Routing one million customers in a handful of minutes

Routing One Million Customers in a Handful of Minutes

Luca Accorsi[1] and Daniele Vigo[2, 3]

[1]*Google*
[2]*Department of Electrical, Electronic and Information Engineering "G. Marconi", University of Bologna, Italy*
[3]*CIRI ICT, University of Bologna, Italy*

accorsi@google.com
daniele.vigo@unibo.it

Abstract

In this paper, we propose a new dataset of Capacitated Vehicle Routing Problem instances which are up to two orders of magnitude larger than those in the currently used benchmarks. Despite these sizes might not have an immediate application to real-world logistic scenarios, we believe they could foster fresh new research efforts on the design of effective and efficient algorithmic components for routing problems. We provide computational results for such instances by running FILO2, an adaptation of the FILO algorithm proposed in Accorsi and Vigo (2021), designed to handle extremely large-scale CVRP instances. Solutions for such instances are obtained by using an a standard personal computer in a considerably short computing time, thus showing the effectiveness of the acceleration and pruning techniques already proposed in FILO. Finally, results of FILO2 on well-known literature instances show that the newly introduced changes improve the overall scalability of the approach with respect to the previous FILO design.

Keywords— Capacitated Vehicle Routing Problem, Metaheuristics, Large-Scale Instances

1 Introduction

The Capacitated Vehicle Routing Problem (CVRP, see Toth and Vigo, 2014) is a challenging and practically relevant combinatorial optimization problem that, since its proposal in Dantzig and Ramser (1959), has attracted a huge research effort for the development of efficient solution methods. Algorithmic advancements and increase in raw computing power made possible the heuristic resolution of larger and larger problem instances. Most of the existing state-of-the-art algorithms for the CVRP are evaluated by using the well-known $\mathbb{X}$ dataset proposed by Uchoa et al. (2017), which include instances with up to 1,000 customers, as has also happened for the recent DIMACS (2022) implementation challenge. To the best of our knowledge, the $\mathbb{B}$ dataset proposed by Arnold, Gendreau, and Sörensen (2019) contains the largest literature instances, including up to thirty thousand customers. Such large instances may have practical applications, indeed, the authors based their construction on real-world parcel distribution problems in Belgium.

Devising algorithms able to scale up to the size of the $\mathbb{B}$ instances and far beyond, other than being a mere research exercise, may promote new research ideas also applicable to instances with a smaller scales. For this reason, in this paper we propose what we call extremely large (XXL) instances, with up to one million customers, as a new additional benchmark for future research on large-scale CVRP. We also provide benchmark results obtained by an evolution of the FILO method proposed in Accorsi and Vigo (2021) explicitly designed to handle XXL instances.

The CVRP is defined on an undirected complete graph $G = (V, E)$ where V is the vertex set and E is the edge set. The set of vertices V is partitioned into $V = \{0\} \cup V_c$ where 0 is the depot and $V_c = \{1, 2, \ldots, N\}$ is a set of N customers. A cost c_{ij} is associated with each edge $(i, j) \in E$ and we assume that the cost matrix $\boldsymbol{c}$ satisfies the

triangle inequality. For a subset $V' \subseteq V$, we identify with $\mathcal{N}_i^k(V')$ the ordered list of the k nearest neighbor vertices $j \in V'$ of vertex i with respect to the cost matrix $\boldsymbol{c}$. Whenever $k = |V'|$, we refer to the ordered list of neighbors of i by omitting the apex k, i.e., $\mathcal{N}_i(V')$. Each customer $i \in V_c$ requires an integer quantity $q_i > 0$ of homogeneous goods from the depot, and for the depot we have $q_0 = 0$. 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 made up of a number $|S|$ of Hamiltonian circuits, called routes, starting from the depot, visiting a subset of customers, and coming back to the depot. A solution is feasible if all customers are visited exactly once and the load of all the used vehicles does not exceed the capacity Q. The cost of a solution S is given by the sum of the costs of the edges defining the routes of S and the goal of the CVRP is to find the feasible solution with the minimum cost.

