 4-6) aimed at further enhancing the initial solution quality. More precisely, the improvement phase may first employ a *route minimization* procedure, to possibly reduce the number of routes in the initial solution when it is considered to be using more routes than necessary. A *core optimization* procedure, which is the central part of our algorithm, then uses an iterative, and localized optimization scheme to further improve the solution quality. Both route minimization and core optimization follow the Iterated Local Search paradigm (ILS, see Lourenço, Martin, and Stützle (2003)) in which shakings, performed in a ruin-and-recreate fashion (see Schrimpf et al. (2000)), and local search applications interleave for a prefixed number of iterations. The result is a Fast ILS Localized Optimization algorithm, shortened to FILO. The following paragraphs provide a detailed description of

the algorithm, which is the outcome of an iterative design process based on the analyses described in Section 4. In particular, we first describe the construction phase, then the local search engine, which is a central component of the improvement phase, and finally, the section ends with the definition of the improvement procedures.

Algorithm 1 High-level FILO structure

```
1: procedure FILO(I, s)
2:     R ← RandomEngine(s)
3:     S ← Construction()
4:     k ← GreedyRoutesEstimate(I)
5:     if |S| > k then S ← RouteMin(S, R)
6:     S ← CoreOpt(S, R)
7:     return S
8: end procedure
```

Notation: $\mathcal{I}$ instance, s seed

2.1. Construction

The initial solution is built using an adaptation of the well-known savings algorithm by Clarke and Wright (1964). As was already shown by other authors (see, e.g., Arnold, Gendreau, and Sörensen (2019)), the savings computation, which is quadratic in nature, can be linearized by considering for each customer $i \in V_c$ a restricted number n_{cw} of neighbors $j \in \mathcal{N}_i^{n_{cw}}(V_c)$ for which the saving value is computed. By limiting the number of savings, one can speed up the construction process without significantly harming the quality and compactness (i.e., the number of routes) of initial solutions especially when working with very large-scale instances. Note in addition that, since the construction phase is executed only once per run, over-optimizing it does not substantially contribute to the efficiency of the whole algorithm. As suggested in Arnold, Gendreau, and Sörensen (2019), we set $n_{cw} = 100$, and we compute the savings values for the arcs connecting each customer i to its n_{cw} neighbor customers j using a lexicographic order for the customers so as to avoid symmetries. More precisely, this set is given by the arcs $\{(i,j) : i \in V_c, j \in \mathcal{N}_i^{n_{cw}}(\{j \in V_c : i < j\})\}$. In fact, as reported in Section 4.1, larger n_{cw} values did not create significant differences either in the quality or compactness of the solutions.

2.2. Local Search Engine

Improvement procedures are designed around a complex local search engine making use of a tight integration of GNs, SMDs, and a novel selective vertex caching whose details are described in Sections 2.2.2 to 2.2.4. The result is a very fast and extremely localized local search execution, exploring neighborhoods induced by the following operators:

- an exchange of a contiguous sequence, or *path*, of n vertices with another path of m vertices belonging either to the same or a different route, see CROSS-exchange, Taillard et al. (1997). In the following this exchange is referred to as nmEX. For example, 21EX identifies the case in which $n = 2$ and $m = 1$. In particular, we implement the neighborhoods associated with $n, m = 0, \ldots, 3$ such that $n \geq m$ and nmEX is equivalent to mnEX;
- variants for 20EX, 21EX, 22EX, 30EX, 31EX, 32EX and 33EX, in which the first path of n vertices is reversed before being exchanged, called nmREX;
- variants for 22EX, 32EX and 33EX, in which both paths are reversed before being exchanged, called nmREX*;
- an intra-route 2-opt procedure, called TWOPT, as it is designed for the Traveling Salesman Problem, see Reinelt and Rinaldi (1994);
- two inter-route adaptations of the 2-opt procedure, called TAIL and SPLIT, both working on two different routes at a time. By denoting with head, a path of vertices belonging to the initial part of a route, and with tail, a path of vertices belonging to the final part of a route, TAIL swaps the tail of the two involved routes at some point, whereas SPLIT cuts the two routes at some point, then it replaces the tail of the first route with the reversed head of the second route and the head of the second route with the reversed tail of the first route;
- finally, an ejection-chain procedure, called EJCH, implementing the first improving sequence of 10EX, found by exploring a restricted number of sequences. That is, starting from an initial 10EX, a tree of at most n_{EC} nodes representing partial sequences is built. The sequence with the most improving value is always explored first, and a number of relocations (10EX) are generated by ejecting customers that restore the feasibility of the current route sequence endpoint. A 10EX may visit the same route more than once and no limit on the length of a sequence is imposed. As soon as a sequence is found to restore the feasibility of the target route, the associated 10EX exchanges are implemented. For more details, see Appendix B.3.8.

The above operators are structured in a Hierarchical Randomized Variable Neighborhood Descent (see Section 2.2.1), whose aim is to define a balanced combination of intensification-diversification, likelihood of escaping from local optima, and execution efficiency.

The next paragraphs provide a detailed description of the individual components of the local search engine that are extensively used in both improvement procedures. Finally, the section ends with characterizations of the route minimization and core optimization.

2.2.1. Hierarchical Randomized Variable Neighborhood Descent. We propose a local search architecture based on a combination of the Variable Neighborhood Descent (VND, see, e.g., Duarte et al. (2018)) and the Randomized VND (RVND, see, e.g., Subramanian, Uchoa, and Ochi

(2013)) principles. Both VND and RVND consider a set of local search operators that are sequentially applied to a solution S, generating a so-called neighborhood of S containing a number of neighbor solutions, or simply neighbors, of S. The key difference between VND and RVND is the criterion defining the order in which those operators are applied.

In the VND, operators are generally sorted in increasing neighborhood cardinality, with larger neighborhoods possibly including smaller ones. A typical example is a sequence of k-opt operators, with $k = 2, 3, \ldots, \ell$. This order has an efficiency purpose, because smaller neighborhoods are faster to explore, and a functional purpose, because larger neighborhoods are used to escape from the local optima of smaller ones. Whenever an improving neighbor is found, the search is restarted from the initial smallest neighborhood.

In contrast, in the RVND, the sequence of operators is randomly shuffled before each local search execution. This approach is used when neighborhoods induced by local search operators are not related one to another or have the same cardinality, because there are no well-defined guidelines providing hints about the order that will eventually lead to the best possible outcome. Relying on randomness is thus a reasonable approach that does not bias the search towards any operator, provides a natural diversification that still improves the objective function, and prevents the designers from enforcing a neighborhood exploration order that might not be ideal. When an improvement is found, all the operators are re-considered (after possibly being re-shuffled).

The Hierarchical RVND (HRVND) we propose mixes the two principles by defining a slightly more structured neighborhood exploration strategy, in which the operator order is neither completely random nor fixed a priori. More precisely, local search operators are arranged in *tiers*. Each tier is a compound operator that applies its subset of local search operators by following the RVND principles. The overall HRVND links the tiers together once they have been ordered according to the criteria defined by the VND, such as the overall computational complexity of the operators involved in the tier. The HRVND can thus be seen as a standard VND in which each tier is a compound local search operator and where successively more expensive tiers are used to escape from the local optima of the previous ones.

The proposed HRVND local search applies the operators described in Section 2.2 organized in the following two tiers: (i) 10EX, 11EX, SPLIT, TAILS, TWOPT, 20EX, 21EX, 22EX, 20REX, 21REX, 22REX, 22REX*, 30EX, 31EX, 32EX, 33EX, 30REX, 31REX, 32REX, 33REX, 32REX* and 33REX*, and (ii) EJCH. The first tier contains operators defining neighborhoods of quadratic cardinality and having very similar execution times; whereas the second tier contains the most expensive operator employed by the local search engine. More details about computing times and improving power are provided in the analysis in Section 4.2 and in Section A.2 of the Appendix.

Algorithm 2 HRVND tier application

procedure TierApplication($S, \mathcal{O}, \mathcal{R}$)
  $\mathcal{O} \leftarrow$ Shuffle($\mathcal{O}, \mathcal{R}$)
  $e \leftarrow 0, c \leftarrow 0$
  repeat
    $S' \leftarrow$ Apply($\mathcal{O}_c, S$)
    if Cost(S') < Cost(S) **then** $S \leftarrow S', e \leftarrow c$
    $c \leftarrow (c+1)$ mod Length($\mathcal{O}$)
  until $c \neq e$
  return S
end procedure

Notation: $\mathcal{O}$ list of tier operators, $\mathcal{O}_c$ operator in position c, $\mathcal{R}$ random engine.

Each tier stores its operators in a circular list which is shuffled before the application (see Algorithm 2). The neighborhood associated with an operator is completely explored and all the improvements are applied before moving to the next operator on the list. More details about the neighborhood exploration are given in Section 2.2.3 and in Section B.2 of the Appendix. The next tier is only applied when the solution is a local optimum for the previous tiers. Moreover, as in the standard VND setting, if the solution improved after a complete tier execution, the search is restarted from the initial tier (see Algorithm 3).

Algorithm 3 HRVND application

procedure hrvnd($S, \boldsymbol{T}, \mathcal{R}$)
  $e \leftarrow 0$
  repeat
    $S' \leftarrow$ TierApplication($S, \boldsymbol{T}_e, \mathcal{R}$)
    if Cost(S') < Cost(S) $\wedge\, e > 0$ **then**
      $S \leftarrow S', e \leftarrow 0$
    else
      $e \leftarrow e + 1$
    end if
  until $e <$ Length($\boldsymbol{T}$)
  return S
end procedure

Notation: $\boldsymbol{T}$ list of tiers, $\boldsymbol{T}_e$ operators in tier indexed e, $\mathcal{R}$ random engine

2.2.2. Move Generators and Granular Neighborhoods. A *move generator* $(i, j) \in E$ is an arc that, as the name suggests, is used to generate and identify a *unique* move in a local search context. In Toth and Vigo (2003), a set T of move generators is used to define a restricted local search

neighborhood, also known as a *granular neighborhood* (GN), containing a subset of the possible moves associated with a local search operator.

One way to select the arcs of interest is to use a *sparsification rule*. For example, in Toth and Vigo (2003) and Accorsi and Vigo (2020), arcs are chosen if their cost is below a given threshold, while in Schneider, Schwahn, and Vigo (2017), the reduced cost coming from a simple relaxation is used for the same purpose. Performing a local search by considering moves induced by move generators in T only allows to linearize the search time to $O(|T|)$.

In our approach, to speed up the local search execution, all neighborhoods induced by local search operators are implemented as GNs.

We define the set T of move generators to contain all arcs connecting a vertex $i \in V$ to its $n_{gs} = 25$ nearest vertices. More precisely, the set $T = \cup_{i \in V}\{(i,j),(j,i) \in E : j \in \mathcal{N}_i^{n_{gs}}(V \setminus \{i\})\}$. Note that move generators are described by arcs instead of edges. In fact, a GN is said to be asymmetric if the move induced by (i,j) is different from that induced by (j,i), and symmetric otherwise. All the local search operators we considered, with the exception of TWOPT and SPLIT, identify asymmetric GNs. The defined sparsification rule comes directly from the original GN definition by Toth and Vigo (2003), where the emphasis was on trying to replace long edges with short ones. However, in their work the sparsification is based on a cost rule selecting all edges having a cost lower than a given threshold, while here we adopt a nearest-neighbor rule that ensures a minimum number of move generators involving any vertex. A cost rule may in fact not be well suited when the standard deviation among arc costs is high, e.g., in clustered instances, and may cause several vertices not to have any move generator associated with them when the threshold is very low.

As described in previous works such as those by Schneider, Schwahn, and Vigo (2017) and Accorsi and Vigo (2020), by using an additional value called a *sparsification factor*, one could further filter the set of move generators according to some criteria, typically based on the arc cost, resulting in a dynamic GN based on a dynamic set of *active* move generators. In the following, a move generator is said to be active when selected by the current sparsification factor.

In our implementation, instead of using a single sparsification factor, we propose a more fine-grained management of dynamic move generators by means of a *vertex-wise* sparsification factor $\gamma_i \in [0,1]$ for each vertex $i \in V$. The dynamic set of active move generators for a *sparsification vector* $\boldsymbol{\gamma} = (\gamma_0, \gamma_1, \ldots, \gamma_N)$ is identified by $T^{\boldsymbol{\gamma}} = \cup_{i \in V}\{(i,j),(j,i) \in E : j \in \mathcal{N}_i^{k_i}(V \setminus \{i\})\}$ and $k_i = \lfloor \gamma_i \cdot n_{gs} \rceil$, where $\lfloor x \rceil$ denotes the nearest integer to x. Because the local search is indeed local, precise control over the move generators may allow the search to be tailored for specific areas where it is more needed. For example, the number of move generators may be increased for a part of the solution in which no (local) improvement has happened after several search attempts. Because GNs are used

in both improvement procedures, the precise description of the sparsification vector $\boldsymbol{\gamma}$ management is detailed in their respective sections.

Finally, several papers (e.g., Schneider, Schwahn, and Vigo (2017)), support the inclusion of all the edges connecting the depot to customers in the set of move generators. However, in our experience, when scaling to large-scale instances, even the dynamic management of the move generators becomes very challenging, causing a significant increase in computing time without guaranteeing any improvements to the solution quality. More details are given in Section 4.4.

2.2.3. Static Move Descriptors. *Static move descriptors* (SMDs), introduced for compound neighborhoods by Zachariadis and Kiranoudis (2010) and later adapted to the VND setting by Beek et al. (2018), enable the efficient execution of local search procedures by replacing the classical for-loop exploration of neighborhoods with a structured inspection of the moves associated with a local search operator, through the careful design of specialized data structures and procedures.

SMDs can be used to thoroughly describe the neighborhood of a solution. Every SMD identifies a *unique* local search move generating a neighbor solution and the associated change in the objective function value, called δ-tag. The combination of GNs and SMDs arises very naturally. In fact, a move generator uniquely defines a move within a GN and thus unambiguously identifies an SMD. On the other hand, an SMD uniquely describes a move (and its effect on the objective function) and thus unambiguously identifies the move generator inducing that move. For this reason, in the rest of the paper we will use SMD and move generator interchangeably.

A local search operator whose neighborhood is designed according to the SMD principles requires four stages. First, an *initialization* stage, executed once at the beginning of the neighborhood exploration, computes the δ-tag for the available SMDs. Then, a sequence of *search*, *execution*, and *update* stages is performed until no more improving solutions are available in the neighborhood. The search stage looks for a *feasible and improving* SMD - that is, an SMD associated with a move that generates a feasible and improving neighbor solution. Moreover, the SMD might also be required to meet some additional criteria (e.g., the SMD associated with the most improving feasible move might be sought). Once found, the move associated with the SMD that meets the criteria is executed during the execution stage. Because a local search application causes only a local change in a solution, most of the SMDs will still hold a correct δ-tag even after (part of) the solution is changed. An operator-specific list of SMDs requiring an update to their δ-tag can thus be considered during the update stage. Finally, the neighborhood exploration ends when the search stage is no longer able to identify a feasible and improving SMD.

In our implementation, all (granular) neighborhoods are designed according to the SMD principles, and a binary heap is used to store the SMDs, corresponding to active move generators, organized

according to their δ-tag. During the initialization stage, only improving SMDs are inserted into the heap, to keep the computational complexity of managing the heap to a minimum. As in Beek et al. (2018), during the search stage, instead of retrieving the most improving SMD by removing infeasible SMDs until a feasible one is found, we linearly scan the vector representing the heap data structure searching for a feasible SMD. This approach does not require the re-insertion of removed SMDs once a feasible move is found, but it no longer guarantees the execution of the most improving move. However, since the SMDs are roughly sorted by the heap's internal procedures, the linear scan provides a so called *rough-best-improvement* move acceptance strategy. Note that the heap data structure is not unique for a set of entries, and the linear scan is highly affected by the order in which heap management operations are executed. For more details we refer the reader to Sections B.2 and B.3 of the Appendix.

Finally, to speed up the initialization stage we coupled it with the selective vertex caching strategy described in Section 2.2.4 that forces the local search to focus on recently changed areas of the solution.

2.2.4. Selective Vertex Caching. Sparsification rules describing GNs are complemented by the identification of a set of vertices of interest by means of a *selective vertex caching* (SVC) strategy. In particular, every solution S keeps track of a subset of vertices $\bar{V}_S \subseteq V$ that, for the current algorithm state, is considered to be highly relevant.

In our implementation, $\bar{V}_S$ consists of the set of vertices directly belonging to solution areas that recently underwent some changes. Without loss of generality, changes in a solution S, seen from a vertex perspective, can be subdivided into insertions and removals. For example, consider the relocation of vertex i from its original position to another one between vertex j and its predecessor π_j. The removal *directly* affects vertex i itself, its successor σ_i, and its predecessor π_i, while the subsequent insertion affects vertices i, j and π_j. Within this setting, after the move execution, we say that i, π_i, σ_i, j, and π_j are *cached* for S: i.e., they are inserted into $\bar{V}_S$. This strategy allows us to easily keep track of solution areas that were recently changed. However, not all changes have the same importance; more recent ones are more likely to be relevant to a forthcoming decision. This aspect is captured by imposing a limit, for a solution S, on the maximum number of vertices that can be cached at the same time equal to a constant value C, by imposing $|\bar{V}_S| \leq C$ and adopting a *least recently used* strategy to maintain $\bar{V}_S$.

2.2.4.1. *SVC to Restrict Local Search Execution.* Vertices can be selectively cached in order to identify a kernel of relevant vertices to be used in local search procedures. As mentioned at the end of Section 2.2.3, we used this method as a heuristic acceleration and filtering technique for the initialization stage of the SMDs, which as shown in the analysis of Section 4.5, may also have a significant influence on subsequent SMD stages and, ultimately, on the overall local search execution.

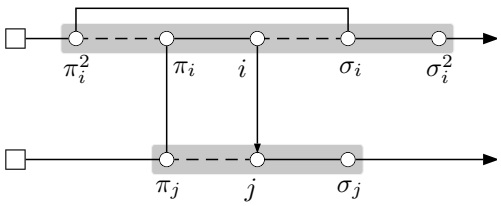


Figure 2 **A 20ex application induced by move generator (i, j) relocating path $(\pi_i - i)$ between π_j and j. SMDs involving vertices in the gray area require an update to their δ-tag after the move execution.**

During the optimization of a solution S, the SMD initialization stage considers, for the heap insertion, the restricted subset of move generators $(i,j) \in \bar{T}^{\gamma}(S) \subseteq T^{\gamma}$, such that at least one of the endpoints i or j belongs to the subset of cached vertices $\bar{V}_S$. More precisely, $\bar{T}^{\gamma}(S) = \cup_{i \in \bar{V}_S}\{(i,j),(j,i) \in E : j \in \mathcal{N}_i^{k_i}(V \setminus \{i\})\}$ with $k_i = \lfloor \gamma_i \cdot n_{gs} \rceil$. During subsequent SMD update stages, additional move generators might, however, be added to the heap due to the search incrementally extending to vertices not belonging to the selective cache and whose SMDs require an update although they have not been directly involved in a change of the solution. To better understand this, consider the scenario shown in Figure 2 in which, during a 20EX neighborhood exploration, a move induced by move generator (i,j) is executed, causing the relocation of path $(\pi_i - i)$ between π_j and j. Once the move is executed, vertices $\pi_i^2, \pi_i, i, \sigma_i, \pi_j$ and j will be marked as cached. However, as shown by the gray overlay, two additional vertices, namely σ_i^2 and σ_j are also (indirectly) affected by the move execution. In particular, active move generators involving σ_i^2, i.e. $\{(\sigma_i^2, j) : j \in V\} \cap T^{\gamma}$, require an update because the predecessor of σ_i changes from i to π_i^2. Similar reasoning applies to some move generators involving σ_j. More details about update lists associated with different local search operators are given in Section B.3 of the Appendix. Note that σ_i^2 and σ_j do not belong to the cache, but their move generators will be updated and, if improving, inserted into the heap and considered during subsequent SMD search stages.

Move generators evaluated during the SMD search stage could hence have been considered, because they involve vertices belonging to the selective vertex cache or vertices indirectly affected by a previous move application. Thus, it is clear that the cache dimension C actually imposes a soft constraint on the SMDs considered during the local search execution: starting from the restricted kernel of cached vertices, the search may extend to include move generators involving all instance vertices.

A possible scenario in any of the improvement procedures, analyzed from the perspective of the number of distinct vertices either cached or (directly and indirectly) reached by the local search execution, is depicted in Figure 3. Improvement procedures, at the beginning of each iteration, work with solutions S having no cached vertices, i.e., $\bar{V}_S = \emptyset$. As mentioned in Section 2.2, both procedures make use of a shaking performed in a ruin (R^-) and recreate (R^+) fashion. The vertices involved in

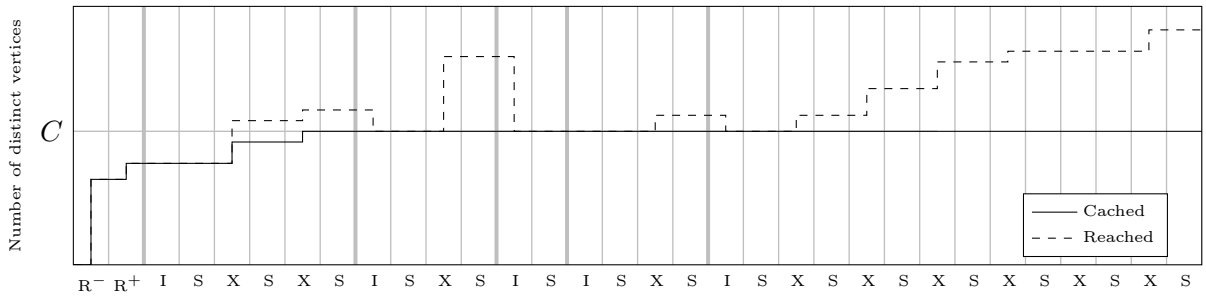


Figure 3 **Evolution of an improvement procedure iteration in terms of number of distinct vertices simultaneously cached and considered during a neighborhood exploration (Reached). The number of vertices is analyzed after the ruin (R^-), recreate (R^+), and local search operator applications. Each local search operator application is partitioned into an SMD initialization (I) and a sequence of search (S) and execute and update (X) stages.**

the disruptive effects of the shaking applied to solution S are added to the set $\bar{V}_S$. This set is the kernel of vertices used to identify the area where the optimization of the subsequent local search execution is focused. In particular, each SMD initialization stage (I) considers the current kernel of cached vertices. Then, a sequence of SMD search (S) and execution and update (X) stages might cause the search to reach far more vertices than those cached (dashed line), potentially covering all vertices. However, as discussed in Section 4.5, the maximum size C of this kernel indirectly affects the exploration power as well as the computing time of the local search. The overall result is an implicit dynamic instance decomposition, induced by a very focused and localized neighborhood exploration strategy which mainly considers the areas of the solution that are more likely to require further optimization because they were more recently changed.

2.3. Improvement

Improvement procedures are iterative randomized local search-based procedures aimed at further enhancing the initial solution quality. Both procedures, namely route minimization and core optimization, work by re-optimizing, through the local search engine, a restricted area disrupted by a ruin-and-recreate application. This area is identified by a number of vertices stored in the selective vertex cache. At the beginning of each improvement procedure iteration, the cache is emptied to perform an optimization focused primarily on the very limited solution area identified by the upcoming shaking application. The route minimization procedure may visit infeasible solutions to perform its route compacting action, while the core optimization procedure only moves in the feasible space and achieves its diversification by means of an effective simulated annealing strategy.

2.3.1. Route Minimization. The CVRP typically does not impose any limit on the number of routes that solutions may have. However, there is often a positive correlation between the number

of routes in a solution and its cost. Moreover, many simple construction algorithms, such as the one we use, typically produce solutions using far more routes than those found in good-quality solutions. We thus include an optional route minimization procedure that may be executed right after the initial solution construction.

This procedure is applied to a solution S built by the initial construction phase whenever the number $|S|$ of its routes is found to be greater than an ideal estimated number of routes k. The value k is computed by heuristically solving a bin-packing problem with an item of weight q_i for each customer $i \in V_c$ and bins of capacity Q through a simple greedy first-fit algorithm (see, e.g., Chapter 8 of Martello and Toth (1990)).

The route minimization procedure, whose pseudocode is shown in Algorithm 4, starts by setting the best found solution S^* to be equal to the initial solution S generated by the construction phase. During each iteration a pair (r_i, r_j) of routes belonging to S is selected and their customers removed from the solution and placed into a list of unrouted customers L (Lines 5-7). The first route r_i is chosen as the route containing a random customer seed i. The second route r_j is identified by considering customer neighbors $j \in \mathcal{N}_i$ in increasing c_{ij} cost until a customer j belonging to a route $r_j \neq r_i$ is found. Customers in L are, with equal probability, either randomly shuffled or sorted according to their demand, in decreasing order. Then, for each unrouted customer $i \in L$, a position in the current set of existing routes is searched, such that with the insertion of i the target route remains feasible and the insertion cost is minimized (Lines 10-21). When such a position cannot be found (i.e., when inserting i violates the capacity constraints of all existing routes), and consequently a new route should be created to accommodate customer i, an action is selected according to the current number of routes $|S|$. If $|S|$ is lower than the estimate k, a new single-customer route with i is created. Otherwise, the single-customer route serving i is created only when a random number drawn from a uniform real distribution in the interval $[0, 1]$ is greater than a threshold $\mathcal{P}$, set to $\mathcal{P} = 1$ at the beginning of the route minimization procedure. When the random number is not greater than $\mathcal{P}$, i is inserted into an additional list $\bar{L}$. Once all customers in L have been considered, the list L is set to $L = \bar{L}$ and the threshold $\mathcal{P}$ is lowered according to an exponential schedule $\mathcal{P} = z \cdot \mathcal{P}$ with $z = (\mathcal{P}_f/\mathcal{P}_0)^{(1/\Delta_{RM})}$, $\mathcal{P}_f = 0.01$, and $\mathcal{P}_0 = 1$. Where $\mathcal{P}_f$ and $\mathcal{P}_0$ are the final and initial probabilities of not creating an additional single-customer route, respectively.

Next, a restricted HRVND, consisting of the first tier only but using all the available move generators, i.e., $\gamma_i = 1, i \in V$, is applied to the (possibly partial) current solution. We restrict the HRVND to the first tier because we noticed it was sufficient to obtain good-quality solutions and resulted in a considerable computing time saving. Moreover, we set $\gamma_i = 1, i \in V$ to avoid a complex management of move generators in this secondary improvement phase which, as shown in the parameters Table 1, is executed for a small number $\Delta_{RM} = 1000$ of iterations.

Algorithm 4 Route minimization procedure

1: **procedure** RouteMin($S, \mathcal{R}$)
2: $S^* \leftarrow S, \mathcal{P} \leftarrow \mathcal{P}_0, L \leftarrow [\,]$
3: **for** $n \leftarrow 1$ to Δ_{RM} **do**
4: $\bar{V}_S \leftarrow \emptyset$
5: $(r_i, r_j) \leftarrow$ PickPairOfRoutes($S, \mathcal{R}$)
6: $L \leftarrow [L,$ CustomersOf$(r_i),$ CustomersOf$(r_j)]$
7: $S \leftarrow S \setminus r_i \setminus r_j$
8: $L \leftarrow$ DefineOrder($L, \mathcal{R}$)
9: $\bar{L} = [\,]$
10: **for** $i \in L$ **do**
11: $p \leftarrow$ BestInsertionPositionInExistingRoutes(i, S)
12: **if** $p \neq none$ **then**
13: $S \leftarrow$ Insert(i, p, S)
14: **else**
15: **if** $|S| < k \vee U(0,1) > \mathcal{P}$ **then**
16: $S \leftarrow$ BuildSingleCustomerRoute(i, S)
17: **else**
18: $\bar{L} \leftarrow [\bar{L}, i]$
19: **end if**
20: **end if**
21: **end for**
22: $L \leftarrow \bar{L}$
23: $S \leftarrow$ hrvnd.tier1($S, \mathcal{R}$)
24: **if** $|L| = 0 \wedge (\text{Cost}(S) < \text{Cost}(S^*) \vee (\text{Cost}(S) = \text{Cost}(S^*) \wedge |S| < |S^*|))$ **then**
25: $S^* \leftarrow S$
26: **if** $|S^*| \leq k$ **then return** S^*
27: **end if**
28: $\mathcal{P} \leftarrow z \cdot \mathcal{P}$
29: **if** $\text{Cost}(S) > \text{Cost}(S^*)$ **then** $S \leftarrow S^*$
30: **end for**
31: **return** S^*
32: **end procedure**

When a solution S in which all customers are routed is found, the best solution S^* is replaced with S if the latter has a lower cost or the same cost but fewer routes. Moreover, we impose an early stopping condition (Line 26) such that if S^* has a number of routes less than or equal to k, the route minimization procedure prematurely ends and returns S^*.

Before proceeding to the next iteration, a partial or feasible solution S having a cost higher than the current best solution S^* is reset to the latter; i.e., $S = S^*$.

Finally, solution S^* is returned after Δ_{RM} iterations if the early stopping condition is not met throughout the route minimization execution.

2.3.2. Core Optimization. The core optimization procedure, whose pseudocode is shown in Algorithm 6, is iterative and randomized. By making use of an adaptive shaking strategy and the local search engine, it implements a powerful localized solution improvement.

First, the best solution S^* is set equal to solution S, generated by the construction phase and possibly improved by the route minimization procedure. A shaking application, whose pseudocode is shown in Algorithm 5, performs a ruin step, removing a number of customers in the customer sub-graph by means of a random walk. Then, a simple greedy recreate step defines the new position for the previously removed customers. More precisely, the ruin step, starting from a randomly selected seed customer $i \in V_c$, identifies a random walk of length ω_i in the sub-graph $G' = (V'_c, E'_c)$ where $V'_c = V_c$ and $E'_c = \{(i,j) : i,j \in V'_c\}$ is the set of arcs connecting customers. When a customer i is visited, it is removed from the solution and the sub-graph G' is updated accordingly by setting $V'_c = V'_c \setminus \{i\}$ and $E'_c = E'_c \setminus \{(i,j),(j,i) : j \in V'_c\}$. A partial walk ending at customer i is extended by moving either forward or backward within the same route r_i, or by jumping to a neighbor route, which can be any route or a not yet visited one (Lines 7-20). First, whether to move along the same route or jump to another is decided; then the possible options associated with that choice are considered. At every step, the choices are considered with equal probability. When a jump to a neighbor route is selected to extend a walk ending at $i \in V'_c$, customers $j \in \mathcal{N}_i(V'_c)$ are examined in increasing c_{ij} cost until a route r_j satisfying the appropriate requirements, i.e., either any route (including $r_j = r_i$) or a not yet visited one, is found. The r_j, identified by scanning the routes to which neighbor customers j of i belong, is considered to be a neighbor route of r_i. In the unlikely case that such a route cannot be found, the ruin procedure is prematurely aborted (Line 18). Note that a jump to a neighbor route is always selected when the current route r_i contains customer i only (Line 7).

The recreate step greedily inserts the removed customers into the position that minimizes the insertion cost after they have, with equal probability, either been randomly shuffled or sorted by decreasing demand or by increasing or decreasing distance from the depot.

The ruin intensity is controlled by a meta-procedure (described in Paragraph 2.3.2.1) that iteratively adapts the random walk length ω_i from a seed customer $i \in V_c$ to identify a disruptive action that best suits the instance and solution under examination.

The HRVND is then applied to the shaken solution S to identify a local optimum S'. Whether to accept S' as the next point in the search trajectory is determined by a simulated annealing acceptance strategy; see Kirkpatrick, Gelatt, and Vecchi (1983). In particular, S' is accepted if

Algorithm 5 Adaptive shaking procedure

procedure SHAKE($S, \mathcal{R}, \bar{\omega}$)
  $i' \leftarrow$ RANDOMCUSTOMER($\mathcal{R}$)
  $i \leftarrow i', e \leftarrow 0, C = [\,], R = \{\}$
  $\mathcal{B} \leftarrow$ RANDOMBOOLEAN($\mathcal{R}$)
  repeat
    $C \leftarrow [C, i], R \leftarrow R \cup \{r_i\}$
    if $|S| > 1 \wedge \mathcal{B}(\,)$ **then**
      if $\mathcal{B}(\,)$ **then**
        $j \leftarrow$ NEXTCUSTOMERINROUTE(i)
      else
        $j \leftarrow$ PREVCUSTOMERINROUTE(i)
      end if
    else
      if $\mathcal{B}(\,)$ **then**
        $j \leftarrow$ NEARESTSERVEDCUSTOMER(S, i)
      else
        $j \leftarrow$ NEARESTSERVEDCUSTOMER(S, i) **such that** $r_j \notin R$
        if $j = none$ **then** $e \leftarrow \infty$
      end if
    end if
    $S \leftarrow$ REMOVECUSTOMER(S, i)
    $i \leftarrow j$
  until $e < \omega_{i'}$
  $C \leftarrow$ DEFINEORDER($C, \mathcal{R}$)
  for $i \in C$ **do**
    $p \leftarrow$ BESTINSERTIONPOSITIONINEXISTINGROUTES(i, S)
    if $p \neq none$ **then**
      $S \leftarrow$ INSERT(i, p, S)
    else
      $S \leftarrow$ BUILDSINGLECUSTOMERROUTE(i, S)
    end if
  end for
  return S, i'
end procedure

$c(S') < c(S) + \mathcal{T} \cdot \ln U(0,1)$. The value of $\mathcal{T}$ is initially set to $\mathcal{T} = \mathcal{T}_0$ and lowered at the end of each core optimization iteration by performing $\mathcal{T} = c \cdot \mathcal{T}$ with $c = (\mathcal{T}_f / \mathcal{T}_0)^{(1/\Delta_{CO})}$; Δ_{CO} is the number of core optimization iterations.

The core optimization makes full use of vertex-wise dynamic move generators. In particular, at the beginning of the procedure, sparsification parameters γ_i are set to a base value γ_{base}. Whenever

$(\delta \cdot \Delta_{CO} \cdot \text{AVERAGE}(|\bar{V}_S|))/|V|$ nonimproving iterations involving a vertex i are performed, the value is updated as $\gamma_i = \min\{\gamma_i \cdot \lambda, 1\}$ where $\delta \in [0,1]$ is a reduction factor, $\text{AVERAGE}(|\bar{V}_S|)$ denotes the average number of vertices cached after previous local search executions, and λ is an increment factor. However, the value of γ_i is reset to γ_{base} whenever a solution S improving S^* is found during the execution of a local search involving i (i.e., $i \in \bar{V}_S$ after the HRVND application). This update rule is a possible vertex-wise implementation of the standard way of handling dynamic move generators described in Schneider, Schwahn, and Vigo (2017) in which the total number of core optimization iterations is partitioned over restricted working areas identified by cached vertices. Note that, when the cache size C is smaller than the total number of instance vertices, i.e., $C < |V|$, some vertex i may not be considered for the γ_i update even if it is involved in a change during the HRVND execution because, due to the limit imposed by C, it is no longer cached after the optimization. However, as shown in the analysis in Section 4.5, an accurate selection of C, which might heuristically filter out some vertices from the update, does not prevent good solutions from being found much faster than in the scenario in which $C = |V|$ and the SVC is completely disabled.

Algorithm 6 Core optimization procedure

procedure CoreOpt($S, \mathcal{R}$)
  $\bar{\omega} \leftarrow (\omega_0, \omega_1, \ldots, \omega_{|V_c|}), \omega_i \leftarrow \omega_{base}\ \forall i \in V$
  $\bar{\gamma} \leftarrow (\gamma_0, \gamma_1, \ldots, \gamma_{|V_c|}), \gamma_i \leftarrow \gamma_{base}\ \forall i \in V$
  $S^* \leftarrow S, \mathcal{T} \leftarrow \mathcal{T}_0$
  for $n \leftarrow 1$ to Δ_{CO} **do**
    $\bar{V}_S \leftarrow \emptyset$
    $\hat{S}, i' \leftarrow \text{Shake}(S, \mathcal{R}, \bar{\omega}), \mathcal{S} \leftarrow \bar{V}_{\hat{S}} \setminus \{0\}$
    $S' \leftarrow \text{hrvnd}(\hat{S}, \mathcal{R}), \mathcal{L} \leftarrow \bar{V}_{S'}$
    if $\text{Cost}(S') < \text{Cost}(S^*)$ **then**
      $S^* \leftarrow S'$
      ResetSparsificationFactors($\bar{\gamma}, \mathcal{L}$)
    else
      UpdateSparsificationFactors($\bar{\gamma}, \mathcal{L}$)
    end if
    UpdateShakingParameters($\bar{\omega}, S', S, i', \mathcal{S}, \mathcal{R}$)
    if AcceptNeighbor($S, S', \mathcal{T}$) **then** $S \leftarrow S'$
    $\mathcal{T} \leftarrow c \cdot \mathcal{T}$
  end for
  return S^*
end procedure

2.3.2.1. *Structure-Aware and Quality-Oriented Shaking Meta-Strategy.* We propose a declarative approach to the selection of the shaking intensity that, if coupled with a shaking procedure able to take advantage of it, improves flexibility and adaptability compared to a random or fixed intensity. This strategy makes use of a number of integer *shaking parameters* $\omega_i, i \in V_c$ defining the intensity of a shaking application seeded at customer i.

The idea is to iteratively adapt the parameters ω_i so that solution S', obtained by re-optimizing the disrupted area of solution S, meets some quality criteria with respect to S. On the one hand, S' could be of lower quality than S or, more precisely, the distance in terms of cost between S' and S is greater than an *intensification upper bound* threshold Ω_{UB}, i.e., $\textsc{Cost}(S') - \textsc{Cost}(S) > \Omega_{UB}$. From a simplified perspective, we may assume this happened because the initial disruption produced by the ruin was too strong, causing so much turbulence on the original solution S that subsequent local search procedures were not able to successfully correct and improve it. On the other hand, S' and S may be of comparable quality and in particular, $0 \leq \textsc{Cost}(S') - \textsc{Cost}(S) < \Omega_{LB}$ with Ω_{LB} an *intensification lower bound* threshold. In this case, the disruptive effect of the ruin was probably not strong enough to jump to a different search space area, and the subsequent local search procedures were able to partially undo the changes. Finally, S' may be better than S, showing that the combination of shaking and subsequent re-optimization was appropriate. In our implementation, we define $\Omega_{LB} = \bar{c}_S \cdot I_{LB}$ and $\Omega_{UB} = \bar{c}_S \cdot I_{UB}$, where $\bar{c}_S$ is the average cost of an arc in solution S, computed as $\bar{c}_S = \textsc{Cost}(S)/(N + 2 \cdot |S|)$; $I_{LB}, I_{UB} \in \mathbb{R}$ are shaking factors.

From the above observations, we can derive a simple update rule for the shaking parameters ω_i that is executed at every core optimization iteration (Line 15 of Algorithm 6). Denoting by $\tilde{w} = w_{i'}$ the shaking parameter value associated with the current seed customer i'

$$\omega_i = \begin{cases} \omega_i + 1, \text{ if } 0 \leq \textsc{Cost}(S') - \textsc{Cost}(S) < \Omega_{LB} \wedge \omega_i < \tilde{\omega} + 1 & (i) \\ \omega_i - 1, \text{ if } \textsc{Cost}(S') - \textsc{Cost}(S) > \Omega_{UB} \wedge \omega_i > \tilde{\omega} - 1 & (ii) \\ \text{randomly select between } (i) \text{ and } (ii), \text{ otherwise} & (iii) \end{cases} \qquad i \in \mathcal{S}$$

where $\mathcal{S} = \bar{V}_{\hat{S}} \setminus \{0\}$ is the set of vertices cached in the shaken solution $\hat{S}$ right after the shaking execution (excluding the depot, which is never considered in the ruin execution); see Line 7 of Algorithm 6. Shaking parameters $\omega_i, i \in \mathcal{S}$ are moved towards the new value for $\tilde{\omega}$, without exceeding it. This limit prevents situations in which a single vertex j surrounded by a set of vertices $\tilde{V}$, all having a very small (respectively, large) shaking parameter value has its w_j indirectly incremented (respectively, decremented) to very large (respectively, small) values due to updates involving some $i \in \tilde{V}$. Furthermore, update rule (iii) describes the scenario in which S' is improving with respect to S or the shaking was of the appropriate strength; i.e, $\Omega_{LB} \leq \textsc{cost}(S') - \textsc{cost}(S) \leq \Omega_{UB}$. Through experimentation, we found it beneficial to perform limited random variations of the involved shaking

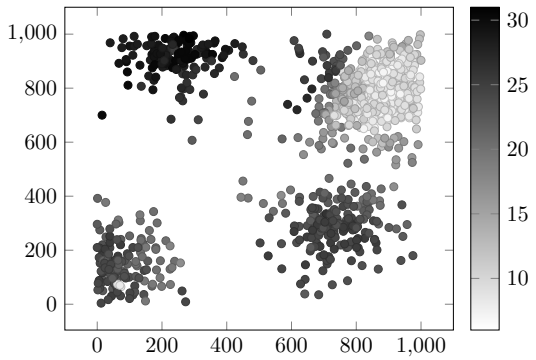


Figure 4 **Shaking parameters values at the end of the core optimization procedure for instance X-n979-k58 of the $\mathbb{X}$ dataset. Each circle represents a customer i in its x_i and y_i coordinates and the color denotes the shaking parameter ω_i value.**

parameter values, to avoid their stagnation to minimum values which satisfied rules (i) and (ii) in order to explore other promising combinations of values. Finally, the update depends on the specific area where the shaking was executed; the parameters are iteratively adjusted according to the effects on nearby areas caused by previous shaking applications. This adaptive procedure thus makes the shaking aware of both the structure of the instance and the solution under examination.

As an example, consider Figure 4, showing shaking parameter values $\omega_i, i \in V$ for instance X-n979-k58 of the $\mathbb{X}$ benchmark after the core optimization procedure. As can be seen from the figure, very dense areas of customers typically require lower values for the shaking parameters, whereas customers in sparse areas are associated with stronger shaking applications.

Note that set $\mathcal{S}$ also contains vertices involved in the recreate step. An ideal update rule should consider only customers involved in the ruin step or, even better, only the seed customer. However, especially for large-scale instances, this rule would require an enormous amount of iterations for the procedure to converge on reasonably effective shaking parameter values, which would likely still require an update as the algorithm evolves. We thus found that updating the shaking parameters for all vertices that are in the selective cache after the shaking application is a reasonable strategy to identify a number of vertices that are somehow related and can be thought of as belonging to the same area.

Finally, the initial value for the shaking parameters is not relevant for small-sized instances in which an initial value of $\omega_{base} = 1$ may be used. On the contrary, it becomes quite important when moving to very large instances, if the total number of core optimization iterations remains constant. In fact, on the one hand, using a small value might cause several fruitless shaking iterations in which ω_i values are slowly increased to more effective values, wasting precious computing resources with insufficient disruptions. On the other hand, a value that is too high might dramatically slow down the overall algorithm execution with the risk of an excessive ruin activity. We experimentally found

a reasonable compromise by setting $\omega_{base} = \lceil \ln |V| \rceil$ as the initial shaking intensity, which is then automatically adjusted by the above update rules.

3. Computational Results

The computational testing has the main objective of assessing the performance of the proposed algorithm. To accomplish this, we first show its effectiveness on the $\mathbb{X}$ dataset proposed by Uchoa et al. (2017), on which state-of-the-art CVRP algorithms are typically evaluated. Then, we proceed to the real target of the paper, showing how the designed components allow the overall algorithm to easily scale to very large-scale instances while still retaining its effectiveness. To this end, we focus on the increasingly popular $\mathbb{B}$ instances proposed by Arnold, Gendreau, and Sörensen (2019) and on two less-studied very large-scale datasets, $\mathbb{K}$ and $\mathbb{Z}$ proposed by Kytöjoki et al. (2007) and Zachariadis and Kiranoudis (2010), respectively.

3.1. Implementation and Experimental Environment

The algorithm was implemented in C++ and compiled using g++ 8.3.0. The experiments were performed on a 64-bit desktop computer with an Intel Xeon CPU E3-1245 v5 central processing unit (CPU), running at 3.5 GHz and with 16 GB of RAM on a GNU/Linux Ubuntu 18.04 operating system. The algorithm source code, together with a library of reusable components, can be downloaded from `https://acco93.github.io/filo/`; detailed instructions are given to accurately reproduce our results. In all the computational testing, we considered a standard version of FILO and a longer version, called FILO (long), which performs ten times more core optimization iterations than the standard version. Because of the randomized nature of the algorithm, for every experiment, we executed a symbolic number of fifty runs for each instance, defining the seed of the pseudorandom engine (the Mersenne twister of Matsumoto and Nishimura (1998)) as equal to the run counter minus one. Moreover, to mitigate the impact of small time-variations due to the overhead of the operating system, we used a clock function that reports running times with the minimum recordable run time set to one second.

To better compare our results with other algorithms executed on different hardware configurations, for which no source code was available, we used the single-thread rating defined by PassMark® Software (2020). At the time of writing, a score of 2285 was assigned to our CPU. Competing methods' CPU times are scaled to match our CPU score; their normalized time is identified by $\hat{t} = t \cdot (P_A/P_B)$, where P_A is the competing method's CPU single-thread rating, P_B is our CPU rating, and t is the raw computing time. All times refer to an average run and are reported in minutes. For randomized algorithms we report, when available, the best (Best), average (Avg) and worst (Worst) gaps of the solution found by the algorithm, with respect to the best known solution value (BKS). Gaps are computed as $100 \cdot (\textsc{Cost}(S) - BKS)/BKS$, where S is the final solution. For deterministic algorithms, we report the gap (Gap) of the solution found by a single run.

Table 1 Parameters of Algorithm FILO.

Initial solution definition – Described in Sections 2.1 and 2.3.1 – Analyzed in Section 4.1	
$n_{cw} = 50$	Number of neighbors considered in the savings computation.
$\Delta_{RM} = 10^3$	Maximum number of route minimization iterations.
Granular neighborhood – Described in Section 2.2.2 – Analyzed in Section 4.4	
$n_{gs} = 25$	Number of neighbors considered by the sparsification rule.
$\gamma_{base} = 0.25$	Base sparsification factor.
$\delta = 0.5$	Reduction factor used in the definition of the fraction of non-improving iterations performed before increasing a sparsification parameter.
$\lambda = 2$	Sparsification increment factor.
Core optimization – Described in Sections 2.2, 2.2.4, 2.3.2, and 2.3.2.1 – Analyzed in Sections 4.2, 4.5 and 4.3	
$\Delta_{CO} = 10^5, 10^6$	Number of core optimization iterations for a short and a long run.
$\mathcal{T}_0$, $\mathcal{T}_f$	Initial and final simulated annealing temperature.
$C = 50$	Maximum number of cached vertices.
$\omega_{base} = \lceil \ln \lvert V \rvert \rceil$	Initial shaking intensity.
$n_{EC} = 25$	Maximum number of sequences explored by EJCH for each move generator.
$I_{LB} = 0.375, I_{UB} = 0.85$	Shaking factors.

3.2. Parameter Tuning

Crafting algorithm FILO and tuning its parameters followed an iterative process whose key decisions are detailed in Section 4. Parameters are summarized in Table 1; their tuning, whose hidden interactions and interconnected effects might be very challenging to analyze, followed a straightforward sequential strategy aimed at keeping the tuning effort low but still able to identify good performing values for each parameter considered individually. First, reasonable values were identified using the authors' judgment and experience, along with a trial-and-error approach. Then we evaluated the algorithm's behavior while changing the value of one parameter at a time (keeping the others fixed). A new value was kept when it allowed an improvement in quality without increasing the computing time. This process was iterated several times until satisfactory results were obtained. In fact, we noticed that the iterated sequential tuning of individual parameters, without a prefixed order, was enough to reach good local optima without exploring all possible combinations of values. The parameter tuning, as well as the algorithm design, was mainly performed by considering the largest $\mathbb{X}$ instances - in particular, those with more than five hundred vertices. The resulting tuned algorithm was then used for all our computational testing.

We briefly summarize, in the following, the key choices made during the parameter tuning procedure, referring to Section 4 for more details.

The number of customers n_{cw} for which the savings are computed in the construction phase and the maximum number of route minimization iterations Δ_{RM} are both low impact parameters. Once they are set to reasonable values, small variations do not significantly change the outcome of the procedures in which they are employed. We set them approximately to one of the smallest values

able to provide results of a quality comparable to larger values, which may, however, have required a longer computing time.

Conversely, the value of n_{gs}, γ_{base}, δ and λ heavily affect the algorithm performance. The number of neighbors n_{gs}, the base sparsification factor γ_{base} and the reduction factor δ were set so as to obtain the best trade-off between computing time and solution quality; see Section 4.4. In particular, we noticed that a milder sparsification (obtained, for example, by increasing n_{gs}, γ_{base} or both; or by decreasing δ) may sometimes provide slightly better solutions, but with an unacceptable increment in the computing time. This is particularly noticeable in instances for which very good-quality (or near-optimal) solutions are found early in the search, and for which the sparsification is just steadily decreased by increasing the $\gamma_i, i \in V$ and never reset; see, e.g., the computing time for instance X-n219-k73 of the $\mathbb{X}$ dataset in Table 9 (Section C of the Appendix) and the outliers associated with FILO in Figure 1. We set the the value for λ as in the original proposal of Toth and Vigo (2003), leaving the other granular parameters depending on it. On average, we found a very aggressive sparsification associated with an large number of optimization iterations to be preferable to a very accurate local search execution performed with fewer iterations.

The number of core optimization iterations Δ_{CO} was set appropriately to suit all the medium to very-large instances we considered. However, to alleviate the above-mentioned indiscriminate increment for γ_i values in small-sized instances, which typically converge to the final value faster, defining the number of iterations as a function of the instance size might be more appropriate. Nonetheless, we preferred not to use that approach in order to better highlight the scalability properties of FILO.

The simulated annealing temperatures $\mathcal{T}_0$ (initial) and $\mathcal{T}_f$ (final) were defined to be proportional to the average cost of an arc in an instance. In particular, the value of $\mathcal{T}_0$ is defined as 0.1 times the average instance arc cost; i.e., $\mathcal{T}_0 = 0.1 \cdot \sum_{i,j \in V: i<j} c_{ij}/(|V| \cdot (|V|-1)/2)$ and $\mathcal{T}_f$ is 0.01 times $\mathcal{T}_0$. In fact, we found this strategy to be better than defining a fixed range when applied to different datasets with completely unrelated arc costs. For example, the average arc cost for the $\mathbb{K}$ dataset is about 3, 500% larger than that for the $\mathbb{X}$ dataset and 164,100% larger than that for the $\mathbb{Z}$ dataset.

The size of the selective cache C was chosen to be the value that identified a good trade-off between computing time and solution quality; see Section 4.5.

As mentioned in Section 2.2.4, both small and large-scale instances are not significantly affected by the choice of the initial shaking intensity ω_{base}; in fact, comparable quality and computing time are obtained by setting $\omega_{base} = 1$. However, when considering very large-scale instances, the difference in the final outcome is much more noticeable. We found that setting ω_{base} as a logarithmic function of the number of vertices provides a reasonable starting point. The shaking procedure then actually

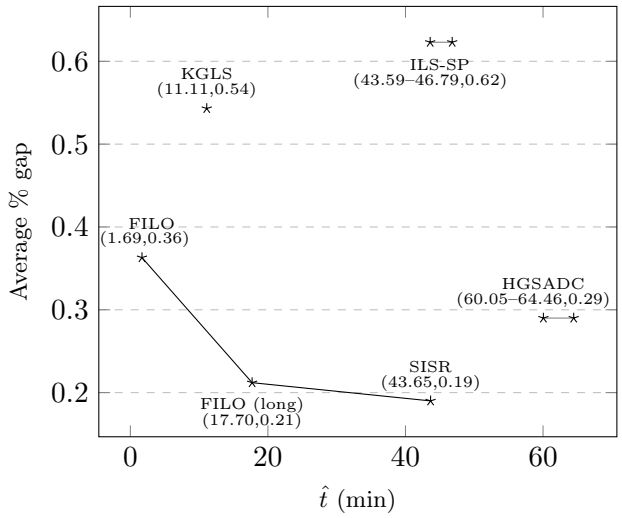


Figure 5 **State-of-the-art CVRP algorithms performance comparison over the $\mathbb{X}$ instances of Uchoa et al. (2017). A tuple describing the computing time and the average percentage gap is reported below each algorithm name. For ILS-SP and HGSADC we report a range of times since the exact CPU model is not specified.**

performs some fruitful iterations in large instances before reaching better-performing values for the shaking parameters $\omega_i, i \in V_c$.