Successful approaches to large-scale CVRP instances typically make use of acceleration and pruning techniques to speed up their optimization process. More specifically, these techniques often support the execution of the local search, which is typically one of the most computationally expensive procedure of a heuristic solution approach. Among the most popular and successful acceleration and pruning techniques we have Granular Neighborhoods (GNs), Static Move Descriptors (SMDs) and Sequential Search (SS). GNs, originally proposed in Toth and Vigo (2003), make use of the fact that for most local search operators, neighboring solutions can be generated by a local search move uniquely determined by an arc $(i, j) \in E$, called the *move generator*. Thus, instead of exploring a complete neighborhood, a sparsification rule is used to specify which arcs are used to generate local search moves. The computational complexity of such a neighborhood exploration can be arbitrarily reduced according to how move generators are defined. The right amount of move generators, defined by proper sparsification rules, experimentally shown to provide a good tradeoff between local search execution time and solution quality. In the SMD framework proposed by Zachariadis and Kiranoudis (2010), the classic for-loop exploration of neighborhoods is replaced with a structured inspection of data structures, called SMDs, that identify a move and its effect, in terms of cost change to the current solution. The use of SMDs makes neighborhood exploration much more efficient, because whenever a change is applied to the current solution it is possible to evaluate a smaller neighborhood affected by the newly applied change, by updating a subset of the existing SMDs, rather than performing a complete reevaluation of the whole neighborhood. GNs and SMDs have been successfully employed in several algorithms, such as Toth and Vigo (2003), Zachariadis and Kiranoudis (2010), Schneider, Schwahn, and Vigo (2017), Accorsi and Vigo (2020), and Accorsi and Vigo (2021). Finally, the SS proposed by Irnich, Funke, and Grünert (2006) decomposes a local search move into basic blocks called *partial moves*. The execution of these partial moves can be aborted if certain conditions are met, thus pruning in advance a non-promising local search move. This technique was successfully used in Arnold and Sörensen (2019) and Arnold, Gendreau, and Sörensen (2019).

Decomposition techniques are another tool for dealing with large-scale instances. Their idea relates to the divide-and-conquer principle. First, the original problem is split into several subproblems which are generally considerably smaller than then original problem. The subproblems are then solved efficiently, and their solutions are combined together to produce an overall solution to the original problem. Santini et al. (2023) provide a systematic characterization of decomposition techniques for vehicle routing heuristics and show how the Hybrid Genetic Search, a state-of-the-art algorithm for a broad class of vehicle routing problems (see Vidal et al. (2012) and Vidal (2022)), could benefit from them when approaching large-scale instances.

Algorithm FILO proposed in Accorsi and Vigo (2021) takes, on the other hand, a different approach to cope with large-scale instances. It employs the above mentioned GNs and SMDs, but instead of explicitly decomposing the instance, it relies on the localized optimization of a very limited and dynamically defined solution area. This is accomplished through a heuristic pruning technique called Selective Vertex Caching (SVC) that consists in using a cache-like data structure to keep track of a limited set of vertices involved in recent solution changes.

More specifically, we can analyze the SVC functioning in FILO by taking a look at the neighbor solution generation occurring during its core optimization phase. First, a neighbor solution S' is obtained by copying the current working solution S. At this point the SVC of S' is empty since no action has ever been performed before directly to S'. A shaking procedure executed in a ruin-and-recreate fashion is applied to S'. During the shaking, a set of vertices is removed and then reinserted with a greedy algorithm. Every removal and insertion causes affected vertices to be

inserted into the SVC. A subsequent local search procedure is applied by exploring local search moves generated by move generators (i, j) such that at least one between i and j belong to the SVC. The net effect is that the neighbor solution generation is largely independent from the actual instance size but rather more affected by several algorithm parameters such as the SVC maximum capacity, the ruin intensity and the number of move generators considered from every vertex used during the local search. Indeed, results proposed in Accorsi and Vigo (2021) show that approximately the same computing time to solve instances with 100 customers is required to solve instances with ten times more customers.

We shall however also note that a major drawback of an SVC approach compared to an explicit decomposition into independent subproblems is given by the inherent sequentiality of the former. Indeed, if several independent subproblems resulting from the decomposition can be easily solved in parallel without synchronization efforts, the implicit decomposition offered by the SVC makes parallelization a much more challenging task.