We set the number n_{EC} of EJCH sequences explored from every move generator, to the minimum value that could provide reasonably good improvements with a much more limited computing time variability compared to larger values; see Section 4.2.

Finally, the shaking factors I_{LB} and I_{UB} can have a major impact on the overall algorithm execution. In fact, assigning them large values allows the guiding meta-strategy to increment the ω_i values, inducing a stronger shaking effect that will eventually require a longer local search re-optimization. On the other hand, values that are too low do not disrupt the current solution sufficiently to allow the re-optimization the possibility of performing some improvements. The identified values, coming from the limited random search analysis (see Bergstra and Bengio (2012)) described in Section 4.3, define an associated shaking which is neither too disruptive nor too gentle, resulting once again in a compromise between computing time and final solution quality.

3.3. Testing on $\mathbb{X}$ Instances

The $\mathbb{X}$ instances introduced in Uchoa et al. (2017) are the current standard benchmark for the CVRP. They consist of a hundred small- to large-sized instances, containing up to one thousand customers and covering a wide range of demand distributions and customer layouts. The performance of FILO is compared with current state-of-the-art algorithms on this dataset.

- The *iterated local search matheuristic* (ILS-SP), proposed by Subramanian, Uchoa, and Ochi (2013), consists of an ILS interacting with a mixed integer programming (MIP) solver. The

MIP solver is used to define new solutions as a combination of routes belonging to previously found local optima, through the time-limited solution of a set partitioning model.

- The *Hybrid Genetic Search with Adaptive Diversity Control* (HGSADC), proposed by Vidal et al. (2012), is a population-based method with an advanced and continuous diversification procedure.
- The *Knowledge-Guided Local Search* (KGLS), proposed by Arnold and Sörensen (2019), is a GLS metaheuristic enhanced by knowledge extracted from previous data mining analyses.
- Finally, the *Slack Induction by String Removal* (SISR) recently introduced by Christiaens and Vanden Berghe (2020), is a sophisticated, yet easily reproducible, ruin-and-recreate-based approach combined with a simulated annealing metaheuristic.

ILS-SP, HGSADC, and SISR are general methods able to solve a broad class of VRP variants, while KGLS also supports the Multi-Depot VRP and the Multi-Trip VRP. ILS-SP, HGSADC, and KGLS are local search-based methods, whereas SISR performs its improvement action through a ruin-and-recreate approach. Finally, KGLS defines a time-based termination condition of three minutes every 100 customers, while all the other methods fix a maximum number of iterations.

Figure 5 provides a graphic comparison of the algorithms' performances in terms of efficacy and efficiency by reporting the average behavior over 50 runs for ILS-SP, HGSADC, SISR, and FILO, and over a single run for KGLS, since it is a deterministic algorithm. The best known solution values (BKS) used to compute gaps are taken, at the time of writing, from CVRPLIB (2020). FILO compares favorably with the best existing algorithms, achieving an excellent compromise between solution quality and computing time. In fact, FILO finds average solutions that are significantly better than those of ILS-SP and KGLS, and similar to those found by HGSADC. However, SISR outperforms FILO. FILO (long), on the other hand, finds average solutions that are significantly better than those of ILS-SP, HGSADC, and KGLS and similar to those found by SISR. See Section C of the Appendix for full details.

As mentioned in Section 3.1, the average computing time for a single run $\hat{t}$ has been roughly normalized to match our CPU score by using the single-thread rating of PassMark® Software (2020), which assigns a score in the range of 1389 – 1491 to the compatible Intel Xeon CPUs used by ILS-SP and HGSADC (for which the precise model is not specified in Uchoa et al. (2017)), a score of 2052 to the AMD Ryzen 3 1300X CPU used by KGLS, and a score of 1662 to the Intel Xeon E5-2650 v2 CPU used by SISR.

Table 2 provides aggregate computations, grouped by instance size. The table highlights the scalability properties of FILO, by showing that the computing time for the largest instances is very similar to that obtained for smaller ones, yet the solution quality remains comparable to that achieved by other state-of-the-art algorithms. In fact, the computing time of FILO is more related

Table 2 Aggregate computations on 𝕏 instances.

		ILS-SP		HGSADC		KGLS		SISR		FILO		FILO (long)	
Size	Vertices	Avg	$\hat{t}$[1]	Avg	$\hat{t}$[1]	Avg	$\hat{t}$	Avg	$\hat{t}$	Avg	t	Avg	t
S	101 – 247	0.31	1.52	0.07	3.79	0.28	4.66	0.11	3.78	0.18	1.69	0.09	18.06
M	251 – 491	0.59	14.57	0.30	19.09	0.64	9.40	0.25	19.64	0.40	1.66	0.25	17.51
L	502 – 1001	0.98	123.32	0.50	169.28	0.69	19.47	0.21	110.54	0.50	1.72	0.30	17.56

[1] obtained by averaging normalized times associated with the fastest and the slowest compatible CPUs.

to the number of core optimization iterations rather than to the instance size. Moreover, when the computing times of FILO and FILO (long), which differ by the number of core optimization iterations by a factor of ten, are compared, the increase in computing time is proportional to the increase in the number of iterations. More experiments can be found in Section 4.6; for the largest instances, we studied the effects of further increasing the number of core optimization iterations which resulted in lowering the average gap to 0.19%.

3.4. Testing on Very Large-Scale Instances

Having assessed the performance of FILO on the standard 𝕏 instances, we now examine the real target of the proposed approach: very large-scale instances. To this end, we tested FILO on three challenging datasets containing instances with several thousands of customers.

The 𝔹 instances proposed by Arnold, Gendreau, and Sörensen (2019) are a set of ten very large-scale instances containing up to thirty thousand customers and reflecting real-world parcel distribution problems in Belgium. They include a first scenario, in which the depot is located centrally with respect to the customers and relatively short routes are performed, and a second one in which the depot is eccentric with respect to the service zone, thus much longer routes are required to visit the customers. We mainly compared FILO with KGLS$^{\text{XXL}}$ proposed in Arnold, Gendreau, and Sörensen (2019), an adaptation for very large-scale instances of the KGLS algorithm introduced in Section 3.3. Like KGLS, KGLS$^{\text{XXL}}$ also defines a time-based termination condition of either three or twelve minutes every 1000 customers for the short or the long version, respectively. KGLS$^{\text{XXL}}$ was executed on the same hardware configuration described for the 𝕏 dataset. Moreover, for the sake of completeness, we include results obtained by the LKH-3 algorithm proposed by Helsgaun (2017) in very long computing sessions (up to several days). The best known solution values (BKS) were taken, at the time of writing, from CVRPLIB (2020) and used to compute gaps. We note that for many of those BKS, no citable publication is available; the results are typically the outcome of very long runs. In other cases, as reported us by the authors (see Cavaliere, Bendotti, and Fischetti (2020) and Uchoa (2020)), the methods were initialized with previously-known best solutions. Therefore, their achievements cannot be compared with monolithic approaches such as the one we propose. As can be seen from Table 3, FILO is able to successfully find very good quality solutions

Table 3 Computations on $\mathbb{B}$ instances.

		KGLS$^{\text{XXL}}$		KGLS$^{\text{XXL}}$(long)		LKH-3	FILO				FILO (long)			
ID ($\lvert V_c\rvert$)	BKS	Gap	$\hat{t}$	Gap	$\hat{t}$	Gap	Best	Avg	Worst	t	Best	Avg	Worst	t
L1 (3000)	193092	0.74	13.47	0.71	53.88	0.67	0.26	0.38	0.50	2.13	-0.02	0.12	0.20	21.50
L2 (4000)	111810	4.18	17.96	2.69	71.84	1.50	0.43	0.62	0.86	3.28	-0.13	0.07	0.29	36.20
A1 (6000)	478091	0.83	26.94	0.73	107.76	0.68	0.35	0.43	0.55	2.76	0.04	0.10	0.17	28.00
A2 (7000)	292597	2.62	31.43	1.18	125.72	1.67	0.53	0.68	0.87	3.11	-0.10	0.00	0.14	33.66
G1 (10000)	470329	0.86	44.90	0.69	179.61	0.82	0.51	0.59	0.67	3.63	0.08	0.14	0.19	36.57
G2 (11000)	259712	2.94	49.39	1.85	197.57	2.33	0.50	0.72	1.06	4.62	-0.39	-0.31	-0.20	59.34
B1 (15000)	503407	1.35	67.35	0.73	269.41	1.20	0.66	0.75	0.82	4.67	0.03	0.09	0.14	47.83
B2 (16000)	349602	3.49	71.84	1.77	287.37	2.23	0.58	0.80	1.08	5.34	-0.72	-0.60	-0.45	62.70
F1 (20000)	7256529	0.97	89.80	0.54	359.21	0.61	0.52	0.56	0.62	7.22	0.08	0.12	0.18	78.44
F2 (30000)	4405678	3.65	134.70	2.24	538.82	2.13	1.21	1.40	1.57	10.99	-0.12	-0.02	0.12	150.93
Mean		2.16	54.78	1.31	219.12	1.38	0.56	0.69	0.86	4.78	-0.12	-0.03	0.08	55.52

New best solutions: (L1, 193052);(L2, 111661);(A2, 292303);(G2, 258700);(B2, 347092);(F2, 4400188).

in a shorter computing time than KGLS$^{\text{XXL}}$. In fact, the average gap of FILO is almost half that of KGLS$^{\text{XXL}}$ (long) in just about five minutes of computing time. Furthermore, FILO (long) is able to find several new BKS in less than three hours of computing time, and the computing times remain consistent across instances with different structures. We again note the scalability of FILO by observing that, to solve an instance with ten times more customers, the computing time only increases approximately fivefold.

The $\mathbb{K}$ dataset proposed by Kytöjoki et al. (2007) contains eight very large-scale instances with up to twelve thousand customers. The first four instances (W, E, S, and M) are derived from real-life waste collection problems in Finland, while the remaining instances contain customers randomly and uniformly distributed. Again, we mainly compared FILO with KGLS$^{\text{XXL}}$ proposed in Arnold, Gendreau, and Sörensen (2019) but we also include results of the GVNS algorithm introduced in Kytöjoki et al. (2007) (the first method used to solve the $\mathbb{K}$ instances). GVNS was run on an AMD Athlon 64 3000+ having a single thread score of 554 and KGLS$^{\text{XXL}}$ was run on the same hardware configuration as for the $\mathbb{B}$ dataset. As can be seen from Table 4, FILO is able to successfully find very good-quality solutions in a relatively short computing time compared to KGLS$^{\text{XXL}}$. Importantly, both FILO and FILO (long) find new best solutions for all instances. The computing time associated with random instances follows the same trend seen for the $\mathbb{B}$ instances. On the other hand, instances derived from real-life problems require a much longer computing time. By analyzing the structure of the final solutions obtained by FILO (long), we note that they are composed of few very long routes with several hundred customers. Moreover, the shaking intensity at the end of a run $\bar{\omega} = \sum_{i \in V_c} \omega_i / |V_c|$, averaged over the W, E S and M instances, has a value four times larger than that associated with the $\mathbb{B}$ dataset (91.37 ± 28.86 and 22.42 ± 4.40, respectively). The reason for such a high average value may be related to the shaking procedure. In particular, given the very low number of routes for those instances (on average 15.00 ± 1.83), the shaking procedure might choose to jump

Table 4 Computations on $\mathbb{K}$ instances.

		GVNS		KGLS$^{\text{XXL}}$		KGLS$^{\text{XXL}}$ (long)		FILO				FILO (long)			
ID ($\lvert V_c\rvert$)	BKS*	Gap	$\hat{t}$	Gap	$\hat{t}$	Gap	$\hat{t}$	Best	Avg	Worst	t	Best	Mean	Worst	t
W (7798)	4481423	1.75	8.36	0.39	7.72	0.00	35.02	-7.35	-5.71	-3.31	9.59	-8.47	-7.74	-6.23	145.23
E (9516)	4507948	5.54	20.34	0.33	18.86	0.00	42.66	-2.87	-2.01	-0.58	14.10	-4.06	-3.41	-2.49	248.11
S (8454)	3189850	4.51	13.63	0.46	12.66	0.00	38.17	-5.04	-3.85	-1.90	9.30	-6.07	-5.48	-4.07	146.82
M (10217)	3071090	3.25	18.81	0.78	17.42	0.00	45.80	-2.54	-1.82	0.44	15.86	-3.46	-3.13	-2.74	263.56
R3 (3000)	182206	2.20	1.16	0.50	1.08	0.00	13.47	-0.50	-0.39	-0.25	2.25	-0.88	-0.81	-0.70	21.09
R6 (6000)	347224	1.58	5.92	0.19	5.48	0.00	26.94	-0.35	-0.27	-0.19	2.97	-0.81	-0.74	-0.68	27.70
R9 (9000)	511378	1.19	13.99	0.05	12.93	0.00	40.41	-0.25	-0.17	-0.09	3.70	-0.70	-0.66	-0.61	33.48
R12 (12000)	672456	1.25	26.28	0.06	24.34	0.00	53.88	-0.09	-0.01	0.07	4.47	-0.60	-0.54	-0.50	39.76
Mean		2.66	13.56	0.35	12.56	0.00	37.04	-2.37	-1.78	-0.73	7.78	-3.13	-2.81	-2.25	115.72

* taken from Arnold, Gendreau, and Sörensen (2019).
New best solutions: (W, 4101686.00);(E, 4324802.50);(S, 2996254.00);(M, 2964867.25);(R3, 180597.91);(R6, 344407.00);(R9, 507787.66);(R12, 668435.00).

to a not-yet-visited neighbor route that is not available. In such cases, the ruin is prematurely aborted, possibly causing a mismatch between the actual shaking intensity and the required one identified by $\omega_i, i \in V_c$ values. The mismatch may cause the average shaking intensity to increase to an abnormally large value. In many cases, the ruin activity is unaffected, because it will be prematurely aborted. However, if the early stop comes after several customers have already been removed, the average ruin activity will be stronger than required, causing a more time-consuming re-optimization.

Finally, the $\mathbb{Z}$ dataset was proposed in Zachariadis and Kiranoudis (2010). The dataset contains four large-scale instances, representing the actual distribution of customers' locations within Greek cities. All instances have three thousand customers whose demand is uniformly distributed in 1–100. The vehicle capacity is set to 1000. We compared FILO with the *Penalized Static Move Descriptors Algorithm* (PSMDA) described in Zachariadis and Kiranoudis (2010), consisting of a Tabu Search metaheuristic in which the local search is executed by means of SMDs considering compound operators and a neighborhood pruning technique similar to that of GNs. The algorithm was run for a prefixed amount of time on an Intel Core2 Duo T5500 CPU with a single-thread score of 573. As can be seen from Table 5, FILO successfully solved the $\mathbb{Z}$ dataset, finding new best solutions for all four

Table 5 Computations on $\mathbb{Z}$ instances.

		PSMDA			FILO				FILO (long)			
ID ($\lvert V_c\rvert$)	BKS	Best	Avg	$\hat{t}^1$	Best	Mean	Worst	t	Best	Avg	Worst	t
ZK1 (3000)	13666.36	0.00	0.94	60.18	-1.45	-1.31	-1.14	2.50	-1.81	-1.71	-1.54	22.68
ZK2 (3000)	3536.25	0.00	1.32	60.18	-2.14	-1.97	-1.76	2.30	-2.68	-2.55	-2.36	20.81
ZK3 (3000)	1170.33	0.00	1.55	60.18	-2.67	-2.53	-2.39	1.97	-3.24	-3.09	-2.96	18.60
ZK4 (3000)	1139.08	0.00	1.32	60.18	-2.23	-2.08	-1.91	1.86	-2.76	-2.65	-2.55	16.84
Mean		0.00	1.28	60.18	-2.12	-1.97	-1.80	2.16	-2.62	-2.50	-2.35	19.73

[1] max (normalized) time per run.
New best solutions: (ZK1, 13419.44);(ZK2, 3441.54);(ZK3, 1132.47);(ZK4, 1107.62).

instances in a very short computing time, which is comparable to that associated with instances of similar size of the $\mathbb{B}$ and $\mathbb{K}$ datasets.

To summarize, FILO and FILO (long) obtain, in short computing time, average solutions that are generally significantly better than those found by the competing algorithms. We refer to Section D of the Appendix for the statistical significance of the above reported results.

4. Algorithmic Components Analysis

The design of FILO followed an iterative process, whose core decisions were driven by the analyses detailed in this section. We review (i) the definition of the initial solution by means of the construction phase, possibly followed by the route minimization procedure; (ii) the acceleration and pruning techniques employed by the local search engine; and (iii) the guiding of the shaking strategy. As for the parameter tuning, if not stated otherwise, the analyses reported here refer to the set of large-size $\mathbb{X}$ instances with more than five hundred vertices. Moreover, when analyzing components involving some randomization, we performed ten runs by setting the pseudorandom engine seed equal to the run counter minus one and reported aggregated results averaged over seeds and instances. Finally, we refer to Section A of the Appendix for additional material.

4.1. Initial Solution Definition

The construction phase depends on the parameter n_{cw} to identify, for each customer $i \in V_c$, the number of neighbors $j \in \mathcal{N}_i^{n_{cw}}(\{j \in V_c : i < j\})\}$ involved in the savings computation. Figure 6 (left) shows the variation of the solution quality (QUALITY GAP) and compactness (ROUTE GAP) when varying n_{cw}. We focused on the subset of instances, listed in Table 6, for which computing an initial solution of good quality is difficult (i.e., instances for which initial solutions have a large gap and use more routes than suggested by the heuristic estimate). Our findings validate what was already proposed in Arnold, Gendreau, and Sörensen (2019). A value of n_{cw} around 100 provides initial solutions of quality comparable to that of larger values, but in slightly shorter computing times (the differences are, however, in the order of a few tens of milliseconds for the largest n_{cw} values we considered). As in Section 3.1, the QUALITY GAP is defined as $100 \cdot (\text{COST}(S) - BKS)/BKS$, where S is the solution resulting from the procedure and, similarly, ROUTE GAP is defined as $100 \cdot (|S| - k)/k$, where k is the heuristically found ideal estimated number of routes described in Section 2.3.1.

Not surprisingly, using larger n_{cw} values is not sufficient to increase solution compactness. Indeed, the route minimization procedure tackles this strategic aspect of the CVRP, which is more concerned with the assignment of customers rather than with their routing. Note, however, that, contrarily to most existing route minimization procedures, the proposed one is still quality-oriented. In fact, a solution with a better objective function is always preferred over a solution with a lower number

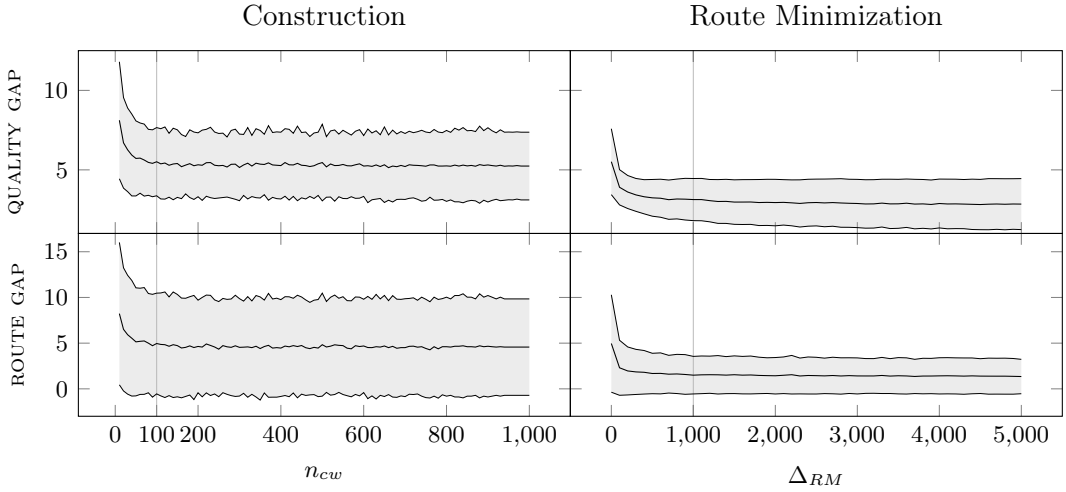


Figure 6 **Tuning of the n_{cw} and Δ_{RM} parameters based on instances listed in Table 6 for which initial solutions have a large gap and use more route than suggested by the heuristic estimate. For each diagram, the middle line represents the average and the grayed area identifies the standard deviation. Selected values for n_{cw} and Δ_{RM} are marked with a vertical line.**

of routes, but the procedure structure is more specifically aimed at reducing the number of routes while also often obtaining the desirable effect of improving the objective function. Figure 6 (right) shows the results of this procedure when it is applied, for different numbers of iterations Δ_{RM}, to a solution S built by the construction phase with $n_{cw} = 100$. The diagrams highlight the procedure's effectiveness, both in improving low-quality initial solutions and in quickly compacting them by (often significantly) reducing the number of routes. As a result, we selected $n_{cw} = 100$ and $\Delta_{RM} = 1000$. The average computing time for largest values of Δ_{RM} is approximately ten seconds.

In addition, Table 6 compares the final algorithm outcome when the route minimization procedure is disabled, i.e., $\Delta_{RM} = 0$. Despite not always being crucial for the final solution quality because of the complex interactions among all the algorithm's components, the route minimization procedure provides substantial improvement for those instances containing several customers with small demand and a few customers with relatively large demand, such as X-n670-k130 and X-n936-k151. We can conclude that, in average, the route minimization positively affects the final algorithm's outcome without any significant impact on the computing time.

4.2. Local Search

We analyzed the local search operators described in Section 2.2 in the context of the core optimization procedure, where all features we propose are fully employed. The effect of a local search operator application is tightly linked to the state of the algorithm in that specific instant. In our approach, randomization plays a major role in selecting the area that, once disrupted by the shaking

Table 6 Computations with and without route minimization (RM) procedure.

	FILO without RM ($\Delta_{RM}=0, \Delta_{CO}=10^5$)				FILO with RM ($\Delta_{RM}=10^3, \Delta_{CO}=10^5$)			
ID[1]	Best	Avg	Worst	t	Best	Avg	Worst	t
X-n524-k153	0.29	0.57	0.81	1.37	0.08	0.39	0.68	1.34
X-n536-k96	0.69	0.80	0.89	1.51	0.71	0.80	0.87	1.60
X-n586-k159	0.46	0.65	0.79	1.73	0.57	0.73	0.87	1.65
X-n599-k92	0.37	0.47	0.60	1.54	0.37	0.47	0.69	1.57
X-n613-k62	0.51	0.68	0.84	1.12	0.39	0.66	0.96	1.16
X-n670-k130	1.32	1.95	2.38	1.36	0.83	1.08	1.32	1.41
X-n685-k75	0.35	0.55	0.67	1.34	0.47	0.63	0.82	1.42
X-n733-k159	0.32	0.38	0.45	1.25	0.25	0.34	0.45	1.25
X-n749-k98	0.57	0.75	0.88	1.40	0.54	0.68	0.85	1.46
X-n766-k71	0.59	0.73	0.93	1.59	0.46	0.59	0.66	1.60
X-n783-k48	0.52	0.60	0.72	1.74	0.34	0.62	0.87	1.75
X-n819-k171	0.60	0.76	0.93	1.37	0.83	0.90	1.03	1.43
X-n936-k151	0.90	1.26	1.52	1.29	0.39	0.83	1.23	1.31
X-n979-k58	0.27	0.36	0.48	2.24	0.26	0.35	0.44	2.37
Mean	0.55	0.75	0.92	1.49	0.47	0.65	0.84	1.52

[1] for which the route minimization procedure is executed.

procedure, is re-optimized, and the evolution of the algorithm affects shaking intensity and sparsification factors. We thus believe that the evaluation of a local search operator should not occur "in a vacuum", but needs to be performed within the algorithm execution. Therefore, we studied the effectiveness of individual operators by sampling the algorithm state throughout the core optimization phase. More specifically, a sample consists of a shaken solution S and its subset of cached vertices $\bar{V}_S$, the shaking vector $\boldsymbol{\omega}$, and the sparsification vector $\boldsymbol{\gamma}$. Each sample, derived from the actual algorithm execution, is a relevant snapshot describing the algorithm evolution. Moreover, according to when the sample is taken, it could describe initial or final algorithm states associated with lower or higher quality solutions.

We tested every local search operator on each sample for a total number of $\Delta_{CO}=10^5$ core optimization iterations. In addition, we considered seven variants for the EJCH operator, named EJCH(n_{EC}), where n_{EC} defines the maximum number of sequences explored from each move generator, when searching for a feasible sequence of relocations. For each local search operator (applied to a shaken solution), we analyzed the gap improvement when successfully applied, the application time, and the success ratio computed as the number of improving applications over the total number of attempts. Full details are available in Section A.2 of the Appendix.

As expected, EJCH($\cdot$) are the most effective, yet time-consuming, operators. Their success ratio also suggest that in most shaken solutions, finding an improving ejection-chain is relatively easy. However, when this is not the case, the computing time can be very large especially for the EJCH with the highest n_{EC}. We also noted that simpler operators, such as 10EX, 11EX, TAILS, and SPLIT have a large success ratio, meaning that they are more likely to be applied, and provide larger gap

improvements compared to all other operators with quadratic cardinality. These differences may occur because their feasibility requirements are more easily met, and thus they are more frequently applied. On the other hand, all the remaining operators, despite having a lower success ratio, still do allow better final solutions than when they are disabled. Surprisingly, TWOPT is seldom useful, meaning that the shaking procedure typically generates routes that are almost two-optimal. However, we kept it because of its very short application time.

When structuring the HRVND, we grouped all the operators with a comparable application time in the first tier (i.e., all the operators but EJCH$(\cdot)$).

Selecting which EJCH to include was guided by an additional analysis comparing the results obtained by the EJCH(n_{EC}), for the different values of n_{EC}, when applied to solutions that are already a local optimum for the first HRVND tier. We observed that by applying the EJCH$(\cdot)$ operator on first tier local optima, the application time (as well as its variability) is dramatically reduced to a magnitude similar to that of other simpler operators. Moreover, the success ratio, the gap improvement, and experiments with an HRVND without EJCH$(\cdot)$ all suggest that its application may be beneficial to obtain high-quality final solutions. In our implementation, we selected EJCH(25) (shortened to EJCH in the rest of the paper), because it is more compatible with the scalability objectives of our approach while still retaining its effectiveness compared to EJCH$(\cdot)$ with greater n_{EC}.

Finally, as we apply the operators of each tier in an RVND fashion, we may expect a lower success ratio and less gap improvement, as well as a shorter application time, when they are applied on solutions that are already local optima for a number of other operators of the same tier.

We can now study how each operator contributes to the total improvement of the defined HRVND structure in more detail. Denoting with $\mathbb{O}$ the set of local search operators we employ, we stored the total gap improvement $D(O)$ achieved in a number of $I(O)$ successful applications for each operator $O \in \mathbb{O}$. Note that a single application consists of a full neighborhood exploration. The ratio $R(O) = D(O)/I(O)$ thus identifies the expected improvement that a successful exploration of O would produce on an average solution. We can compute the percentage *Relative Neighborhood Improvement* index of O with respect to the set of the available operators $\mathbb{O}$ as RNI$(O, \mathbb{O}) = 100 \cdot R(O)/\sum_{O' \in \mathbb{O}} R(O')$, which is shown in Figure 7. As shown in the figure, all the operators positively contribute to the overall improvement process.

4.3. Shaking Guiding Strategy

The structure-aware and quality-oriented strategy employed by the core optimization procedure to guide the shaking intensity uses two factors to determine whether to increment, reduce, or randomly change the parameters $\omega_i, i \in V_c$ of customers involved in a shaking application. More precisely,

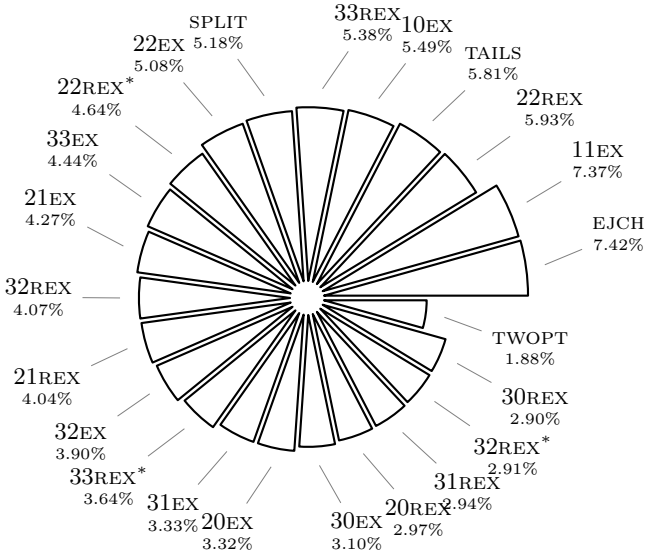


Figure 7 **Percentage relative neighborhood improvement index for each local search operator in the engine. The index summarizes the contribution of each operator to the total improvement.**

I_{LB} determines when shaking parameters are increased, while I_{UB} defines when they are decreased. Together, they identify the range in which the guiding strategy actively operates. To determine reasonably effective values for I_{LB} and I_{UB}, we performed a limited random search, testing a hundred unique combinations for $I_{LB} \in [0.2, 0.4]$ discretized with steps of 0.025, and $I_{UB} \in [0.5, 1.5]$ discretized with steps of 0.05. A graphic representation for the different configurations and their performances is depicted in Figure 8. The proposed ranges are the outcome of an iterative process in which larger ones were narrowed down to focus on combinations producing good-quality solutions in a short computing time. As shown in Figure 8, the values in the selected ranges have a minor impact on the solution quality and a slightly greater one on the computing time.

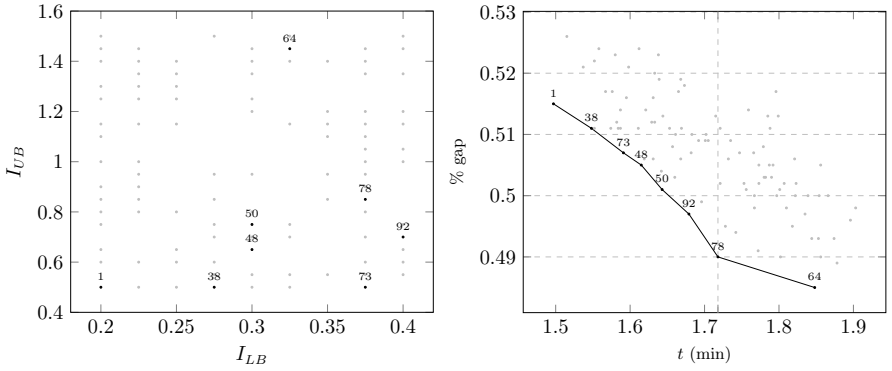


Figure 8 **Tuning of I_{LB} and I_{UB}. In the left diagram, the configurations we considered in the random search. In the right, the performance associated with each configuration obtained by running FILO with the ones we plot on the left diagram. Only the best configurations are numbered.**

In our implementation, we selected configuration 78 ($I_{LB} = 0.375$ and $I_{UB} = 0.85$). We noticed that slightly better solutions can be obtained when using moderately larger values particularly for I_{UB}. However, as shaking parameters reach larger values, we expect the associated re-optimization time to increase and selecting values that are too large may also result in poor quality final solutions. Finally, once the factors, which are used in combination with the average cost of a solution arc (see Section 2.3.2.1), are set to reasonable values, the procedure can generalize well to solutions and instances with a very different structure, as shown by our computational results.

4.4. Move Generators and Granular Neighborhoods

The benefits of GNs are well documented in several papers, such as Toth and Vigo (2003) and Schneider, Schwahn, and Vigo (2017). However, careful tuning is necessary to get the best out of GNs. In fact, different values for the number of neighbors considered in the sparsification rule n_{gs}, the base sparsification factor γ_{base}, and the reduction factor δ affecting the number of nonimproving iterations before increasing a sparsification factor, can dramatically alter the performance of the algorithm and, more specifically, its computing time. The sparsification increment factor λ may also play a role; however, we fixed it to $\lambda = 2$, as in the original GN definition, and made the other parameters depend on its value.

As for the shaking parameters, we studied the effect of varying n_{gs}, γ_{base} and δ by performing a limited random search among reasonable ranges of values. The selected configurations and the associated performances can be seen in Figure 9. In particular, we generated a hundred unique combinations for $n_{gs} \in \{25, 50, 75, 100\}$, $\gamma_{base} \in [0.1, 0.5]$ discretized with steps of 0.025, and $\delta \in [0.1, 1]$ discretized with steps of 0.05. As expected, a larger value for n_{gs} is generally associated with a better final solution quality. However, the associated computing time increment provides only an extremely limited gap improvement.

In our implementation, we selected configuration 40 ($n_{gs} = 25$, $\gamma_{base} = 0.25$, and $\delta = 0.5$) because it allows solutions of very good quality in a relatively short computing time. We classified those Pareto optimal configurations producing an average gap lower than or equal to 0.55% within a computing time not longer than 5 minutes as high-performing, and the remaining as low-performing; thus we obtained a dataset of 14 high-performing and 16 low-performing configurations. We gained some insights about the granular-related characteristics of these configurations by analyzing a simple J48 decision tree trained with WEKA 3.8 (Frank, Hall, and Witten (2016)) on the above dataset. A configuration can be classified as high-performing with an accuracy of about 97% if $n_{gs} \leq 50 \wedge 0.12 < \gamma_{base} \leq 0.35$ or $n_{gs} > 50 \wedge \gamma_{base} \leq 0.15$. Apparently, δ does not provide useful insights for classifying configuration performances. These rules suggest that a very aggressive sparsification is preferable for obtaining reasonably good results in short computing times. The reason may be related to the

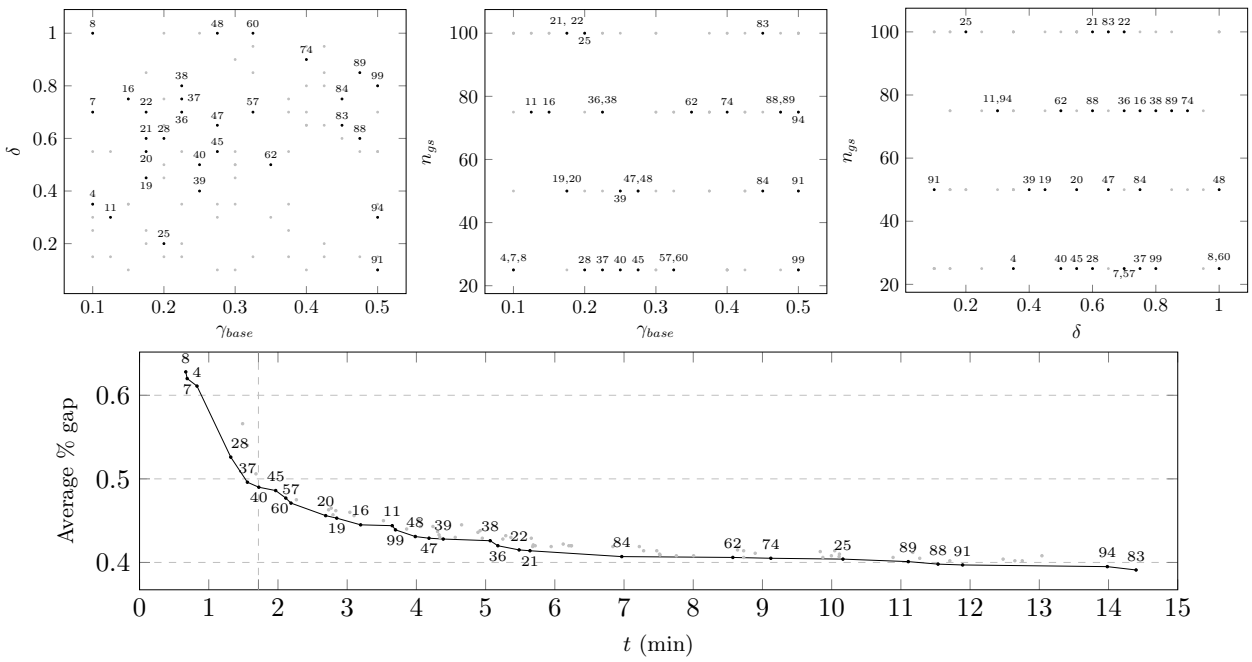


Figure 9 **Tuning of n_{gs}, γ_{base}, and δ. In the top row, the configurations we considered in the random search. In the bottom row, the performance associated with each configuration obtained by running FILO with the configuration values. Only the best configurations are numbered; seven suboptimal ones, with computing times greater that 15 minutes, are omitted.**

neighbor acceptance strategy used in FILO that is based on a simulated annealing rule. In fact, the current search trajectory may be far worse than the current best solution, thus making additional core optimization iterations more convenient than very accurate neighborhood explorations.

As suggested in several papers, we then studied the effect of including arcs incident into the depot in the sparsified set.We considered the set $T_0 = \cup_{i \in V_c}\{(i,0)(0,i)\}$ and defined $T' = T \cup T_0$, where T is the set of move generators defined in Section 2.2.2. Moreover, we defined the dynamic set of move generators $T_0^{\gamma_0} = \{(i,0),(0,i) : i \in \mathcal{N}^k(V_c)\} \subseteq T_0$ with $k = \lfloor \gamma_0 \cdot |V_c| \rfloor$. The new complete set of dynamic move generators is thus $T'^{\gamma} = T^{\gamma} \cup T_0^{\gamma_0}$. Note that, by filtering each set separately, we avoid the possibility that one set completely overshadows another in case it considers shorter arcs. The average gap obtained by running FILO with this new set of move generators was 0.51%, compared to the 0.49% obtained with move generators defined by the rule of Section 2.2.2. Moreover, the computing time was approximately 115% larger (i.e., 3.70 minutes compared to 1.72). In light of this result, including all arcs incident into the depot may not be appropriate when moving to very large-scale instances.

Finally, we compared the vertex-wise management of move generators with the more standard one, in which after a number of $\delta \cdot \Delta_{CO}$ nonimproving iterations the total number of active move generators is doubled; i.e., $\gamma_i = \min\{\gamma_i \cdot \lambda, 1\}, i \in V$. When a solution improving the current best

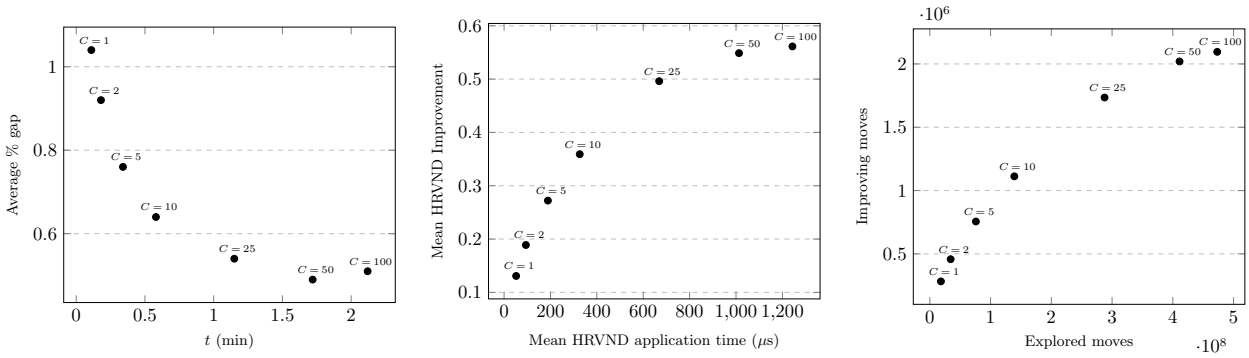


Figure 10 **Tuning of C. Computational results (left) and local search statistics (center, right) obtained by running FILO with the associated C value.**

solution is identified, all sparsification factors are reset; i.e., $\gamma_i = \gamma_{base}, i \in V$. By running FILO with move generators managed in the standard way we obtained a gap of 0.51% compared to a gap of 0.49% with the vertex-wise management, and computing times were similar. However, we believe both strategies to be equally effective when properly tuned. To conclude, the proposed vertex-wise management of move generators might be a reasonable and effective alternative generalization of the standard one and better fits the localized optimization design of FILO.

4.5. Selective Vertex Caching

The dimension of the cache C may considerably affect the overall algorithm outcome by indirectly acting on the different components employed: a smaller C value will promote a milder shaking intensity. In fact, most of the customers involved in stronger shakings would not be considered by subsequent local search application because they were not cached, due to the limit imposed by C. Furthermore, a low C value would cause the local search to perform fewer improvements per iteration, reducing the likelihood of improving the best solution S^*. As a result, sparsification factors $\gamma_i, i \in V$ might reach larger values compared to scenarios using a larger C. Figure 10 illustrates the average performance of FILO and statistics related to the local search execution as C is varied. In our implementation, we selected $C = 50$ because it produces solutions of a quality comparable to larger values but with shorter computing times. Note, however, that the SVC and shaking guiding strategy are highly interconnected components. In fact, even when $C >> 100$, results and statistics remain comparable to those with $C \approx 100$, because of the limits on the number of ruined customers imposed by the shaking guiding strategy.

Figure 11 provides a number of hints about the scalability of FILO. In fact, we observed that the number of moves explored by the local search, as well as its average application time, does not depend on the instance size. There is, however, a positive correlation between the number of routes and the number of explored moves.

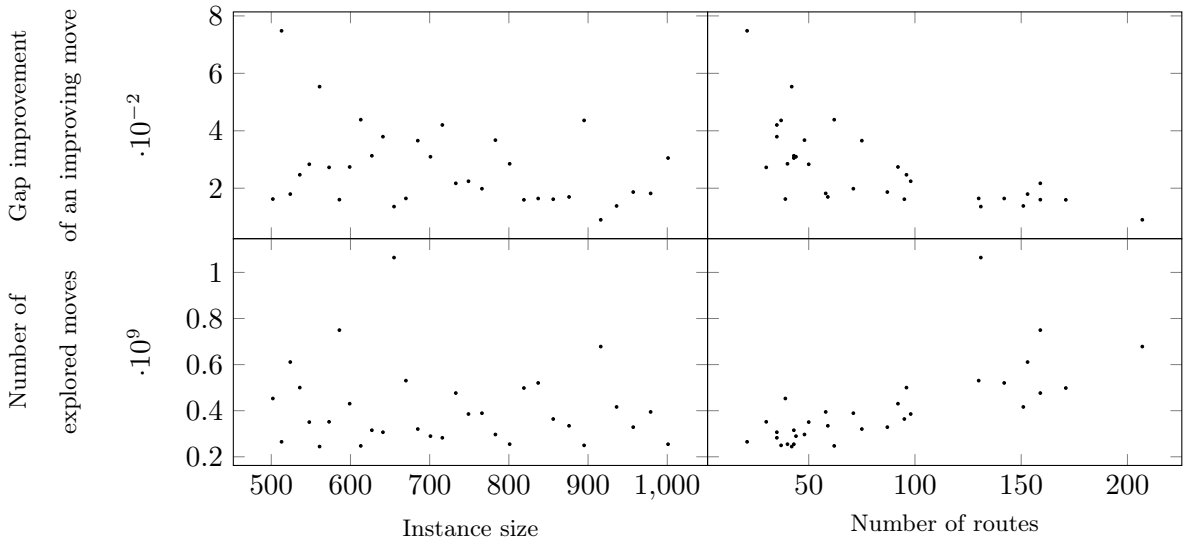


Figure 11 **Statistics for local search operators while varying the instance size and the nominal number of routes, i.e., defined in the $\mathbb{X}$ instance name.**

Finally, we tested FILO without the SVC: we set $C = |V|$ and included all vertices in the cache (never removing them). Recall that, normal cache behavior requires that it be emptied at the beginning of each improvement iteration. Several components make use of the cache to identify areas where it may be worth working. In this scenario, the update of the shaking parameters will try to identify a set of values that are globally good. The average results were comparable with those with $C = 50$ and the cache enabled: the average gap was 0.50%, but the computing time increased by a factor of ten, increasing to 20.15 minutes.

4.6. Extreme Runs

In this section we investigate whether allowing longer computations is enough to improve the quality of final solutions. This question is partially answered by the computational results of Section 3, where we considered the FILO (long) version. In addition we performed an even longer run by setting $\Delta_{CO} = 10^7$. The average gap for the subset of large instances of the $\mathbb{X}$ decreased from the 0.30% with FILO (long) to 0.19% in 183.74 minutes. Moreover, we found three new best solutions. The increment in the computing time remains quite constant: i.e., increasing the number of iterations by a factor of ten increases the computing time about ten times. Full details are given in Section A.3 of the Appendix.

Finally, we studied whether performing a larger number of runs per instance drastically changes the average result and computing time. To this end, we compared the results obtained on the large scale $\mathbb{X}$ instances in 50 runs (described in Section 3.3) with the results obtained by performing 100 runs. Seeds were selected as described in Section 3.1. The average gap (rounded to two decimal

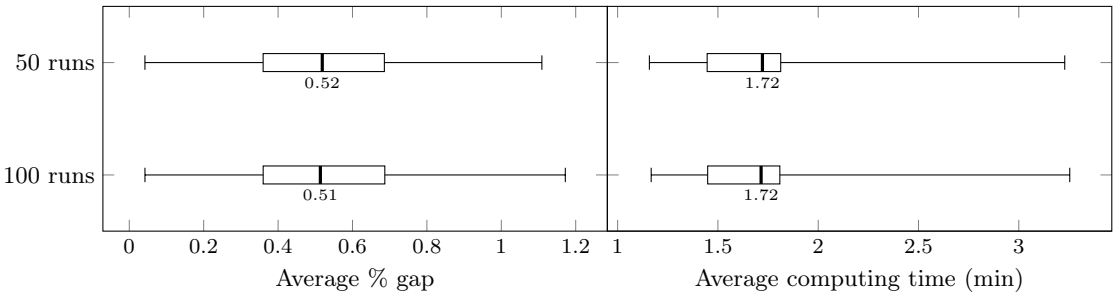


Figure 12 **Average gap and computing time comparison while varying the number of runs per instance. The median value is shown below each boxplot.**

places) when performing 100 runs increased to 0.51% and the computing time remained the same; see Figure 12.

To better assess whether there was a statistically significant difference among the averages between runs with seeds 0–49 and runs with seeds 0–99, we performed a Wilcoxon signed-rank test (Wilcoxon (1945)) using the R software (R Core Team (2020)). The null hypothesis states that the two samples of averages are identical; that is, they have the same median. Both the average gap and computing time do not statistically differ, whether performing 50 or 100 runs. In fact, assuming a confidence level $\alpha = 0.025$, the p-values are 0.977935 and 0.500047 for the average solution quality and computing time, respectively. In both cases, the p-value is greater than α and thus we cannot reject the null hypothesis that the samples are not statistically different.

4.7. Simplified Versions of FILO

In this section, we study the behavior of two simplified versions of FILO to better understand and assess the contribution of the main algorithmic components we developed when moving from large- to very large-scale instances. To this end, we concentrate our analysis on the $\mathbb{X}$ and $\mathbb{B}$ datasets.

The first version, called FILO1, complies with the initial design objectives of FILO: realizing an effective algorithm which exhibits an almost-linear computing time growth based on a localized and tailored local search optimization. More precisely, FILO1 contains the following changes with respect to FILO, which will make the implementation considerably easier:

- The HRVND is replaced by a RVND, containing only simple local search operators, identified as the most effective according to Figure 15 of Section A.2 of the Appendix. In particular, we selected 11EX, 10EX, TAILS and SPLIT because they have a high success rate (greater than 45%).
- The selected neighborhoods are explored according to a best-improvement strategy, without exploiting SMDs.
- The route minimization phase is never executed.

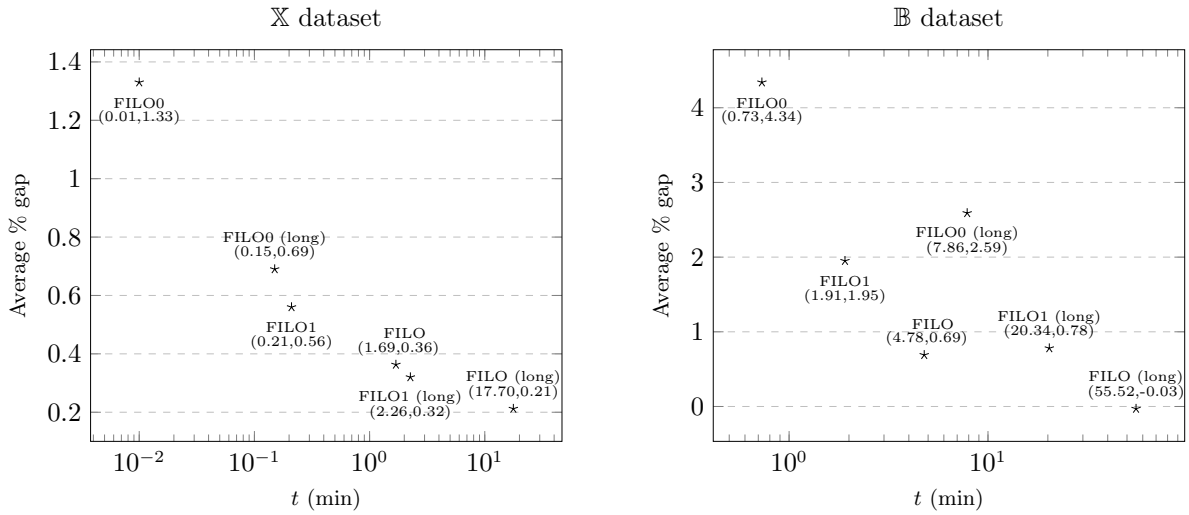


Figure 13 **Performance comparison of FILO variants on the 𝕏 (left) and 𝔹 (right) instances.**

Although their implementation may be easier because of the above changes, the following components remain the same as in FILO:

- The vertex-wise management of move generators is still used to provide a tailored intensification. However, its implementation becomes trivial when SMDs are not used.
- The SVC is still used to keep the optimization localized.
- The core optimization phase remains unchanged; e.g., the shaking procedure still contains the adaptive update of shaking parameters to perform a tailored optimization.

Note, however, that removing the SMDs means that the cache size becomes a hard constraint, and each neighborhood exploration will only consider those moves for which at least one of the vertices is cached.

The second version, called FILO0, further simplifies FILO1 by retaining only the optimization due to the random walk ruin-and-recreate, without the application of any local search - thus removing all components associated with the local search engine.

Each 𝕏 and 𝔹 instance was solved 50 times with seeds defined as in Section 3.1. Parameters for the preserved components have the same tuning as in Section 3.2. From the results (shown in Figure 13) we can summarize that:

- FILO0, inspired by the effectiveness of the work by Christiaens and Vanden Berghe (2020), confirms that an accurate design of a ruin-and-recreate procedure, coupled with an effective diversification strategy, may be enough to find good-quality solutions for medium to large instances in negligible computing time. However, results on very large-scale instances show that, given the same number of iterations, other strategies are necessary to improve the effectiveness of the method.

- FILO1, given enough iterations, proved to be competitive with FILO on the $\mathbb{X}$ dataset. However, it is not always Pareto-optimal on the largest $\mathbb{B}$ instances. The reduced number of local search operators thus has a significant impact for the proposed algorithm when moving to very large-scale instances. Finally, the SVC and GNs allow us to limit the computing time of FILO1 considerably, because the filtering of GNs is fairly aggressive with the proposed parametrizations. We strongly suggest, however, the adoption of SMDs in VND settings, which would allow considerable savings in the computational effort (see, Zachariadis and Kiranoudis (2010) and Beek et al. (2018)). In fact, the computing time of FILO1 is reduced in average of about 10% when the implementation exploits SMDs even if using few local search operators and very aggressive neighborhood filtering strategies.

5. Conclusions

In this paper, we presented FILO: an effective and scalable algorithm for the CVRP.

The proposed algorithm combines an efficient implementation of existing speedup techniques for local search engines together with several new algorithmic components whose role and impact are extensively analyzed. In particular, FILO performs its main improving action by re-optimizing a very limited area that was previously disrupted by a localized shaking application. The shaking is performed in a ruin-and-recreate fashion and guided by a meta-strategy that iteratively tailors the ruin intensity to the current instance and solution. A sophisticated implementation of a local search engine then re-optimizes the disrupted area by means of a number of interconnected components, both novel and revisited, which characterize the effectiveness and scalability of the proposed algorithm. More precisely:

- An innovative Selective Vertex Caching strategy is used to focus the optimization process on solution areas that were recently changed.
- Dynamic Granular Neighborhoods (GN), managed in a more general way than was originally proposed by Toth and Vigo (2003), are used to identify a number of promising neighbor solutions by intensifying the search only where it is more required.
- A considerable number of local search operators are implemented using the Static Move Descriptors (SMD), and structured into a Hierarchical Randomized Variable Neighborhood Descent to actually perform the improvements.
- A new adaptive shaking strategy is proposed to iteratively modulate the intensity of the shaking based on the quality and structure of instances and solutions.