The contributions of this paper are the following:

- We introduce a dataset of XXL instances obtained by projecting to the Euclidean plane the geographical coordinates of existing addresses in various Italian regions. The dataset contains instances with a number of customers ranging from 20 thousand to 1 million.
- We develop FILO2 as an evolution of FILO to show the scalability properties of the techniques already proposed in Accorsi and Vigo (2021) as well as some limitations of the original algorithm. In fact, as discussed in Section 2, as we move towards much larger instances, given the localization of the neighbor solution generation, other procedures become the actual bottleneck of the approach and some changes are required for the overall algorithm to retain its efficiency.
- We provide computational results of FILO2 on existing literature instances, showing that it matches the quality of FILO and greatly improves its scalability, and on the newly introduced instances, showing how FILO2 is able to tackle such XXL instances within an ordinary computing system.

The paper is structured as follows. Section 2 describes FILO2, highlighting the differences with respect to FILO. Section 3 provides the experimental results, and in Section 4 we provide some analysis on the behaviour of the algorithm. Finally, the Appendix contains supplemental material about detailed computational results and statistical analysis.

2 Solution Approach

The proposed solution approach, called FILO2, is an evolution of FILO designed to handle XXL instances of the CVRP. Our goal is indeed showing that the original FILO algorithm, that was among the best performing algorithms on very large-scale instances during the 12th DIMACS implementation challenge (see DIMACS (2022)), is already capable of tackling much larger instances provided that relatively minor adjustments are performed.

In the following sections, we first provide an introductory description of FILO, then we move to a detailed description of FILO2, by highlighting the main changes with respect to the original algorithm along with their motivation.

2.1 The FILO Metaheuristic

Algorithm FILO is a randomized, effective, and efficient algorithm specifically designed for large-scale instances of the CVRP. Algorithm 1 and the following paragraphs provide an high-level overview of the approach. Furthermore, Table 1 provides a reference for most of the parameters used in FILO and mentioned in this overview. A thorough description of the original approach is beyond the scope of this paper, we thus refer the interested reader to Accorsi and Vigo (2021) for more details about its design.

The algorithm begins with a construction phase that builds an initial feasible solution with a restricted version of the savings algorithm (see Clarke and Wright (1964)) proposed by Arnold, Gendreau, and Sörensen (2019). More

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
```

Input parameters: instance $\mathcal{I}$, seed s

precisely, instead of computing all possible savings, which grow quadratically with the instance size, only a linear number of them are computed. Such a limited savings algorithm has experimentally shown to generate, for medium to large-scale instances, initial solutions with a quality comparable to that of a complete savings algorithm. Once an initial solution is available, an improvement phase is executed. During this phase a route minimization procedure is optionally performed if it is estimated that more routes than necessary are employed. The estimation of the ideal number of routes k is performed by greedily solving the bin packing problem associated with the CVRP instance with a simple first-fit algorithm (see, e.g., Chapter 8 of Martello and Toth (1990)). Then, a core optimization procedure, which is the main procedure of FILO, is executed.

Both the route minimization and the core optimization procedures are based on the Iterated Local Search paradigm (ILS, Lourenço, Martin, and Stützle (2003)), in which shaking and local search applications interleave for a fixed number of iterations. In both procedures, the shaking is performed in a *ruin-and-recreate* fashion (see Schrimpf et al. (2000)), and the local search makes use of GNs to heuristically prune unpromising moves, and SMDs to avoid unnecessary neighborhood reevaluations. Moreover, the SVC is always used to keep the optimization mainly localized to the specific area that has just been affected by the shaking procedure. In particular, a vertex i is inserted into the SVC whenever an operation involving i is performed on the current solution. As an example, the removal of vertex i from its route r_i would insert into the SVC i itself, as well as the predecessor π_i and the successor σ_i of i in r_i. The maximum cache capacity is defined by a parameter C and is managed with a first-in-first-out policy: whenever a new vertex should be inserted when the cache is full, the least recently inserted vertex is removed.

Table 1: Parameters of Algorithm FILO mentioned in the overview.

Initial solution definition	
$n_{cw} = 100$	Number of neighbors considered in the savings computation of the construction phase.
$\Delta_{RM} = 10^3$	Maximum number of route minimization iterations.
Granular neighborhood	
$n_{gs} = 25$	Number of neighbors considered by the sparsification rule.
$\gamma_{base} = 0.25$	Base sparsification factor, i.e., at least $n_{gs} \cdot \gamma_{base}$ move generators are always considered for every vertex.
$\delta = 0.5$	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	
$\Delta_{CO} = 10^5, 10^6$	Number of core optimization iterations for short and long runs, respectively.
$\mathcal{T}_0, \mathcal{T}_f$	Initial and final simulated annealing temperature.
$C = 50$	Maximum capacity of the SVC.
$\omega_{base} = \lceil \ln \lvert V \rvert \rceil$	Initial shaking intensity.

Several local search operators such as CROSS-exchange (see Taillard et al. (1997)), 2-opt (see Reinelt and Rinaldi (1994)) variants, and ejection chain (see Glover (1996)) are explored in a variable neighborhood descent fashion (VND, see Mladenović and Hansen (1997)) and concur in making a powerful optimization action.

The route minimization procedure may visit infeasible solutions to perform its route compacting action, while the core optimization procedure explores only the feasible space and achieves its diversification by means of an effective simulated annealing acceptance rule (SA, see Kirkpatrick, Gelatt, and Vecchi (1983)).

2.1.1 Route Minimization

Algorithm 2 Route minimization procedure