Despite the exploration of several neighborhoods, the method is still very fast, thanks to an accurate design; it also greatly benefits from the above-mentioned acceleration and pruning techniques. Moreover, FILO exhibits a computing time that grows linearly with respect to the instance size,

making it very suitable for solving very large-scale instances without sacrificing the quality of the final solutions. In fact, it was able to find several new best solutions for very large-scale instances and two new best solutions for the well-studied $\mathbb{X}$ dataset proposed by Uchoa et al. (2017).

The effectiveness and potential of the new algorithmic components proposed in FILO are further assessed by analyzing two simplified versions, which maintain the main elements of FILO listed above, but are considerably easier to implement. The first one does not make use of SMDs to speed up the local search and removes some minor components, such as the route minimization step. In contrast, the second one, inspired by the effective algorithm of Christiaens and Vanden Berghe (2020), completely removes the local search engine and only performs a ruin-and-recreate, while adopting the diversification and intensification strategies of FILO. Overall, the simplified versions proved able to obtain very good solutions thanks to the contribution of the new components, particularly on medium and large instances. However, the combination of these new ideas with the faster execution provided by the combination of GNs and SMDs in the local search engine leads to an algorithm which is able to obtain much better solutions than the simplified versions, within the same amount of time. In addition, the computational requirement of FILO grows almost linearly with the instance size, so it is possible to efficiently solve very large-scale instances with much shorter computing times than the existing methods from the literature.

Acknowledgments

We gratefully thank Birger Raa for initial discussions on Static Move Descriptors and Granular Neighborhoods, Luca Ghedini for managing the cluster of machines we used for the experiments, and Florian Arnold for sharing the $\mathbb{K}$ dataset. This research was partially funded by the U.S. Air Force Office of Scientific Research [Award FA9550-17-1-0234]. Finally, we gratefully thank the Associate Editor and referees for their valuable comments.

Appendix A: Algorithmic Components Analysis: Additional Material

This section contains additional material related to the Algorithmic Components Analysis Section.

A.1. Initial Solution Definition

Consider the scenario depicted in Figure 14, showing a single run for instance X-n936-k136. This example illustrates the evolution of the route minimization procedure with $\Delta_{RM} = 1000$ on the left and of the core optimization procedure with $\Delta_{CO} = 5000$ on the right. Both procedures ran for about three seconds and were applied to the same starting solution generated by the construction phase. By moving into the infeasible space, the route minimization procedure is very effective in quickly improving and compacting trivially bad initial solutions. However, since its structure is specifically designed to reduce the number of routes, the improvements vanish after a few hundred iterations.

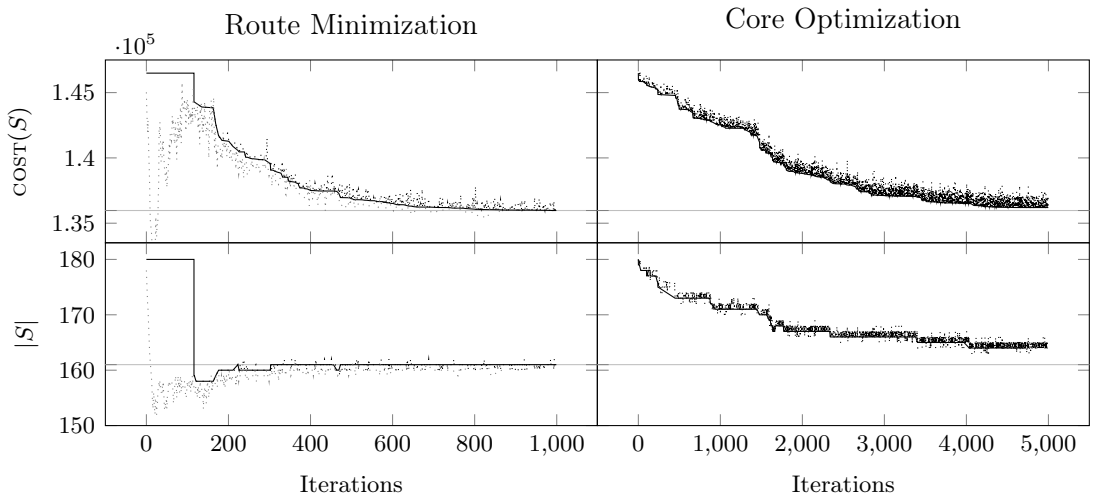


Figure 14 **Example of improvement procedures evolution when applied to the same initial solution for instance X-n936-k151. The continuous line shows the cost (top) and number of routes (bottom) associated with the best solution (in terms of cost) found up to that iteration; dots represents the current search trajectory. Gray and black dots are associated with infeasible and feasible solutions, respectively.**

A.2. Local Search

Figure 15 shows, for each local search operator applied to a shaken solution, the gap improvement when successfully applied, the application time in 10^{-6} seconds, and the success ratio computed as the number of improving applications over the total number of attempts. Similarly, Figure 16 shows the effect of EJCH$(\cdot)$ when applied to solutions that are already a local optimum for the first HRVND tier.

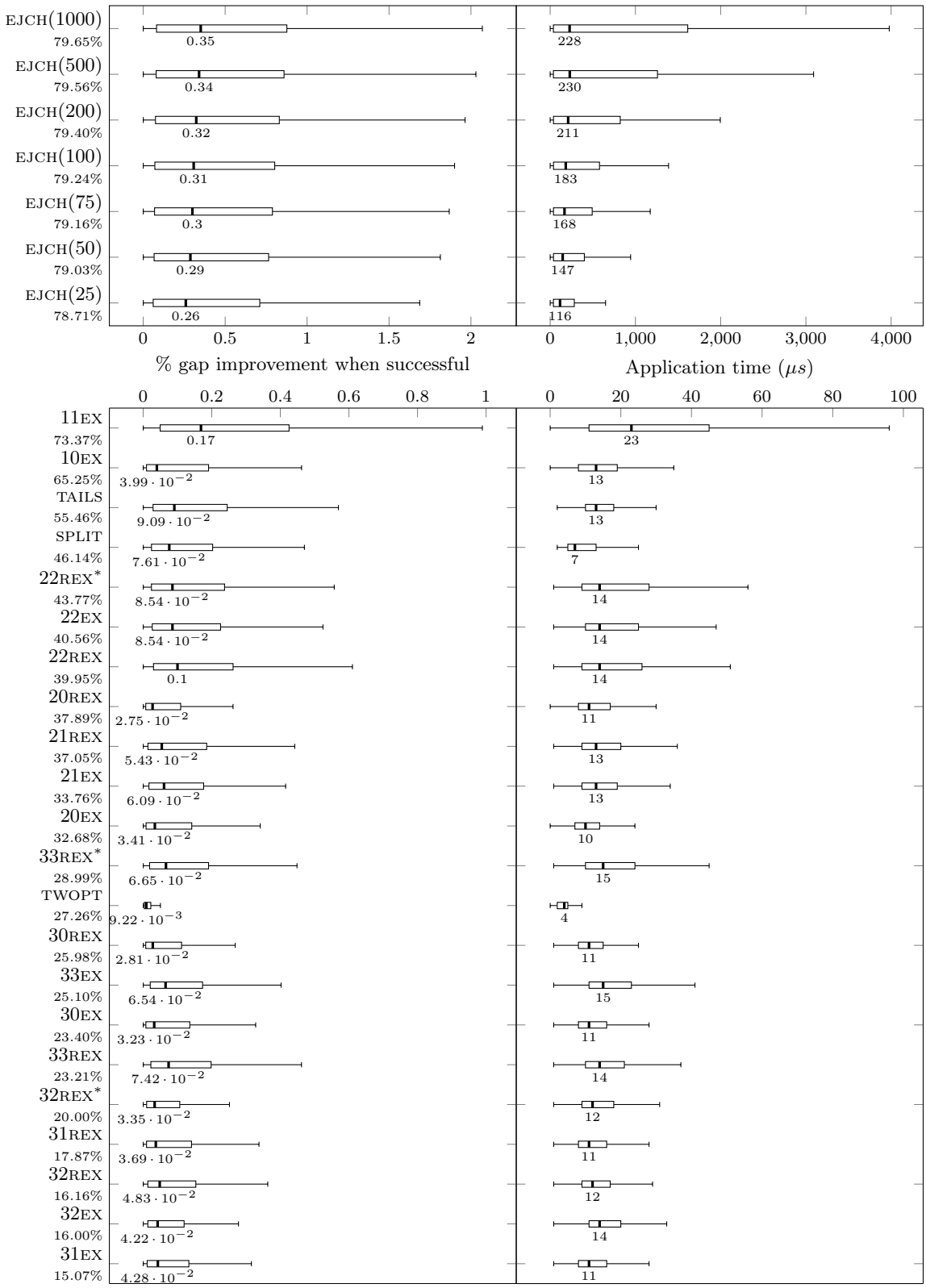


Figure 15 **Statistics for local search operators when applied to shaken solutions. For each operator, we report the expected gap improvement when successfully applied (left), the total exploration time (right), and the success ratio (below each operator's name). The median value is shown below each boxplot.**

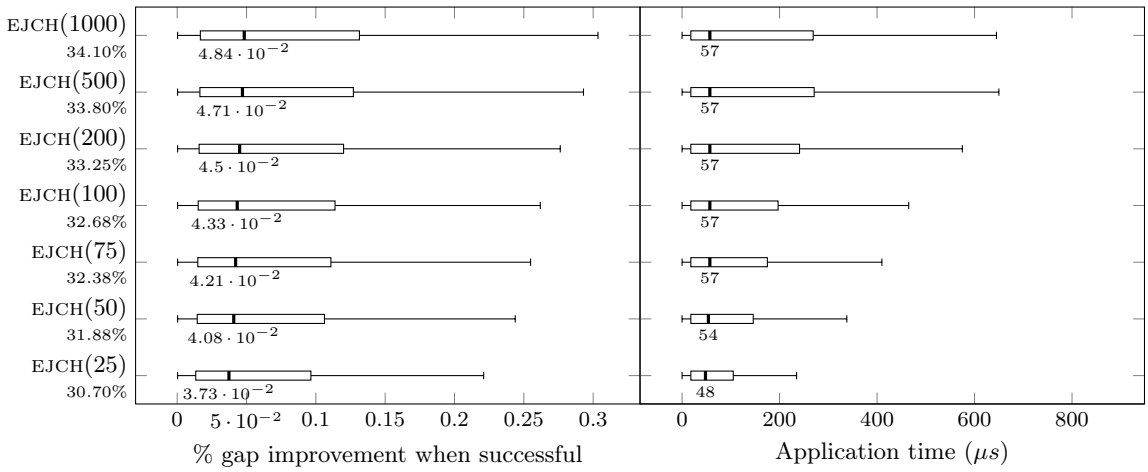


Figure 16 **Statistics for local search operators when applied to HRVND first tier local optima. For each operator, we report the expected gap improvement when successfully applied (left), the total exploration time (right), and the success ratio (below each operator's name). The median value is shown below each boxplot.**

A.3. Extreme Runs

Table 7 shows the results we obtained by setting $\Delta_{CO} = 10^7$.

A.4. Computations with Limited Memory Footprint

Random access memory (RAM) is relatively cheap nowadays, and large amounts are easily supported, even by low-cost laptops. Time, on the other hand, is much more valuable; new chips are moving to massive parallelization rather than an increase in their working frequency. Solution methods, however, seldom make use of parallel processing to solve a single instance, even though this approach might be a very interesting, yet challenging, research direction for very large-scale instances. We can thus state that the real bottleneck is probably not the amount of available RAM, but the computing time used to solve an instance. Indeed, the largest instance we considered, F2 from the $\mathbb{B}$ dataset, with thirty thousand vertices, can be easily processed on a laptop with 16GB of RAM. During the main core optimization procedure, only about 66% of the available RAM would be used. Despite the fact that RAM is not currently a limiting factor, in this section we investigate the behavior of FILO when the cost matrix, which is one of the most RAM-consuming data structures we use, is not stored but, instead, arc costs are computed on demand. In this case, the computing time increases by about 52% (i.e., from 1.72 minutes to 2.60 minutes). As an example, without storing the cost matrix, the F2 instance's RAM requirements drop from 10.56 GB to 3.68GB. This may allow the algorithm to be applicable to instances even larger than the one we considered. An alternative, more sophisticated, method proposed in Arnold, Gendreau, and Sörensen (2019), consists of storing a number of arcs connecting close vertices that are supposed to be used more frequently than others, in a hashmap.

Table 7 Long computations on large-sized $\mathbb{X}$ instances.

ID	BKS	Best	Avg	Worst	t
X-n502-k39	69226	0.00	0.01	0.04	262.65
X-n513-k21	24201	0.00	0.07	0.16	222.27
X-n524-k153	154593	0.01	0.14	0.27	146.99
X-n536-k96	94868	0.50	0.60	0.71	161.76
X-n548-k50	86700	0.01	0.02	0.09	207.85
X-n561-k42	42717	0.08	0.20	0.28	138.73
X-n573-k30	50673	0.12	0.22	0.26	205.74
X-n586-k159	190316	0.20	0.25	0.31	181.93
X-n599-k92	108451	0.15	0.22	0.33	189.87
X-n613-k62	59545	0.12	0.20	0.39	121.34
X-n627-k43	62173	0.03	0.12	0.32	198.31
X-n641-k35	63705	0.05	0.11	0.18	206.51
X-n655-k131	106780	0.00	0.02	0.03	378.73
X-n670-k130	146332	0.50	0.64	0.90	136.33
X-n685-k75	68225	0.17	0.34	0.52	142.94
X-n701-k44	81923	0.03	0.11	0.35	163.87
X-n716-k35	43387	0.09	0.16	0.29	176.98
X-n733-k159	136190	0.06	0.13	0.20	135.04
X-n749-k98	77314	0.15	0.25	0.38	141.90
X-n766-k71	114456	0.16	0.27	0.34	156.24
X-n783-k48	72394	0.10	0.17	0.26	191.01
X-n801-k40	73331	-0.03	0.09	0.15	188.16
X-n819-k171	158121	0.39	0.44	0.53	141.35
X-n837-k142	193737	0.12	0.21	0.26	191.22
X-n856-k95	88990	0.01	0.07	0.13	188.52
X-n876-k59	99303	0.11	0.16	0.24	176.58
X-n895-k37	53928	-0.04	0.13	0.31	189.72
X-n916-k207	329179	0.19	0.23	0.31	200.72
X-n936-k151	132812	0.16	0.26	0.38	127.16
X-n957-k87	85469	-0.00	0.06	0.11	195.38
X-n979-k58	118988	0.05	0.16	0.24	244.49
X-n1001-k43	72369	0.06	0.17	0.27	169.54
Mean		0.11	0.19	0.30	183.74

New best solutions: (X-n801-k40, 73311);(X-n895-k37, 53906);(X-n957-k87, 85467).

Appendix B: Move Generators and Static Move Descriptors

Efficiently managing move generators is crucial for any local search-based algorithm making use of GNs. In addition, flexibility might be required when experimenting different sparsification rules and composition of them, as we did in the analysis proposed in Section 4.4. In the following sections, we first discuss a flexible but still efficient implementation of move generators for the symmetric CVRP, supporting the union of different sparsification rules and a vertex-wise dynamic management. Then, we show how it can be used to efficiently implement SMD-based local search operators.

B.1. Move Generators: Storage and Management

Given a set R of sparsification rules, the complete set of move generators T is defined by the union of a number of move generator sets T_r, each one defined by a sparsification rule $r \in R$; that is, $T = \bigcup_{r \in R} T_r$. Each set T_r may be filtered according to a sparsification vector $\boldsymbol{\gamma} = (\gamma_0, \gamma_1, \ldots, \gamma_N)$ defining, for each vertex $i \in V$, a percentage $\gamma_i \in [0,1]$ of move generators in T_r to be considered as active. More precisely, the dynamic set of move generators $T_r^{\boldsymbol{\gamma}}$ filtered according to $\boldsymbol{\gamma}$ is defined as $T_r^{\boldsymbol{\gamma}} = \bigcup_{i \in V} \{(i,j),(j,i) : j \in V \wedge \textsc{Condition}(i,j,\gamma_i)\}$, where $\textsc{Condition}(i,j,\gamma_i)$ is a criterion that determines whether (i,j) and (j,i) are active based on the value of γ_i.

In the following, we describe a possible implementation of the above defined general framework for move generators. An illustrative example, representing the implementation of a set of move generators T defined by the union of two sparsification rules r_0 and r_1, is shown in Figure 17.

A list of move generators $L(T)$ is built by considering unique move generators defined by the different sparsification rules $r \in R$. By denoting with $L(T)_\ell$ the move generator $(i,j)_\ell$ indexed by ℓ in $L(T)$, we structured the list $L(T)$ so as to satisfy:

1. $L(T)_\ell = (i,j)_\ell \wedge L(T)_{\ell+1} = (j,i)_{\ell+1}$ for each even index ℓ;
2. $L(T) \not\ni (i,i), \forall i \in V$.

Condition 1 asks that both (i,j) and (j,i) are considered. This ensures the evaluation of a consistent set of moves when exploring asymmetric neighborhoods. Moreover, (i,j) and (j,i) are stored contiguously into $L(T)$ so that given a move generator indexed by ℓ, its reversed counterpart can be efficiently retrieved, when necessary. Finally, Condition 2 discards self-moves which are typically not used in local search procedures.

A number of lists $L(T_r, v)$, one for each vertex $v \in V$, is associated with each sparsification rule $r \in R$. Those lists identify a portion of $L(T)$ consisting of move generators $(i,j) \in T_r$. In particular, each list $L(T_r, v)$ keeps track of the even indices ℓ of move generators $(i,j)_\ell$ such that $v = i$ or $v = j$. Because of Condition 1 on $L(T)$, it is not necessary to store the index of the counterpart of (i,j) that can be found accessing $L(T)_{\ell+1}$.

In addition, each list $L(T_r, i)$, along with indices ℓ, stores an inclusion percentage value $p_{ri\ell}$ used to define whether move generators $(i,j)_\ell$ and $(j,i)_{\ell+1}$ are active according to the current sparsification factor γ_i. More precisely, the above defined $\textsc{Condition}(i,j,\gamma_i)$ is implemented as $\textsc{Condition}(i,j,\gamma_i) = p_{ri\ell} \leq \gamma_i$ where ℓ is the even index pointing to move generator $L(T)_\ell$ having i and j as endpoints.

Depending on how the inclusion percentage values are defined we can model the classical or the vertex-wise management of the dynamic move generators for a sparsification rule $r \in R$. In the classical management $\sum_{i \in V} \sum_{\ell \in L(T_r,i)} p_{ri\ell} = 1$ whereas in the vertex-wise management $\sum_{\ell \in L(T_r,i)} p_{ri\ell} = 1$ for each $i \in V$.

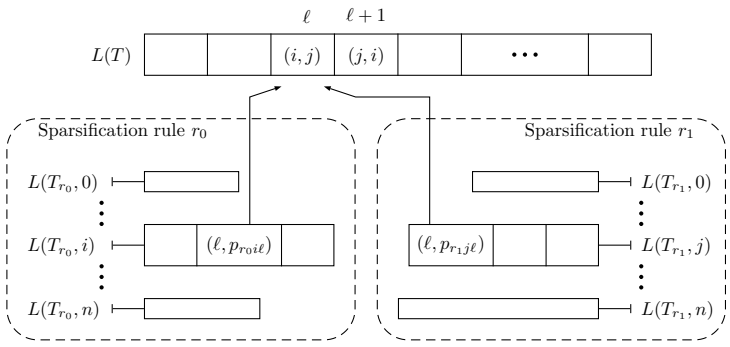


Figure 17 **Implementation of the list $L(T)$ of move generators defined by two sparsification rules r_0 and r_1. Two move generators (i,j) and (j,i) indexed ℓ and $\ell+1$ respectively, are explicitly shown. As can be seen, those move generators are defined by both sparsification rules and the associated entry in the lists point to the same shared entry in $L(T)$. The inclusion percentages $p_{r_0i\ell}$ and $p_{r_1j\ell}$ might however cause those move generators to be active at different times according to the value of γ_i and γ_j.**

The complete dynamic set of move generators can thus be addressed by iterating over each sparsification rule $r \in R$, each vertex $i \in V$, each list $L(T_r, i)$, and considering move generators $(i,j)_\ell$ and $(j,i)_{\ell+1}$ such that $p_{ri\ell} \leq \gamma_i$.

Note that by storing indices instead of move generators, different sparsification rules identifying the same subset of move generators would point to the same entry of $L(T)$.

In our implementation, the Sparsification Rule r_0 described in Section 2.2.2 is implemented by defining $p_{r_0v\ell} = n/|L(T_{r_0}, v)|$, where n is the index of move generator ℓ in a list of move generators $(i,j) \in L(T_{r_0}, i)$ sorted in increasing c_{ij} cost. The additional Sparsification Rule r_1 employed in the analysis of Section 4.4 defines instead $p_{r_1v\ell} = n/|T_{r_1}|$, where n is the index of move generator ℓ in a list of move generators $(i,j) \in L(T_{r_1})$ sorted in increasing c_{ij} cost.

B.2. An Abstract SMD-based Local Search Operator

The general structure of all SMD-based local search operators we used is shown in Algorithm 7. Note that, as mentioned in Section 2.2.3, we use the terms SMD and move generator interchangeably.

First, during a *pre-processing* step, additional computation useful for the actual neighborhood exploration may be executed. As an example, TAILS and SPLIT benefit from pre-computing route cumulative loads.

A heap data structure $\mathcal{H}$ is initialized with the currently active move generators according to the sparsification vector $\boldsymbol{\gamma}$, and such that at least one of the endpoints belongs to the set $\bar{V}_S$ of cached vertices for the solution S under examination. In particular, those move generators, denoted by $T^{\boldsymbol{\gamma}}(S)$, can be easily retrieved by using the data structures defined in Section B.1 and, more specifically, $T^{\boldsymbol{\gamma}}(S) = \bigcup_{r \in R, i \in \bar{V}_S} \{(i,j)_\ell : p_{ri\ell} \leq \gamma_i, \ell \in L(T_r, i)\}$. When dealing with local search operators defining asymmetric neighborhoods, both $(i,j)_\ell$ and $(j,i)_{\ell+1}$ have to be included. Given condition 1 on list $L(T)$ defined in Section B.1, this can be done by setting $T^{\boldsymbol{\gamma}}(S) = T^{\boldsymbol{\gamma}}(S) \cup \bigcup_{r \in R, i \in \bar{V}_S} \{(j,i)_{\ell+1} : p_{ri\ell} \leq \gamma_i, \ell \in L(T_r, i)\}$.

For each move generator $(i,j) \in T^{\boldsymbol{\gamma}}(S)$, the δ-tag $\delta(i,j)$, identifying the effect of the application of the move induced by (i,j) on S, is computed. Every (i,j) inducing an improving move (i.e., $\delta(i,j) < 0$) is inserted into

the heap data structure $\mathcal{H}$. By only inserting improving move generators, the heap computational complexity is kept at its minimum.

The heap $\mathcal{H}$ is linearly scanned until a feasible move is found. If such a move cannot be found, then S is considered to be a local optimum with respect to the local search operator under examination. Note that by heuristically restricting the initialization stage to only consider move generators involving vertices in $\bar{V}_S$ some improvement may be overlooked, but, as shown by the experiments in Section 4.5, this does not significantly affect the final solution quality.

Once a move generator (i, j) inducing a feasible and improving change to S is found, a list A of operator-dependant affected vertices is assembled. The application of (i, j) during the execute stage will change some of the δ-tag of move generators involving vertices in A.

The update stage recomputes the δ-tag for active move generators $U_{(i,j)} = \bigcup_{r \in R, i \in A} \{(v_1, v_2)_\ell : \ell \in L(T_r, i) \wedge (v_1 = i \vee v_2 = i) \wedge p_{ri\ell} \leq \gamma_i\}$. In case of an asymmetric neighborhood, the updates are extended to $U_{(i,j)} = U_{(i,j)} \cup \bigcup_{r \in R, i \in A} \{(v_2, v_1)_{\ell+1} : \ell \in L(T_r, i) \wedge (v_1 = i \vee v_2 = i) \wedge p_{ri\ell} \leq \gamma_i\}$. For each move generator requiring an update $(v_1, v_2) \in U_{(i,j)}$, the following cases are possible:

- (v_1, v_2) is removed from the heap $\mathcal{H}$ if $(v_1, v_2) \in \mathcal{H}$ and $\delta(v_1, v_2) \geq 0$;
- (v_1, v_2) is inserted into the heap $\mathcal{H}$ if $(v_1, v_2) \notin \mathcal{H}$ and $\delta(v_1, v_2) < 0$;
- the heap property is checked and possibly restored if $(v_1, v_2) \in \mathcal{H}$ and $\delta(v_1, v_2) < 0$;
- finally, (v_1, v_2) is ignored if $(v_1, v_2) \notin \mathcal{H}$ and $\delta(v_1, v_2) \geq 0$.

Since different sparsification rules may refer to the same move generators, a timestamp associated with each move generator (i, j) can be used to avoid evaluating it more than once, both in the initialization and in the update stages. Note that also when using a single sparsification rule r, a double evaluation may occur when $i, j \in A$ and $T_\ell = (i, j)_\ell$ is such that $\ell \in L(T_r, i) \wedge p_{ri\ell} < \gamma_i$ and $\ell \in L(T_r, j) \wedge p_{rj\ell} < \gamma_j$.

Algorithm 7 Abstract SMD-Based Local Search Operator

```
1: procedure Apply(S, T^γ)
2:     Preprocess(S)
3:     H ← Initialization(S, T^γ)
4:     n ← 0
5:     while n < len(H) do
6:         (i, j) ← Peek(H, n)
7:         n ← n + 1
8:         if ¬IsFeasible(S, (i, j)) then continue
9:         A ← AffectedVertices(S, (i, j))
10:        S ← Execute(S, (i, j))
11:        Update(H, T^γ, A)
12:        n ← 0
13:    end while
14: end procedure
```

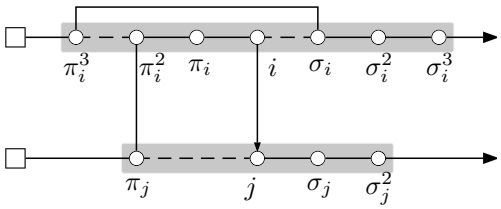


Figure 18 **A 30ex application induced by move generator (i, j) relocating path $(\pi_i^2 - i)$ between π_j and j. Some SMDs involving vertices in the gray area require an update to their δ-tag after the move execution.**

B.2.0.1. *Restricted Update for Asymmetric Neighborhoods.* The δ-tag update of move generators involving a vertex $i \in A$ for a local search operator defining an asymmetric neighborhood may sometimes be restricted from $\{(i, v_2), (v_1, i) : v_1, v_2 \in V\} \cap T^\gamma$ to only one between $\{(i, v_2) : v_2 \in V\} \cap T^\gamma$ and $\{(v_1, i) : v_1 \in V\} \cap T^\gamma$. As an example, consider the 30EX application shown in Figure 18. The set of affected vertices is $A = \{\pi_i^3, \pi_i^2, \pi_i, i, \sigma_i, \sigma_i^2, \sigma_i^3, \pi_j, j, \sigma_j, \sigma_j^2\}$. However, not all move generators $\{(i, j), (j, i) : i \in A, j \in V\} \cap T^\gamma$ have to be updated. For example, considering vertex π_i^3, move generators $(\pi_i^3, j), j \in V$ require an update since the successor of π_i^3 changes after the move execution, however, move generators $(j, \pi_i^3), j \in V$ do not, in fact, the predecessor of π_i^3 remains the same. For operator 30EX (and ignoring the current sparsification level), the total number of move generators requiring an update after a 30EX application can be reduced of approximately 36%.

Finally, we refer to Section B.3 of the Appendix for the operator-dependant definitions of the feasibility check, the assembly of the list A of vertices, the restricted update and the execution stage.

B.2.1. Dynamic Vertex-wise Move Generators for SMD-based Local Search Operators. Vertex-wise management of move generators requires a little extra care for the SMD update stage to be correctly performed. To highlight this, consider a scenario in which a vertex j is currently marked as cached at the beginning of a neighborhood exploration for a solution S; that is, $j \in \bar{V}_S$. An illustrative example is shown in Figure 19. The figure represents a portion of an instance with six customers together with a number of move generators depicted as lines ending with little circles. In particular, for a move generator (v_1, v_2), a full circle near to v_1 means that the move generator is currently active in v_1, i.e., $p_{rv_1\ell} \leq \gamma_{v_1}$ where ℓ is the index of $(v_1, v_2)_\ell$ in $L(T)$, while an empty circle represents the opposite scenario, i.e., $p_{rv_1\ell} > \gamma_{v_1}$. In the

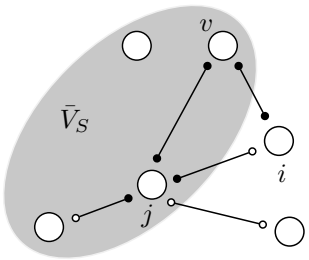


Figure 19 **A portion of an instance containing six customers (circles) and five move generators (lines). A move generator $(v_1, v_2)_\ell$ active in v_1, i.e., $p_{rv_1\ell} \leq \gamma_{v_1}$, is represented with a full circle near to v_1, whereas an empty circle denotes the opposite, i.e., $p_{rv_1\ell} > \gamma_{v_1}$. As an example (i, j) is active in j but not in i. The direction of move generators is not shown because not relevant.**

example, (i, j) is active in j but not in i. Finally, the grayed area identifies the set $\bar{V}_S$ of currently cached vertices for S.

During the SMD initialization stage, move generators involving vertices in $\bar{V}_S$ are considered to be inserted into the heap data structure $\mathcal{H}$. Suppose that, because inducing an improving move and currently active in j, move generator (i, j) along with with other improving move generators including (i, v) are inserted into $\mathcal{H}$. During the SMD search stage, move generator (i, v) is found feasible before the evaluation of (i, j). The move induced by (i, v) is then applied and a number of affected vertices A may be such that $i \in A$ but $j \notin A$. Note that i shares (i, j) with j but (i, j) is not currently active in i because of the sparsification factor γ_i. The SMD update stage requires that, from each vertex $v \in A$ active move generators are updated. *Should* (i, j) *still be updated even if not active in* i*?* The answer is yes. Being (i, j) active in j, it may be, as described in this scenario, already into the heap $\mathcal{H}$ and possibly be extracted during future search stages. In fact, being i among the affected vertices A as a result of the application of (i, v), the δ-tag of any move generator involving a vertex $v \in A$ requires an update, and hence (i, j) does. On the other hand, if (i, j) were not active in both j and i, updating it after (i, v) was not required because it could have never been inserted into $\mathcal{H}$.

The dynamic management of vertex-wise move generators may thus require the update of move generators that are not active in one of the affected vertices but only active in the other endpoint that may not be in the list of vertices affected by the move application. A possible implementation uses two additional flags per move generator (i, j) storing whether it is active in i and/or in j. During the SMD update stage the flags are checked to identify whether an update is required.

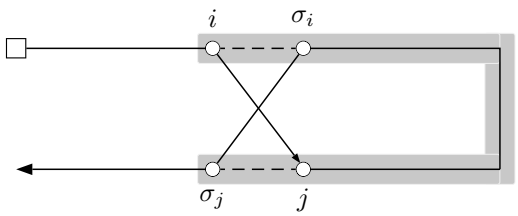
Finally, note that this approach works correctly under the assumption that the sparsification vector $\boldsymbol{\gamma}$ is not changed during a neighborhood exploration.

B.3. SMD Implementation Details of Local Search Operators

In the following, we provide a detailed description of the implementation for the local search operators used in FILO. In particular, we detail the operator-dependant procedures to be defined when applying a move induced by a move generator (i, j) within an SMD-based operator. To better accomplish this, we introduce few additional notation elements. In particular, we denote by π_i^ℓ and σ_i^ℓ the ℓ^{th}- predecessor and successor of vertex $i \in V$, respectively, in the solution under examination. We omit the apex when $\ell = 1$. Moreover, we identify with q_i^{up} and q_i^{from} the cumulative load up to and from any customer $i \in V_c$ included, i.e., $q_i^{up} = q_i + q_{\pi_i} + q_{\pi_i^2} + \ldots + q_0$ and $q_i^{from} = q_i + q_{\sigma_i} + q_{\sigma_i^2} + \ldots + q_0$. In the following paragraphs, a figure is shown for each local search operator highlighting the move induced by a move generator (i, j) and the set of vertices affected by its application (grayed area). Note that the move generator direction does not necessarily reflect the crossing direction in the resulting route. Finally, as defined in Section 2.2, path is used to refer to a contiguous sequence of vertices belonging to a route, and head and tail are used to denote a path belonging respectively to the initial and final part of a route.

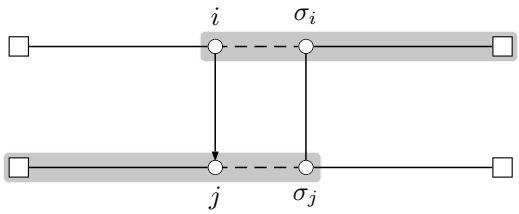
B.3.1. TWOPT.

Replace a path of vertices with its reverse.



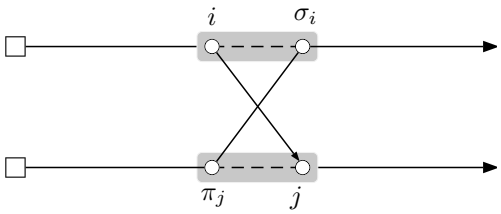
- Type: symmetric.
- Pre-processing: none.
- Cost computation: $\delta_{ij} = -c_{i\sigma_i} + c_{ij} - c_{j\sigma_j} + c_{\sigma_i\sigma_j}$.
- Feasibility Check: $r_i = r_j$.
- Update List: all vertices between i and σ_j, for which successors and predecessors change after the move application.
- Execution: reverse the path between σ_i and j included.

B.3.2. **SPLIT.** Replace the tail of route r_i with the reversed head of route r_j and replace the head of route r_j with the reversed tail of route r_i.



- Type: symmetric.
- Pre-processing: compute q_i^{up} and q_i^{from} for any customer $i \in V_c$.
- Cost Computation: $\delta_{ij} = -c_{i\sigma_i} + c_{ij} - c_{j\sigma_j} + c_{\sigma_i\sigma_j}$.
- Feasibility Check: $(r_i \neq r_j) \wedge (q_i^{up} + q_j^{up} \leq Q) \wedge (q_{\sigma_j}^{from} + q_{\sigma_i}^{from} \leq Q)$.
- Update List: all vertices from i to the depot and from the depot to σ_j, for which successors and predecessors change after the move application.
- Execution: replace (i, σ_i) and (j, σ_j) with (i, j) and (σ_i, σ_j) and reverse paths from depot to j and from σ_i to depot. Update the cumulative load q_v^{up} and q_v^{from} for customers v belonging to r_i and r_j, if not empty.

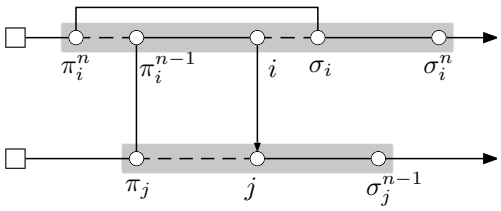
B.3.3. TAILS. Swap the tails of two different routes.



- Type: asymmetric.
- Pre-processing: compute q_i^{up} and q_i^{from} for any customer $i \in V_c$.
- Cost Computation: $\delta_{ij} = -c_{i\sigma_i} + c_{ij} - c_{j\pi_j} + c_{\pi_j\sigma_i}$.
- Feasibility Check: $(r_i \neq r_j) \wedge (q_i^{up} + q_j^{from} \leq Q) \wedge (q_{\pi_j}^{up} + q_{\sigma_i}^{from} \leq Q)$.
- Update List: i, σ_i, j and π_j.

  The update can be restricted to move generators $\{(i,v),(v,\sigma_i),(v,j),(\pi_j,v) : v \in V\} \cap T^{\gamma}$.
- Execution: replace (i,σ_i) and (π_j,j) with (i,j) and (π_j,σ_i). Update the cumulative load q_v^{up} and q_v^{from} for customers v belonging to r_i and r_j, if not empty.

B.3.4. n0EX with $n \geq 1$. Relocate a path of n vertices within the same or into a different route.



- Type: asymmetric.
- Pre-processing: none.
- Cost Computation: $\delta_{ij} = -c_{\pi_i^n\pi_i^{n-1}} - c_{i\sigma_i} - c_{j\pi_j} + c_{\pi_i^n\sigma_i} + c_{\pi_j\pi_i^{n-1}} + c_{ij}$.
- Feasibility Check: logical disjunction of
  - $(r_i = r_j) \wedge \bigwedge_{\ell=1}^{n-1}(j \neq \pi_i^{\ell}) \wedge (j \neq \sigma_i)$.

    When $r_i = r_j$ and $\bigvee_{\ell=1}^{n-1} j = \pi_i^{\ell}$, the path to relocate overlaps with the destination position.

    When $r_i = r_j$ and $j = \sigma_i$, the move does nothing but the δ_{ij} computation is not correct.
  - $(r_i \neq r_j) \wedge (i \neq 0) \wedge \bigwedge_{\ell=1}^{n-1}(\pi_i^{\ell} \neq 0) \wedge (q_{r_j} + q_i + \sum_{\ell=1}^{n-1} q_{\pi_i^{\ell}} \leq Q)$.

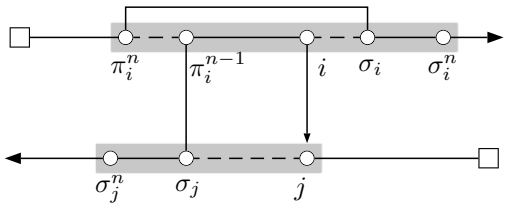
    The condition makes sure the depot is not relocated and the capacity constraint of the target route is respected.
- Update List: $\bigcup_{\ell=1}^{n} \pi_i^{\ell} \cup i \cup \bigcup_{\ell=1}^{n} \sigma_i^{\ell} \cup \pi_j \cup j \cup \bigcup_{\ell=1}^{n-1} \sigma_j^{\ell}$.

  The update can be restricted to move generators

— $\{(\pi_i^n, v), (\pi_i^{n-1}, v), (v, \pi_i^{n-1}) : v \in V\} \cap T^{\gamma}$;
— $\{(\pi_i^{\ell}, v) : \ell = n-2, \ldots, 1 \text{ and } v \in V\} \cap T^{\gamma}$;
— $\{(i, v) : v \in V\} \cap T^{\gamma}$;
— if $n = 1$ include $\{(v, i) : v \in V\} \cap T^{\gamma}$;
— $\{(\sigma_i, v), (v, \sigma_i) : v \in V\} \cap T^{\gamma}$;
— $\{(\sigma_i^{\ell}, v) : \ell = 2, \ldots, n \text{ and } v \in V\} \cap T^{\gamma}$;
— $\{(\pi_j, v) : v \in V\} \cap T^{\gamma}$;
— $\{(j, v), (v, j) : v \in V\} \cap T^{\gamma}$;
— $\{(\sigma_j^{\ell}, v) : \ell = 1, \ldots, n-1 \text{ and } v \in V\} \cap T^{\gamma}$.

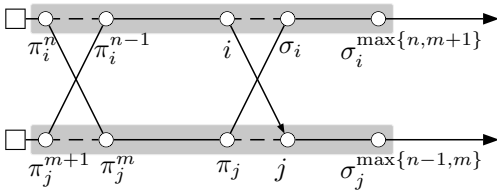
- Execution: replace $(\pi_i^n, \pi_i^{n-1}), (i, \sigma_i)$ and (j, π_j) with $(\pi_i^n, \sigma_i), (\pi_i^{n-1}, \pi_j)$ and (i, j).

B.3.5. *n*0REX with $n \geq 2$. Relocate a reversed path of n vertices within the same or into a different route.



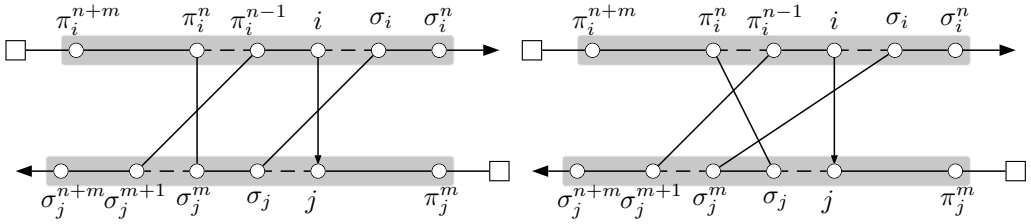
- Type: asymmetric.
- Pre-processing: none.
- Cost Computation: $\delta_{ij} = -c_{\pi_i^n \pi_i^{n-1}} - c_{i\sigma_i} - c_{j\sigma_j} + c_{\pi_i^n \sigma_i} + c_{\sigma_j \pi_i^{n-1}} + c_{ij}$.
- Feasibility Check: logical disjunction of
  — $(r_i = r_j) \wedge \bigwedge_{\ell=1}^{n} (j \neq \pi_i^{\ell})$.
  When $r_i = r_j$ and $\bigvee_{\ell=1}^{n-1} j = \pi_i^{\ell}$, the path to relocate overlaps with the destination position.
  When $r_i = r_j$ and $j = \pi_i^n$, the move could be reduced to a TWOPT induced by move generator (j, i) but would require a special handling in this context.
  — $(r_i \neq r_j) \wedge (i \neq 0) \wedge \bigwedge_{\ell=1}^{n-1} (\pi_i^{\ell} \neq 0) \wedge (q_{r_j} + q_i + \sum_{\ell=1}^{n-1} q_{\pi_i^{\ell}} \leq Q)$.
  The condition makes sure the depot is not relocated and the capacity constraint of the target route is respected.
- Update List: $\bigcup_{\ell=1}^{n} \pi_i^{\ell} \cup i \cup \bigcup_{\ell=1}^{n} \sigma_i^{\ell} \cup j \cup \bigcup_{\ell=1}^{n} \sigma_j^{\ell}$.
  The update can be restricted to move generators
  — $\{(\pi_i^{\ell}, v), (\pi_i^{\ell}, v), (i, v), (v, i) : \ell = n, \ldots, 1 \text{ and } v \in V\} \cap T^{\gamma}$;
  — $\{(\sigma_i^{\ell}, v), (\sigma_j^{\ell}, v) : \ell = 1, \ldots, n \text{ and } v \in V\} \cap T^{\gamma}$;
  — $\{(j, v), (v, j) : v \in V\} \cap T^{\gamma}$.
- Execution: replace $(\pi_i^n, \pi_i^{n-1}), (i, \sigma_i)$ and (j, σ_j) with $(\pi_i^n, \sigma_i), (\pi_i^{n-1}, \sigma_j)$ and (i, j).

B.3.6. *nm*EX with $1 \leq m \leq n$. Swap a path of n vertices with a path of m vertices within the same or between different routes.



- Type: asymmetric.
- Pre-processing: none.
- Cost Computation: $\delta_{ij} = -c_{\pi_i^n \pi_i^{n-1}} - c_{i\sigma_i} - c_{\pi_j^{m+1} \pi_j^m} - c_{j\pi_j} + c_{\pi_i^n \pi_j^m} + c_{\pi_j \sigma_i} + c_{\pi_j^{m+1} \pi_i^{n-1}} + c_{ij}$.
- Feasibility Check: logical disjunction of
  - $(r_i = r_j) \wedge \bigwedge_{\ell=1}^{n-1} (j \neq \pi_i^\ell) \wedge \bigwedge_{\ell=1}^{m+1} (j \neq \sigma_i^\ell)$.
    When $r_i = r_j$ and $\bigvee_{\ell=1}^{m} j = \sigma_i^\ell \vee \bigvee_{\ell=1}^{n-2} j = \pi_i^\ell$, the paths overlap.
    When $r_i = r_j$ and $j = \sigma_i^{m+1}$, the move could be reduced to a m0EX induced by move generator (σ_i, π_i^{n-1}) or to a n0EX induced by move generator (i, j) but would require a special handling in this context.
    When $r_i = r_j$ and $j = \pi_i^{n-1}$, vertex j is at the same time part of the path to move and destination of the movement.
  - $(r_i \neq r_j) \wedge (i \neq 0) \wedge \bigwedge_{\ell=1}^{n-1} (\pi_i^\ell \neq 0) \wedge \bigwedge_{\ell=1}^{m} (\pi_j^\ell \neq 0) \wedge (q_{r_j} + q_i + \sum_{\ell=1}^{n-1} q_{\pi_i^\ell} - \sum_{\ell=1}^{m} q_{\pi_j^\ell} \leq Q) \wedge (q_{r_i} - q_i - \sum_{\ell=1}^{n-1} q_{\pi_i^\ell} + \sum_{\ell=1}^{m} q_{\pi_j^\ell} \leq Q)$.
    The condition makes sure the depot is not relocated and the capacity constraints are not violated.
- Update List: $\bigcup_{\ell=1}^{n} \pi_i^\ell \cup i \cup \bigcup_{\ell=1}^{\max\{n,m+1\}} \sigma_i^\ell \cup \bigcup_{\ell=1}^{m+1} \pi_j^\ell \cup j \cup \bigcup_{\ell=1}^{\max\{n-1,m\}} \sigma_j^\ell$.
  The update can be restricted to move generators
  - $\{(\pi_i^n, v) : v \in V\} \cap T^\gamma$;
  - $\{(\pi_i^\ell, v) : \ell = n-1, \ldots, 1 \text{ and } v \in V\} \cap T^\gamma$;
  - $\{(v, \pi_i^\ell) : \ell = n-1, \ldots, n-1-m \text{ and } v \in V\} \cap T^\gamma$;
  - $\{(i, v) : v \in V\} \cap T^\gamma$;
  - if $n - m < 2$ include $\{(v, i) : v \in V\} \cap T^\gamma$;
  - $\{(\sigma_i, v), (v, \sigma_i) : v \in V\} \cap T^\gamma$;
  - $\{(\sigma_i^\ell, v) : \ell = 2, \ldots, n \text{ and } v \in V\} \cap T^\gamma$;
  - $\{(v, \sigma_i^\ell) : \ell = 2, \ldots, m+1 \text{ and } v \in V\} \cap T^\gamma$;
  - $\{(\pi_j^{m+1}, v) : v \in V\} \cap T^\gamma$;
  - $\{(\pi_j^\ell, v), (v, \pi_j^\ell) : \ell = m, \ldots, 1 \text{ and } v \in V\} \cap T^\gamma$;
  - $\{(j, v), (v, j) : v \in V\} \cap T^\gamma$
  - $\{(\sigma_j^\ell, v) : \ell = 1, \ldots, n-1 \text{ and } v \in V\} \cap T^\gamma$;
  - $\{(v, \sigma_j^\ell) : \ell = 1, \ldots, m \text{ and } v \in V\} \cap T^\gamma$.
- Execution: replace (π_i^n, π_i^{n-1}), (i, σ_i), (π_j^{m+1}, π_j^m) and (j, π_j) with (π_i^n, π_j^m), $(\pi_i^{n-1}, \pi_j^{m+1})$, (π_j, σ_i) and (i, j).

B.3.7. nmREX (and nmREX*) with $1 \leq m \leq n$. Swap a reversed path of n vertices with a path of m vertices within the same or between different routes (nmREX). If $m \geq 2$, we consider an additional variant that also reverses the path of m vertices (nmREX*).

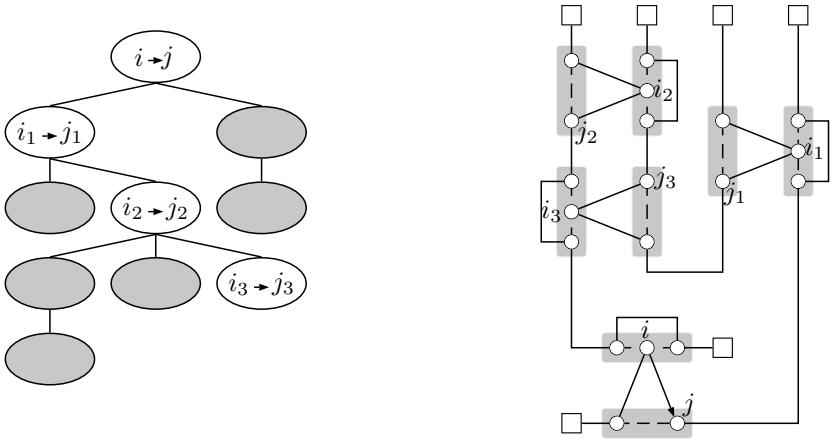


- Type: asymmetric.
- Pre-processing: none.
- Cost Computation:
  - nmREX*: $\delta_{ij} = -c_{\pi_i^n \pi_i^{n-1}} - c_{i\sigma_i} - c_{j\sigma_j} - c_{\sigma_j^m \sigma_j^{m+1}} + c_{\pi_i^n \sigma_j^m} + c_{\pi_i^{n-1} \sigma_j^{m+1}} + c_{\sigma_j \sigma_i} + c_{ij}$.
  - nmREX: $\delta_{ij} = -c_{\pi_i^n \pi_i^{n-1}} - c_{i\sigma_i} - c_{j\sigma_j} - c_{\sigma_j^m \sigma_j^{m+1}} + c_{\pi_i^n \sigma_j} + c_{\pi_i^{n-1} \sigma_j^{m+1}} + c_{\sigma_j^m \sigma_i} + c_{ij}$.
- Feasibility Check: logical disjunction of
  - $(r_i = r_j) \wedge \bigwedge_{\ell=1}^{n+m} (j \neq \pi_i^\ell)$.
    When $r_i = r_j$ and $\bigvee_{\ell=1}^{n+m-1} j = \sigma_i^\ell$, the paths overlap.
    When $r_i = r_j$ and $j = \sigma_i^{m+n}$
    - nmREX*: the move could be reduced to a TWOPT induced by move generator (j, i);
    - nmREX: the move could be reduced to a $n0$REX induced by move generator (i, j);

    but, in both cases, this would require a special handling.
  - $(r_i \neq r_j) \wedge (i \neq 0) \wedge \bigwedge_{\ell=1}^{n-1} (\pi_i^\ell \neq 0) \wedge \bigwedge_{\ell=1}^{m} (\sigma_j^\ell \neq 0) \wedge (q_{r_j} + q_i + \sum_{\ell=1}^{n-1} q_{\pi_i^\ell} - \sum_{\ell=1}^{m} q_{\sigma_j^\ell} \leq Q) \wedge (q_{r_i} - q_i - \sum_{\ell=1}^{n-1} q_{\pi_i^\ell} + \sum_{\ell=1}^{m} q_{\sigma_j^\ell} \leq Q)$.
    The condition makes sure the depot is not relocated and the capacity constraints are not violated.
- Update List: $\bigcup_{\ell=1}^{n+m} \pi_i^\ell \cup i \cup \bigcup_{\ell=1}^{n} \sigma_i^\ell \cup \bigcup_{\ell=1}^{m} \pi_j^\ell \cup j \cup \bigcup_{\ell=1}^{n+m} \sigma_j^\ell$.
  The update can be restricted to move generators
  - $\{(v, \pi_i^\ell) : \ell = n+m, \ldots, n+1 \text{ and } v \in V\} \cap T^\gamma$;
  - $\{(\pi_i^\ell, v), (v, \pi_i^\ell) : \ell = n, \ldots, 1 \text{ and } v \in V\} \cap T^\gamma$;
  - $\{(i, v), (v, i) : v \in V\} \cap T^\gamma$;
  - $\{(\sigma_i^\ell, v) : \ell = 1, \ldots, n \text{ and } v \in V\} \cap T^\gamma$;
  - $\{(\sigma_j^\ell, v) : \ell = n+m, \ldots, m+1 \text{ and } v \in V\} \cap T^\gamma$;
  - $\{(\sigma_j^\ell, v), (v, \sigma_j^\ell) : \ell = m, \ldots, 1 \text{ and } v \in V\} \cap T^\gamma$;
  - $\{(j, v), (v, j) : v \in V\} \cap T^\gamma$;
  - $\{(v, \pi_j^\ell) : \ell = 1, \ldots, m \text{ and } v \in V\} \cap T^\gamma$.
- Execution:
  - nmREX*: Replace $(\pi_i^n, \pi_i^{n-1}), (i, \sigma_i), (j, \sigma_j)$ and $(\sigma_j^m, \sigma_j^{m+1})$ with $(\pi_i^n, \sigma_j^m), (\pi_i^{n-1}, \sigma_j^{m+1}), (\sigma_j, \sigma_i)$ and (i, j). Reverse the paths between $\pi^n - 1$ and i and the path between σ_j and σ_j^m.

— nmREX: Replace $(\pi_i^n, \pi_i^{n-1}), (i, \sigma_i), (j, \sigma_j)$ and $(\sigma_j^m, \sigma_j^{m+1})$ with $(\pi_i^n, \sigma_j), (\pi_i^{n-1}, \sigma_j^{m+1}), (\sigma_j^m, \sigma_i)$ and (i, j). Reverse the path between $\pi^n - 1$ and i.

B.3.8. **EJCH.** Perform a sequence of 10EX applications.



- Type: asymmetric.
- Pre-processing: none.
- Cost Computation: same as for 10EX operator.
- Feasibility Check: a tree of nodes associated with 10EX moves is generated by using (i, j) as tree root. A path from the tree root (i, j) to any other node in the tree is a sequence of 10EX moves. Every sequence s stores a number of state variables defining the effect of its application on the current solution. The goal of the feasibility check is to find a sequence whose application generates a feasible and improving solution. In particular, each sequence s contains
  - the modified loads q_r^s for each route r affected by 10EX moves of s;
  - the change in the objective function δ_s due to the application of 10EX moves of s;
  - a set $\mathcal{F}_i^s$ storing customers that cannot be relocated because already involved in previous relocate moves within s. More precisely, the successors or predecessors of customers in $\mathcal{F}_i^s$ have changed in previous 10EX moves of s but the current structure of the solution does not reflect these changes. Note, in fact, that during the feasibility check we are only simulating the 10EX effects to find a feasible and improving sequence without really changing the solution;
  - finally, a set $\mathcal{F}_j^s$ storing customers that cannot be the target of a relocate move because already involved in previous relocate moves within s. More precisely, the predecessors of customers in $\mathcal{F}_j^s$ have changed in previous 10EX moves of s but, as described above, the current structure of the solution does not reflect these changes.

  Note that a different handling without $\mathcal{F}_i^s$ and $\mathcal{F}_j^s$ would require, for each tree node associated with a sequence s, to keep a copy of the solution.

A sequence s of 10EX moves generating a cost change δ_s in the objective function, ending with a move (i_n, j_n) that relocates customer $i_n \in V_c$ from route r_{i_n} to route r_{j_n} before vertex $j_n \in V$, is extended by scanning all customers i_{n+1} belonging to $r_{j_n} = r_{i_{n+1}}$ that satisfy the following joint conditions:

— $q^s_{r_{i_{n+1}}} - q_{i_{n+1}} \leq Q$.

The removal of i_{n+1} restores the feasibility of route $r_{i_{n+1}}$ that was violated by the previous insertion of j_n.

— $i_{n+1} \notin \mathcal{F}^s_i$.

Customer i_{n+1} can be relocated.

Every customer i_{n+1} satisfying the previous conditions is considered as the new starting point for a 10EX for which a potential endpoint is generated by scanning the currently active move generators T^γ that have i_{n+1} as the object of the relocation, i.e., $\{(i_{n+1}, j_{n+1}), j_{n+1} \in V_c\} \cap T^\gamma$. A customer $j_{n+1} \in V_c$ belonging to route $r_{j_{n+1}}$ is selected to be the target of the relocation if it satisfies the following joint conditions:

— $j_{n+1} \neq 0$.

The depot alone does not allow to identify a specific route.

— $r_{i_{n+1}} \neq r_{j_{n+1}}$.

Customer i_{n+1} is relocated into a route different from the origin one.

— $\delta_s + \delta_{i_{n+1} j_{n+1}} < 0$.

Sequence s still provides an improvement to the objective function.

— $j_{n+1} \notin \mathcal{F}^s_j$.

Customer j_{n+1} can be the target of a relocation.

Every customer j_{n+1} satisfying the previous conditions is considered as an endpoint for the (i_{n+1}, j_{n+1}) 10EX move and a new tree node s', son of the current one s, is created. State variables of s' are defined as follows:

— $q^{s'}_{r_{i_{n+1}}} = q^s_{r_{i_{n+1}}} - q_{i_{n+1}}$;
— $q^{s'}_{r_{j_{n+1}}} = q^s_{r_{j_{n+1}}} + q_{j_{n+1}}$;
— $\delta_{s'} = \delta_s + \delta_{i_{n+1} j_{n+1}}$;
— $\mathcal{F}^{s'}_i = \mathcal{F}^s_i \cup \{\pi_{i_{n+1}}, i_{n+1}, \sigma_{i_{n+1}}, \pi_{j_{n+1}}, j_{n+1}\}$;
— $\mathcal{F}^{s'}_j = \mathcal{F}^s_j \cup \{i_{n+1}, \sigma_{i_{n+1}}, j_{n+1}\}$.

The tree frontier is explored by following a best-δ-first strategy, that is, the sequence providing the greatest improvement is always extended first. This is obtained by using an additional heap data structure managing the tree nodes. As can be inferred by the above description, sequences are not limited in depth and the same route can be accessed by a 10EX move more than once. A limit is, however, imposed on the total maximum number of explored tree nodes which in the proposed implementation is $n_{EC} = 25$. Finally, the feasibility check step ends as soon as a feasible sequence is found, i.e., the last 10EX move does not violate the capacity of the target route, or the maximum number of tree nodes is explored.

- Update List: $\mathcal{F}_i^{s^*}$ with s^* a feasible improving sequence.

  For each 10EX (i, j) move generator composing the sequence s^*, the update can be restricted to move generators $\{(\pi_i, v), (i, v), (v, i), (\sigma_i, v), (v, \sigma_i), (j, v), (v, j), (\pi_j, v) : v \in V\} \cap T^{\gamma}$

- Execution: execute the 10EX moves of a feasible sequence. Note that, due to the restrictions imposed during the sequence space exploration, the order in which moves are executed does not affect the final result.

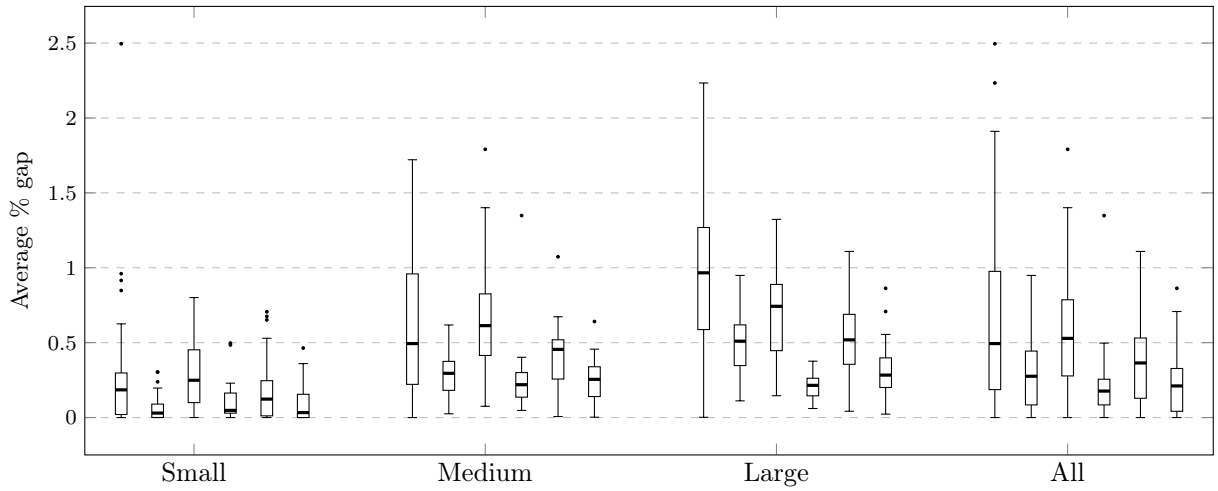


Figure 20 **Comparison of average gaps obtained by algorithms on the $\mathbb{X}$ dataset. For each group, boxplots, from left to right, are associated with: ILS-SP, HGSADC, KGLS, SISR, FILO, and FILO (long).**

Appendix C: Computational details for $\mathbb{X}$ instances

Detailed results about computations on the $\mathbb{X}$ dataset can be found in Tables 9 – 11 and Figure 20. Moreover, to better assess whether the results obtained by FILO are statistically different with respect to competing algorithms, we followed the procedure used in Christiaens and Vanden Berghe (2020). In particular, we conducted a one-tailed Wilcoxon signed-rand test (Wilcoxon (1945)) in which we consider a null hypothesis H_0

$$H_0 : \textsc{AverageCost}(FILO) = \textsc{AverageCost}(X)$$

and an alternative hypothesis H_1

$$H_1 : \textsc{AverageCost}(FILO) > \textsc{AverageCost}(X)$$

where X can be ILS-SP, HGSADC, KGLS, and SISR. We tested the above hypotheses on small, medium, large and over all the instances. The p-values associated with the tested hypothesis are shown in Table 8 (left). The p-values for a similar analysis in which we compared FILO (long) with competing methods are shown in Table 8 (right).

A hypothesis is rejected when its associated p-value is greater than a significance level α. Failing to reject H_0 means that the average results of the two compared methods are not statistically different. On the other hand, when H_0 is rejected, the average results obtained by the methods are statistically different and the alternative hypothesis H_1 can be tested to find whether the average results obtained by FILO are statistically greater than those of the competing method. Rejecting H_1 implies that FILO performs better than the competing method.

When performing multiple comparisons involving the same data, the probability of erroneously rejecting a null hypothesis increases. To control these errors, the significance level α is typically adjusted to lower values. Bonferroni correction (Dunn (1961)) is a simple method used to adjust α when performing multiple comparisons. In particular, given n comparisons, the significance level is set to α/n. In our case, for each

Table 8 Computations on the 𝕏 dataset: p-values for FILO on the left and FILO (long) on the right.

	Small					Small			
	ILS-SP	HGSADC	KGLS	SISR		ILS-SP	HGSADC	KGLS	SISR
H_0	0.309491	**0.000273**	0.020427	0.064141	H_0	**0.000273**	0.808654	**0.000140**	0.130121
H_1	0.154746	0.999876	0.010214	0.969381	H_1	**0.000136**	0.404327	**0.000070**	0.065061
	Similar	Worse	Similar	Similar		Better	Similar	Better	Similar
	Medium					Medium			
	ILS-SP	HGSADC	KGLS	SISR		ILS-SP	HGSADC	KGLS	SISR
H_0	0.071754	**0.001737**	**0.000000**	**0.000066**	H_0	**0.001057**	0.046584	**0.000000**	0.346122
H_1	0.035877	0.999183	**0.000000**	0.999970	H_1	**0.000528**	0.023292	**0.000000**	0.830925
	Similar	Worse	Better	Worse		Better	Similar	Better	Similar
	Large					Large			
	ILS-SP	HGSADC	KGLS	SISR		ILS-SP	HGSADC	KGLS	SISR
H_0	**0.000335**	1.000000	**0.000234**	**0.000000**	H_0	**0.000000**	**0.000837**	**0.000000**	**0.002227**
H_1	**0.000167**	0.503678	**0.000117**	1.000000	H_1	**0.000000**	**0.000419**	**0.000000**	0.998963
	Better	Similar	Better	Worse		Better	Better	Better	Worse
	All					All			
	ILS-SP	HGSADC	KGLS	SISR		ILS-SP	HGSADC	KGLS	SISR
H_0	**0.000036**	0.007739	**0.000000**	**0.000000**	H_0	**0.000000**	**0.000111**	**0.000000**	0.065432
H_1	**0.000018**	0.996173	**0.000000**	1.000000	H_1	**0.000000**	**0.000056**	**0.000000**	0.967546
	Better	Similar	Better	Worse		Better	Better	Better	Similar

p-values in bold are associated with rejected hypothesis when $\alpha = 0.003125$.
The last row of each group contains a p-value interpretation when $\alpha = 0.003125$. In particular, FILO is not statistically different from the competing method when H_0 cannot be rejected (Similar), FILO is statistically better when both H_0 and H_1 are rejected (Better), and, finally, FILO is statistically worse when H_0 is rejected and H_1 is not rejected (Worse).

FILO configuration, we tested a total number of $n = 8$ hypotheses corresponding to the partitioning of instances (Small, Medium, Large, and All) and to the two hypotheses (H_0 and H_1). Thus, by assuming an initial significance level $\alpha_0 = 0.025$, the adjusted value becomes $\alpha = \alpha_0/8 = 0.003125$.

As can ben seen from Table 8 (left)

- FILO performs better than ILS-SP on large instances and on all the 𝕏 dataset, and it has a similar performance on small and medium instances;
- FILO has a similar performance compared to HGSADC on large instances and on the whole 𝕏 dataset, however, HGSADC performs better on small and medium instances;
- FILO performs better than KGLS on medium and large instances, as well as on all the 𝕏 dataset, and it has a similar performance on small instances;
- finally, FILO has a similar performance compared to SISR on small instances, however, SISR performs better on medium, large and on all the 𝕏 dataset.

Table 8 (right) shows a similar analysis comparing FILO (long) with the other methods. In particular,

- FILO (long) performs better than ILS-SP on all partitions of instances;

- FILO (long) performs better than HGSADC on large and on all the $\mathbb{X}$ dataset, and it has a similar performance on small and medium instances;
- FILO (long) performs better than KGLS an all partitions of instances;
- finally, FILO has a similar performance compared to SISR on small, medium and on the whole $\mathbb{X}$ dataset, however, SISR performs better on large instances.

Table 9 Computations on small-sized $\mathbb{X}$ instances.

		ILS-SP		HGSADC		KGLS		SISR		FILO				FILO (long)			
ID	BKS	Avg	$\hat{t}^1$	Avg	$\hat{t}^1$	Avg	$\hat{t}$	Avg	$\hat{t}$	Best	Avg	Worst	t	Best	Avg	Worst	t
X-n101-k25	27591	0.00	0.06–0.07	0.00	0.85–0.91	0.21	2.69	0.00	0.58	0.00	0.00	0.22	1.10	0.00	0.00	0.00	11.89
X-n106-k14	26362	0.05	1.22–1.31	0.08	2.43–2.61	0.19	2.83	0.07	0.95	0.00	0.06	0.11	1.82	0.00	0.02	0.08	19.64
X-n110-k13	14971	0.00	0.12–0.13	0.00	0.97–1.04	0.00	2.94	0.00	0.73	0.00	0.00	0.00	1.63	0.00	0.00	0.00	16.43
X-n115-k10	12747	0.00	0.12–0.13	0.00	1.09–1.17	0.00	3.07	0.00	0.15	0.00	0.00	0.00	1.65	0.00	0.00	0.00	16.94
X-n120-k6	13332	0.04	1.03–1.11	0.00	1.40–1.50	0.00	3.21	0.00	1.16	0.00	0.00	0.00	1.74	0.00	0.00	0.00	18.36
X-n125-k30	55539	0.24	0.85–0.91	0.01	1.64–1.76	0.47	3.34	0.03	2.25	0.00	0.53	1.37	1.24	0.00	0.16	0.61	11.75
X-n129-k18	28940	0.20	1.15–1.24	0.03	1.64–1.76	0.11	3.45	0.03	1.09	0.00	0.06	0.19	1.58	0.00	0.03	0.05	17.72
X-n134-k13	10916	0.29	1.28–1.37	0.17	2.01–2.15	0.00	3.58	0.22	2.04	0.00	0.15	0.30	1.58	0.00	0.06	0.16	16.81
X-n139-k10	13590	0.10	0.97–1.04	0.00	1.40–1.50	0.00	3.72	0.04	1.45	0.00	0.00	0.00	1.98	0.00	0.00	0.00	21.84
X-n143-k7	15700	0.29	0.97–1.04	0.00	1.88–2.02	0.18	3.83	0.04	1.53	0.15	0.17	0.43	1.55	0.00	0.14	0.17	17.78
X-n148-k46	43448	0.01	0.49–0.52	0.00	1.95–2.09	0.35	3.96	0.05	2.04	0.00	0.12	0.35	1.36	0.00	0.01	0.33	15.10
X-n153-k22	21220	0.85	0.30–0.33	0.03	3.34–3.59	0.80	4.10	0.04	4.07	0.02	0.22	0.69	1.66	0.02	0.05	0.34	15.30
X-n157-k13	16876	0.00	0.49–0.52	0.00	1.95–2.09	0.00	4.20	0.02	2.69	0.00	0.00	0.00	2.57	0.00	0.00	0.00	28.74
X-n162-k11	14138	0.16	0.30–0.33	0.02	2.01–2.15	0.06	4.34	0.14	2.47	0.02	0.18	0.23	1.82	0.00	0.09	0.18	18.95
X-n167-k10	20557	0.25	0.55–0.59	0.03	2.25–2.41	0.16	4.47	0.02	2.33	0.00	0.05	0.17	1.91	0.00	0.00	0.02	20.19
X-n172-k51	45607	0.02	0.36–0.39	0.00	2.31–2.48	0.44	4.61	0.03	3.85	0.00	0.01	0.14	1.15	0.00	0.00	0.00	12.65
X-n176-k26	47812	0.92	0.67–0.72	0.30	4.62–4.96	0.39	4.71	0.08	3.78	0.00	0.65	1.25	1.35	0.00	0.19	1.13	14.28
X-n181-k23	25569	0.01	0.97–1.04	0.09	3.83–4.11	0.23	4.85	0.04	4.00	0.00	0.02	0.10	2.22	0.00	0.00	0.01	23.70
X-n186-k15	24145	0.17	1.03–1.11	0.01	3.59–3.85	0.20	4.98	0.14	2.91	0.01	0.10	0.30	1.55	0.00	0.04	0.21	15.94
X-n190-k8	16980	0.96	1.28–1.37	0.05	7.36–7.90	0.33	5.09	0.03	6.62	0.00	0.09	0.44	1.80	0.00	0.02	0.05	18.47
X-n195-k51	44225	0.02	0.55–0.59	0.04	3.71–3.98	0.49	5.23	0.17	4.44	0.00	0.18	0.53	1.25	0.00	0.06	0.26	12.03
X-n200-k36	58578	0.20	4.56–4.89	0.08	4.86–5.22	0.29	5.36	0.10	4.87	0.18	0.68	1.91	1.32	0.08	0.36	0.51	16.17
X-n204-k19	19565	0.31	0.67–0.72	0.03	3.22–3.46	0.52	5.47	0.50	3.56	0.00	0.13	0.70	1.41	0.00	0.01	0.12	14.90
X-n209-k16	30656	0.36	2.31–2.48	0.08	5.23–5.61	0.27	5.60	0.04	4.36	0.00	0.12	0.27	1.31	0.00	0.05	0.12	13.70
X-n214-k11	10856	2.50	1.40–1.50	0.20	6.20–6.66	0.67	5.74	0.48	6.33	0.13	0.43	1.11	1.37	0.04	0.23	0.93	14.22
X-n219-k73	117595	0.00	0.49–0.52	0.01	4.68–5.02	0.09	5.87	0.05	5.82	0.00	0.00	0.01	4.94	0.00	0.00	0.00	54.01
X-n223-k34	40437	0.24	5.17–5.55	0.15	5.05–5.42	0.63	5.98	0.23	5.53	0.08	0.28	0.66	1.22	0.00	0.16	0.27	12.84
X-n228-k23	25742	0.21	1.46–1.57	0.14	5.96–6.39	0.34	6.12	0.19	7.64	0.03	0.21	0.45	1.33	0.00	0.16	0.25	13.84
X-n233-k16	19230	0.55	1.82–1.96	0.30	4.13–4.44	0.58	6.25	0.21	5.89	0.16	0.42	0.71	1.56	0.00	0.30	0.54	17.24
X-n237-k14	27042	0.14	2.13–2.28	0.09	5.41–5.81	0.38	6.36	0.18	5.24	0.00	0.03	0.50	2.13	0.00	0.01	0.16	23.67
X-n242-k48	82751	0.15	10.82–11.61	0.24	7.54–8.09	0.47	6.49	0.16	7.20	0.09	0.31	0.58	1.51	0.00	0.16	0.39	16.81
X-n247-k50	37274	0.63	1.28–1.37	0.03	12.40–13.31	0.17	6.63	0.13	13.38	0.06	0.71	1.30	1.56	0.00	0.46	1.05	16.08
Mean		0.31	1.46 – 1.57	0.07	3.65 – 3.92	0.28	4.66	0.11	3.78	0.03	0.18	0.47	1.69	0.00	0.09	0.25	18.06

[1] computed by considering the range of single-thread rating of compatible CPUs.

Table 10 Computations on medium-sized $\mathbb{X}$ instances.

		ILS-SP		HGSADC		KGLS		SISR		FILO				FILO (long)			
ID	BKS	Avg	$\hat{t}$[1]	Avg	$\hat{t}$[1]	Avg	$\hat{t}$	Avg	$\hat{t}$	Best	Avg	Worst	t	Best	Avg	Worst	t
X-n251-k28	38684	0.40	6.57–7.05	0.29	7.11–7.63	0.60	6.74	0.28	7.13	0.17	0.36	0.66	1.57	0.00	0.25	0.38	17.38
X-n256-k16	18839	0.24	1.22–1.31	0.22	3.95–4.24	0.32	6.87	0.26	8.36	0.22	0.22	0.23	2.13	0.22	0.22	0.22	23.80
X-n261-k13	26558	1.17	4.07–4.37	0.27	7.72–8.29	0.55	7.00	0.32	8.58	0.17	0.48	1.27	1.56	0.01	0.31	0.50	16.98
X-n266-k58	75478	0.11	6.08–6.53	0.37	13.01–13.96	0.65	7.14	0.19	7.86	0.18	0.51	0.87	1.80	0.07	0.38	0.57	20.67
X-n270-k35	35291	0.21	5.53–5.94	0.22	6.81–7.31	0.46	7.25	0.20	8.29	0.05	0.26	0.44	1.40	0.05	0.15	0.27	14.17
X-n275-k28	21245	0.05	2.19–2.35	0.17	7.29–7.83	0.26	7.38	0.11	9.67	0.00	0.08	0.37	1.87	0.00	0.02	0.39	20.23
X-n280-k17	33503	0.80	5.84–6.26	0.31	11.61–12.46	0.61	7.52	0.37	12.87	0.05	0.46	0.76	1.47	0.04	0.36	0.52	15.52
X-n284-k15	20215	1.16	5.23–5.61	0.35	12.10–12.99	0.86	7.62	0.35	11.13	0.15	0.53	1.03	1.58	0.06	0.26	0.53	15.51
X-n289-k60	95151	0.31	9.79–10.51	0.33	12.95–13.90	0.77	7.76	0.21	10.40	0.31	0.60	0.86	1.77	0.24	0.40	0.61	20.07
X-n294-k50	47161	0.20	7.54–8.09	0.21	8.94–9.59	0.61	7.89	0.24	10.69	0.14	0.32	0.59	1.12	0.12	0.23	0.35	11.80
X-n298-k31	34231	0.37	4.19–4.50	0.18	6.63–7.11	0.30	8.00	0.13	10.55	0.00	0.27	0.46	1.18	0.01	0.19	0.31	12.36
X-n303-k21	21738	0.73	8.63–9.27	0.52	10.52–11.29	0.64	8.14	0.18	12.58	0.21	0.45	0.89	1.23	0.09	0.33	0.66	11.91
X-n308-k13	25859	0.94	5.77–6.20	0.14	9.30–9.98	0.82	8.27	1.35	18.69	0.01	0.51	1.83	1.62	0.02	0.46	1.42	18.27
X-n313-k71	94044	0.27	10.64–11.42	0.24	13.62–14.62	0.85	8.41	0.15	13.75	0.29	0.52	0.81	1.39	0.15	0.32	0.57	15.23
X-n317-k53	78355	0.00	5.23–5.61	0.04	13.62–14.62	0.08	8.51	0.05	16.00	0.00	0.01	0.04	3.02	0.00	0.00	0.01	31.68
X-n322-k28	29834	0.53	8.94–9.59	0.41	9.24–9.92	0.68	8.65	0.31	12.29	0.25	0.48	0.88	1.25	0.05	0.34	0.54	13.19
X-n327-k20	27532	1.02	11.61–12.46	0.35	11.06–11.88	0.44	8.78	0.36	15.71	0.17	0.47	0.81	1.71	0.09	0.29	0.55	19.83
X-n331-k15	31102	0.43	9.54–10.24	0.19	14.83–15.92	0.13	8.89	0.08	14.84	0.00	0.02	0.30	2.00	0.00	0.00	0.01	21.59
X-n336-k84	139111	0.25	13.01–13.96	0.30	23.10–24.80	1.40	9.03	0.19	16.58	0.36	0.61	0.93	1.41	0.19	0.38	0.55	13.80
X-n344-k43	42050	0.56	13.74–14.75	0.38	13.19–14.16	0.83	9.24	0.26	15.64	0.28	0.58	0.80	1.35	0.03	0.33	0.56	12.67
X-n351-k40	25896	0.98	15.32–16.44	0.46	20.49–21.99	1.03	9.43	0.33	19.27	0.33	0.67	1.03	1.36	0.26	0.45	0.72	13.06
X-n359-k29	51505	1.11	29.73–31.91	0.42	21.21–22.77	0.94	9.64	0.14	16.80	0.16	0.48	0.85	1.51	0.00	0.24	0.45	16.30
X-n367-k17	22814	0.83	7.96–8.55	0.11	13.37–14.36	0.69	9.86	0.09	26.26	0.00	0.19	0.68	1.65	0.00	0.04	0.15	17.03
X-n376-k94	147713	0.00	4.32–4.63	0.03	17.20–18.47	0.11	10.10	0.05	23.28	0.00	0.01	0.03	4.17	0.00	0.01	0.03	43.35
X-n384-k52	65940	0.66	20.97–22.51	0.50	24.44–26.23	0.70	10.32	0.25	18.84	0.19	0.46	0.74	1.71	0.13	0.27	0.51	18.01
X-n393-k38	38260	0.52	12.64–13.57	0.30	17.39–18.66	0.32	10.56	0.35	22.11	0.10	0.29	0.61	1.41	0.03	0.12	0.37	14.97
X-n401-k29	66187	0.80	36.72–39.41	0.27	30.09–32.30	0.58	10.78	0.09	27.64	0.09	0.22	0.44	1.98	0.02	0.10	0.20	19.96
X-n411-k19	19712	1.23	14.47–15.53	0.16	21.09–22.64	1.79	11.05	0.29	42.48	0.22	0.52	1.49	1.82	0.24	0.37	0.84	18.87
X-n420-k130	107798	0.04	13.49–14.49	0.12	32.34–34.71	0.51	11.29	0.08	34.84	0.08	0.25	0.50	1.16	0.04	0.14	0.26	11.29
X-n429-k61	65467	0.43	23.22–24.93	0.28	25.23–27.08	0.54	11.53	0.19	25.46	0.19	0.39	0.75	1.40	0.01	0.19	0.33	14.55
X-n439-k37	36391	0.14	24.07–25.84	0.17	20.97–22.51	0.31	11.80	0.23	30.62	0.01	0.09	0.25	1.64	0.01	0.02	0.10	17.74
X-n449-k29	55254	1.72	36.41–39.09	0.54	39.45–42.35	0.89	12.07	0.28	27.64	0.34	0.62	1.02	1.46	0.18	0.34	0.67	15.14
X-n459-k26	24145	1.31	36.84–39.54	0.53	26.02–27.93	0.39	12.34	0.40	41.10	0.09	0.31	0.86	1.57	0.00	0.21	0.39	16.51
X-n469-k138	221824	0.16	22.07–23.69	0.36	52.70–56.57	0.73	12.61	0.18	34.91	0.73	1.07	1.36	1.62	0.47	0.64	0.80	16.03
X-n480-k70	89449	0.47	30.64–32.89	0.35	40.73–43.72	0.58	12.90	0.12	36.73	0.15	0.36	0.55	1.63	0.02	0.21	0.41	17.09
X-n491-k59	66487	1.11	31.73–34.06	0.62	43.71–46.92	1.16	13.20	0.24	37.39	0.29	0.53	0.84	1.40	0.12	0.32	0.50	13.88
Mean		0.59	14.05 – 15.09	0.30	18.42 – 19.77	0.64	9.40	0.25	19.64	0.17	0.39	0.75	1.66	0.08	0.25	0.45	17.51

[1] computed by considering the range of single-thread rating of compatible CPUs.

Table 11 Computations on large-sized $\mathbb{X}$ instances.

		ILS-SP		HGSADC		KGLS		SISR		FILO				FILO (long)			
ID	BKS	Avg	$\hat{t}^1$	Avg	$\hat{t}^1$	Avg	$\hat{t}$	Avg	$\hat{t}$	Best	Avg	Worst	t	Best	Avg	Worst	t
X-n502-k39	69226	0.17	49.12–52.72	0.15	38.66–41.50	0.15	13.50	0.07	44.30	0.00	0.04	0.10	2.52	0.00	0.03	0.07	25.92
X-n513-k21	24201	0.96	21.28–22.84	0.40	20.12–21.60	0.36	13.79	0.38	56.08	0.06	0.35	0.86	1.84	0.00	0.15	0.40	19.54
X-n524-k153	154593	0.27	16.60–17.81	0.25	49.06–52.66	0.48	14.09	0.14	110.12	0.08	0.50	0.99	1.38	0.04	0.24	0.56	13.68
X-n536-k96	94868	0.88	37.75–40.52	0.49	65.35–70.15	1.06	14.41	0.32	54.33	0.71	0.83	1.00	1.58	0.53	0.71	0.83	16.02
X-n548-k50	86700	0.20	38.90–41.76	0.34	51.18–54.94	0.25	14.74	0.11	46.91	0.00	0.10	0.25	1.86	0.00	0.05	0.13	19.60
X-n561-k42	42717	0.97	41.88–44.96	0.35	36.84–39.54	0.64	15.09	0.35	53.68	0.21	0.43	0.74	1.32	0.08	0.29	0.54	13.49
X-n573-k30	50673	0.99	68.08–73.08	0.48	114.40–122.80	0.68	15.41	0.26	82.19	0.22	0.37	0.66	1.91	0.15	0.25	0.38	20.00
X-n586-k159	190316	0.32	47.72–51.22	0.27	106.56–114.39	0.41	15.76	0.15	62.77	0.45	0.70	0.98	1.62	0.22	0.37	0.63	16.39
X-n599-k92	108451	0.86	44.38–47.63	0.57	76.53–82.15	0.87	16.11	0.22	54.84	0.30	0.51	0.74	1.68	0.19	0.30	0.45	17.58
X-n613-k62	59545	1.51	45.47–48.81	0.70	71.30–76.54	0.93	16.49	0.31	64.08	0.39	0.67	1.11	1.16	0.06	0.40	0.76	11.35
X-n627-k43	62173	1.18	98.90–106.16	0.56	145.71–156.41	0.71	16.87	0.23	64.95	0.17	0.34	0.54	1.80	0.09	0.20	0.41	18.65
X-n641-k35	63705	1.41	85.35–91.61	0.76	96.53–103.62	0.52	17.24	0.23	67.28	0.13	0.38	0.66	1.88	0.11	0.22	0.45	19.10
X-n655-k131	106780	0.00	28.69–30.80	0.11	91.49–98.20	0.18	17.62	0.06	79.72	0.01	0.04	0.08	3.23	0.00	0.02	0.05	33.44
X-n670-k130	146332	0.92	37.20–39.93	0.61	160.54–172.33	1.00	18.02	0.27	144.67	0.66	1.11	1.70	1.42	0.43	0.86	1.24	14.16
X-n685-k75	68225	1.12	44.86–48.16	0.63	95.25–102.25	1.03	18.43	0.21	98.27	0.44	0.63	0.88	1.42	0.16	0.43	0.67	13.65
X-n701-k44	81923	1.37	127.72–137.09	0.69	153.91–165.22	0.77	18.86	0.17	89.10	0.36	0.53	0.69	1.57	0.06	0.28	0.51	15.91
X-n716-k35	43387	1.81	137.26–147.34	0.59	160.66–172.46	0.89	19.26	0.22	115.14	0.49	0.70	1.06	1.67	0.14	0.28	0.50	17.14
X-n733-k159	136190	0.63	67.84–72.82	0.29	148.63–159.54	0.86	19.72	0.15	104.16	0.18	0.36	0.48	1.26	0.09	0.21	0.31	12.84
X-n749-k98	77314	1.24	77.32–83.00	0.71	190.81–204.82	1.32	20.15	0.25	106.41	0.54	0.71	0.88	1.45	0.28	0.42	0.63	13.90
X-n766-k71	114456	1.12	147.17–157.97	0.60	232.82–249.91	0.84	20.61	0.27	126.85	0.46	0.68	1.07	1.59	0.24	0.43	0.76	15.76
X-n783-k48	72394	1.84	143.16–153.67	0.85	163.94–175.98	0.89	21.07	0.37	123.80	0.34	0.61	0.87	1.75	0.18	0.35	0.57	18.54
X-n801-k40	73331	0.92	262.97–282.28	0.55	175.80–188.71	0.26	21.55	0.14	99.72	0.02	0.23	0.41	1.80	-0.02	0.11	0.26	18.28
X-n819-k171	158121	0.82	90.51–97.16	0.49	227.53–244.24	0.89	22.04	0.19	125.47	0.65	0.84	1.05	1.39	0.44	0.56	0.67	14.31
X-n837-k142	193737	0.67	105.28–113.02	0.38	281.69–302.38	0.76	22.52	0.12	121.32	0.37	0.53	0.70	1.78	0.21	0.31	0.42	18.57
X-n856-k95	88990	0.32	93.43–100.29	0.28	175.31–188.19	0.34	23.03	0.16	116.38	0.02	0.14	0.28	1.76	-0.00	0.06	0.16	18.00
X-n876-k59	99303	1.12	248.80–267.07	0.59	301.14–323.26	0.95	23.57	0.18	158.13	0.32	0.47	0.63	1.74	0.15	0.26	0.38	17.46
X-n895-k37	53928	1.91	249.35–267.66	0.95	195.68–210.05	0.73	24.09	0.29	154.56	0.33	0.58	0.94	1.77	0.02	0.26	0.55	17.93
X-n916-k207	329179	0.54	137.44–147.53	0.31	340.90–365.93	0.58	24.65	0.10	156.60	0.50	0.70	0.85	1.75	0.17	0.39	0.55	18.98
X-n936-k151	132812	1.29	123.10–132.13	0.53	323.09–346.81	0.87	25.19	0.23	300.18	0.39	0.91	1.38	1.32	0.24	0.50	0.79	12.70
X-n957-k87	85469	0.55	189.17–203.06	0.41	263.15–282.47	0.34	25.76	0.18	147.22	0.04	0.14	0.24	1.86	0.00	0.09	0.20	18.93
X-n979-k58	118988	1.06	417.73–448.41	0.43	336.76–361.49	0.63	26.35	0.11	201.19	0.26	0.37	1.02	2.36	0.12	0.23	0.35	23.76
X-n1001-k43	72369	2.23	481.93–517.32	0.81	333.72–358.23	0.95	26.94	0.22	206.79	0.41	0.65	0.83	1.65	0.15	0.33	0.53	16.40
Mean		0.98	118.95 – 127.68	0.50	163.28 – 175.27	0.69	19.47	0.21	110.54	0.30	0.50	0.77	1.72	0.14	0.30	0.49	17.56

[1] computed by considering the range of single-thread rating of compatible CPUs.
New best solutions: (X-n801-k40, 73313);(X-n856-k95, 88989)

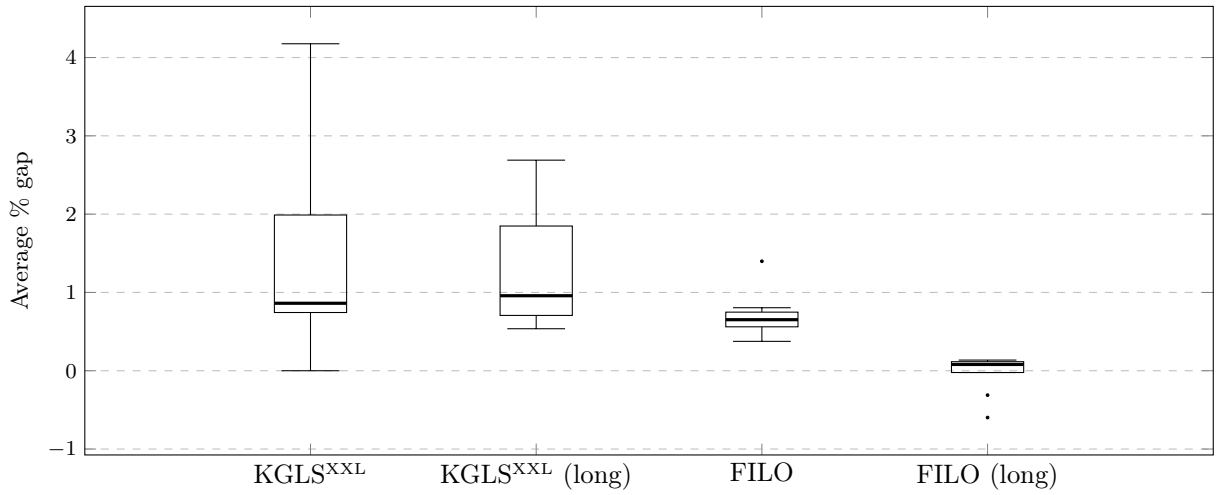


Figure 21 **Comparison of average gaps obtained by algorithms on the $\mathbb{B}$ dataset.**

Table 12 **Computations on the $\mathbb{B}$ dataset: p-values for FILO on the left and for FILO (long) on the right.**

	KGLS$^{\text{XXL}}$	KGLS$^{\text{XXL}}$ (long)
H_0	**0.001953**	0.037109
H_1	**0.000977**	0.018555
	Better	Similar

	KGLS$^{\text{XXL}}$	KGLS$^{\text{XXL}}$ (long)
H_0	**0.001953**	**0.001953**
H_1	**0.000977**	**0.000977**
	Better	Better

p-values in bold are associated with rejected hypothesis when $\alpha = 0.0125$.
The last row contains a p-value interpretation when $\alpha = 0.0125$. In particular, FILO is not statistically different from the competing method when H_0 cannot be rejected (Similar), FILO is statistically better when both H_0 and H_1 are rejected (Better), and, finally, FILO is statistically worse when H_0 is rejected and H_1 is not rejected (Worse).

Appendix D: Computational details for very large-scale instances

This section contains computational details associated with large-scale datasets. In particular, Figures 21 – 23 show by means of boxplots the average gaps obtained by algorithms on the $\mathbb{B}$, $\mathbb{K}$, and $\mathbb{Z}$ dataset, respectively. Average solution values are analyzed by conducting analyses similar to those for the $\mathbb{X}$ dataset described in Section C. In particular, the null hypothesis H_0 and the alternative hypothesis H_1 are the same. However, contrarily to the previous analysis, we did not partition dataset instances in smaller groups. Thus the total number of analysis performed for each dataset is $n = 2$, one for each hypothesis. The initial confidence level $\alpha_0 = 0.025$ is thus adjusted through the Bonferroni correction to $\alpha = 0.025/2 = 0.0125$. Tables 12 and 13 show the p-values associated with the $\mathbb{B}$ and $\mathbb{K}$ datasets, respectively. Finally, due to the very limited number of instances of the $\mathbb{Z}$ dataset, the Wilcoxon signed-rank test cannot be used because it cannot give a significant result.

As can be seen from Table 12

- FILO performs better than KGLS$^{\text{XXL}}$ and it has a performance similar to that of KGLS$^{\text{XXL}}$ (long);
- FILO (long) performs better than KGLS$^{\text{XXL}}$ and KGLS$^{\text{XXL}}$ (long).

As can be seen from Table 13, both FILO and FILO (long) performs better than KGLS$^{\text{XXL}}$ and KGLS$^{\text{XXL}}$ (long).

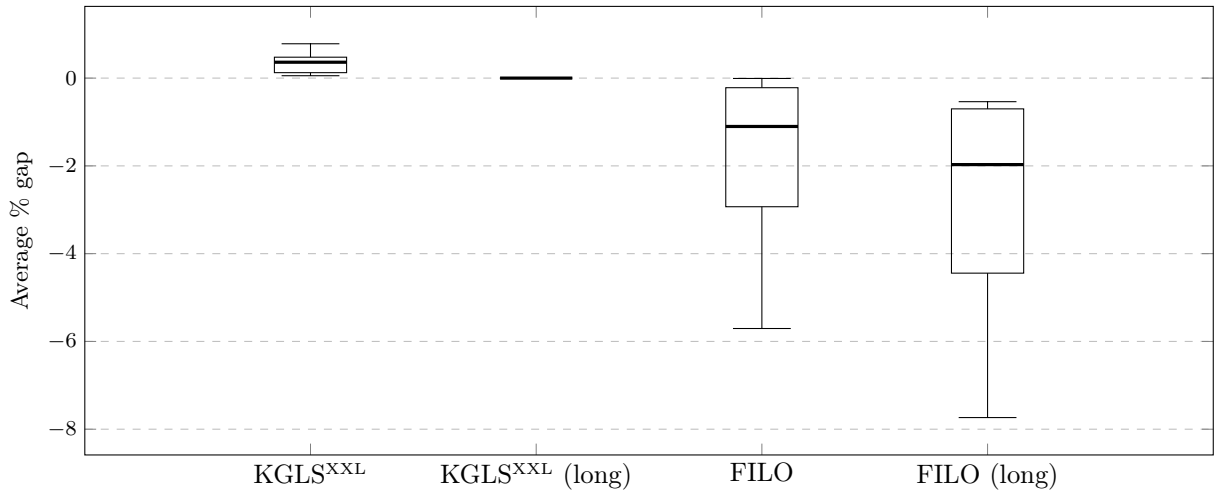


Figure 22 **Comparison of average gaps obtained by algorithms on the $\mathbb{K}$ dataset.**

Table 13 **Computations on the $\mathbb{K}$ dataset: p-values for FILO on the left and for FILO (long) on the right.**

	KGLS$^{\text{XXL}}$	KGLS$^{\text{XXL}}$ (long)
H_0	**0.0078125**	**0.0078125**
H_1	**0.00390625**	**0.00390625**
	Better	Better

	KGLS$^{\text{XXL}}$	KGLS$^{\text{XXL}}$ (long)
H_0	**0.0078125**	**0.0078125**
H_1	**0.00390625**	**0.00390625**
	Better	Better

p-values in bold are associated with rejected hypothesis when $\alpha = 0.0125$.
The last row contains a p-value interpretation when $\alpha = 0.0125$. In particular, FILO is not statistically different from the competing method when H_0 cannot be rejected (Similar), FILO is statistically better when both H_0 and H_1 are rejected (Better), and, finally, FILO is statistically worse when H_0 is rejected and H_1 is not rejected (Worse).

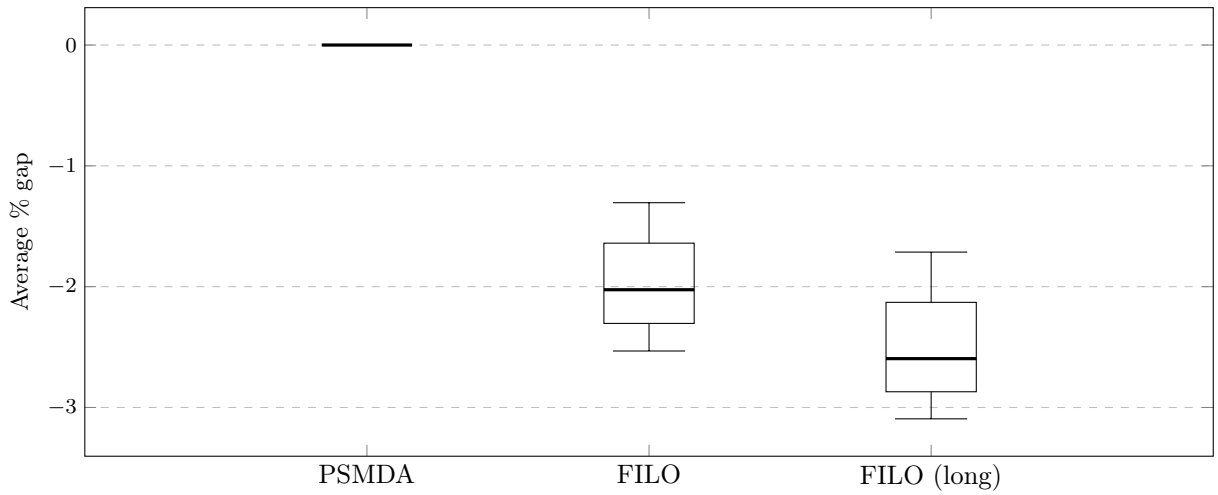


Figure 23 **Comparison of average gaps obtained by algorithms on the $\mathbb{Z}$ dataset.**

References

Accorsi L, Vigo D, 2020 *A hybrid metaheuristic for single truck and trailer routing problems. Transportation Science* 0(Available Online):null, URL http://dx.doi.org/10.1287/trsc.2019.0943.

Arnold F, Gendreau M, Sörensen K, 2019 *Efficiently solving very large-scale routing problems. Computers & Operations Research* 107:32 – 42, URL http://dx.doi.org/10.1016/j.cor.2019.03.006.

Arnold F, Sörensen K, 2019 *Knowledge-guided local search for the vehicle routing problem. Computers & Operations Research* 105:32 – 46, URL http://dx.doi.org/10.1016/j.cor.2019.01.002.

Beek O, Raa B, Dullaert W, Vigo D, 2018 *An efficient implementation of a static move descriptor-based local search heuristic. Computers & Operations Research* 94:1 – 10, URL http://dx.doi.org/https://doi.org/10.1016/j.cor.2018.01.006.

Bergstra J, Bengio Y, 2012 *Random search for hyper-parameter optimization. Journal of Machine Learning Research* 13(10):281–305, URL http://jmlr.org/papers/v13/bergstra12a.html.

Cavaliere F, Bendotti E, Fischetti M, 2020 *Personal communication.*

Christiaens J, Vanden Berghe G, 2020 *Slack induction by string removals for vehicle routing problems. Transportation Science* 54(2):417–433, URL http://dx.doi.org/10.1287/trsc.2019.0914.

Clarke G, Wright JW, 1964 *Scheduling of vehicles from a central depot to a number of delivery points. Operations Research* 12(4):568–581, URL http://dx.doi.org/10.1287/opre.12.4.568.

CVRPLIB, 2020 *Capacitated vehicle routing problem library.* URL http://vrp.galgos.inf.puc-rio.br/index.php/en/, visited on 2020-07-19.

Duarte A, Sánchez-Oro J, Mladenović N, Todosijević R, 2018 *Variable Neighborhood Descent*, 341–367 (Cham: Springer International Publishing), ISBN 978-3-319-07124-4, URL http://dx.doi.org/10.1007/978-3-319-07124-4_9.

Dunn OJ, 1961 *Multiple comparisons among means. Journal of the American Statistical Association* 56(293):52–64, URL http://www.jstor.org/stable/2282330.

Etiemble D, 2018 *45-year cpu evolution: one law and two equations.*

Frank E, Hall MA, Witten IH, 2016 *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"* (Morgan Kaufmann), 4 edition.

Glover F, 1989 *Tabu search—part i. ORSA Journal on Computing* 1(3):190–206, URL http://dx.doi.org/10.1287/ijoc.1.3.190.

Glover F, 1996 *Ejection chains, reference structures and alternating path methods for traveling salesman problems. Discrete Applied Mathematics* 65(1):223 – 253, URL http://dx.doi.org/10.1016/0166-218X(94)00037-E, first International Colloquium on Graphs and Optimization.

Helsgaun K, 2017 *An Extension of the Lin-Kernighan-Helsgaun TSP Solver for Constrained Traveling Salesman and Vehicle Routing Problems: Technical report* (Roskilde Universitet).

Irnich S, Funke B, Grünert T, 2006 *Sequential search and its application to vehicle-routing problems. Computers & Operations Research* 33(8):2405 – 2429, URL http://dx.doi.org/10.1016/j.cor.2005.02.020.

Johnson DS, 1999 *A theoretician's guide to the experimental analysis of algorithms. Data Structures, Near Neighbor Searches, and Methodology.*

Kirkpatrick S, Gelatt CD, Vecchi MP, 1983 *Optimization by simulated annealing. Science* 220(4598):671–680, URL http://dx.doi.org/10.1126/science.220.4598.671.

Kytöjoki J, Nuortio T, Bräysy O, Gendreau M, 2007 *An efficient variable neighborhood search heuristic for very large scale vehicle routing problems. Computers & Operations Research* 34(9):2743 – 2757, URL http://dx.doi.org/10.1016/j.cor.2005.10.010.

Lourenço HR, Martin OC, Stützle T, 2003 *Iterated Local Search*, 320–353 (Boston, MA: Springer US), ISBN 978-0-306-48056-0, URL http://dx.doi.org/10.1007/0-306-48056-5_11.

Martello S, Toth P, 1990 *Knapsack Problems: Algorithms and Computer Implementations* (USA: John Wiley & Sons, Inc.), ISBN 0471924202.

Matsumoto M, Nishimura T, 1998 *Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul.* 8(1):3–30, URL http://dx.doi.org/10.1145/272991.272995.

Mladenović N, Hansen P, 1997 *Variable neighborhood search. Computers & Operations Research* 24(11):1097 – 1100, URL http://dx.doi.org/https://doi.org/10.1016/S0305-0548(97)00031-2.

PassMark® Software, 2020 *Professional cpu benchmarks.* URL https://www.passmark.com/index.html, visited on 2020-07-19.

R Core Team, 2020 *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, URL https://www.R-project.org/.

Reinelt G, Rinaldi G, 1994 *The traveling salesman problem.*

Schneider M, Schwahn F, Vigo D, 2017 *Designing granular solution methods for routing problems with time windows. European Journal of Operational Research* 263(2):493–509, URL http://dx.doi.org/10.1016/j.ejor.2017.04.059.

Schrimpf G, Schneider J, Stamm-Wilbrandt H, Dueck G, 2000 *Record breaking optimization results using the ruin and recreate principle. J. Comput. Phys.* 159(2):139–171, URL http://dx.doi.org/10.1006/jcph.1999.6413.

Subramanian A, Uchoa E, Ochi LS, 2013 *A hybrid algorithm for a class of vehicle routing problems. Computers & OR* 40(10):2519 – 2531, URL http://dx.doi.org/10.1016/j.cor.2013.01.013.

Taillard É, Badeau P, Gendreau M, Guertin F, Potvin JY, 1997 *A tabu search heuristic for the vehicle routing problem with soft time windows. Transportation Science* 31(2):170–186, URL http://dx.doi.org/10.1287/trsc.31.2.170.

Toth P, Vigo D, 2003 *The granular tabu search and its application to the vehicle-routing problem. INFORMS Journal on Computing* 15(4):333–346, URL http://dx.doi.org/10.1287/ijoc.15.4.333.24890.

Uchoa E, 2020 *Personal communication.*

Uchoa E, Pecin D, Pessoa A, Poggi M, Vidal T, Subramanian A, 2017 *New benchmark instances for the capacitated vehicle routing problem. European Journal of Operational Research* 257(3):845 – 858, URL http://dx.doi.org/10.1016/j.ejor.2016.08.012.

Vidal T, Crainic TG, Gendreau M, Lahrichi N, Rei W, 2012 *A hybrid genetic algorithm for multidepot and periodic vehicle routing problems. Operations Research* 60(3):611–624, URL http://dx.doi.org/10.1287/opre.1120.1048.

Voudouris C, Tsang E, 1999 *Guided local search and its application to the traveling salesman problem. European Journal of Operational Research* 113(2):469 – 499, URL http://dx.doi.org/10.1016/S0377-2217(98)00099-X.

Wilcoxon F, 1945 *Individual comparisons by ranking methods. Biometrics Bulletin* 1(6):80–83, URL http://www.jstor.org/stable/3001968.

Zachariadis EE, Kiranoudis CT, 2010 *A strategy for reducing the computational complexity of local search-based methods for the vehicle routing problem. Comput. Oper. Res.* 37(12):2089–2105, URL http://dx.doi.org/10.1016/j.cor.2010.02.009.