```
 1: procedure RouteMin(I, S*, k, R)
 2:     S ← S*, P ← 1, L ← [ ]
 3:     for n ← 1 to Δ_RM do
 4:         (r_i, r_j) ← PickPairOfRoutes(S, R)
 5:         S ← S \ r_i \ r_j
 6:         L ← [L, CustomersOf(r_i, r_j)]
 7:         L̄ = [ ]
 8:         for i ∈ Ordered(L, I, R) do
 9:             p ← BestInsertionPositionInExistingRoutes(i, S)
10:             if p ≠ none then
11:                 S ← Insert(i, p, S)
12:             else
13:                 if |S| < k ∨ U(0, 1) > P then
14:                     S ← BuildSingleCustomerRoute(i, S)
15:                 else
16:                     L̄ ← [L̄, i]
17:                 end if
18:             end if
19:         end for
20:         L ← L̄
21:         S ← LocalSearch(S, R)
22:         if |L| = 0 ∧ (Cost(S) < Cost(S*) ∨ (Cost(S) = Cost(S*) ∧ |S| < |S*|)) then
23:             S* ← S
24:             if |S*| ≤ k then return S*
25:         end if
26:         P ← Decrease(P)
27:         if Cost(S) > Cost(S*) then S ← S*
28:     end for
29:     return S*
30: end procedure
```

Input parameters: instance $\mathcal{I}$, solution S^*, route estimate k, random engine $\mathcal{R}$

The route minimization procedure, whose pseudocode is shown in Algorithm 2, is optionally applied to the initial solution S generated by the construction phase.

Initially, the working solution S is set to be equal to the current best found solution S^* originally obtained by the construction phase (line 2). Then, for a prefixed number Δ_{RM} of iterations, the following main steps are performed: (i) a pair of routes (r_i, r_j) and their associated customers are removed from S (lines 4 and 5), (ii) customers are greedily reinserted, after been ordered according to certain criteria, in the resulting partial solution S in the position that minimizes their insertion cost (lines 8 to 19), and finally, (iii) a local search is applied to S (line 21).

The key procedure idea is that whenever an insertion position for a customer cannot be found within the existing routes, i.e., all the possible insertion positions would violate the route feasibility by requiring a total route load greater than the maximum vehicle capacity Q, such customer can probabilistically remain unserved for the current iteration (lines 13 to 17). When this happens, in the subsequent iteration step (ii) would also consider these unserved customers. The probability of leaving a customer unserved decreases over the iterations by following an exponential

schedule, so that in later iterations customers are less likely to be left unserved. Whenever, after the local search execution in step (iii), the solution S results to be better than S^*, the best found solution S^* is replaced with S (lines 22 to 25). The comparison between S and S^* happens through the evaluation of their costs and number of routes. In particular, S is considered to be better than S^* whenever it has a lower cost, or in the unlikely case in which it has the same cost but a lower number of routes. This criterion follows the CVRP definition where a better objective function is always preferred over a lower number of routes. Nevertheless, the route minimization procedure experimentally proved to be both fast, requiring only few seconds for medium and large instances, and on average beneficial in terms of final solution quality for those instances to which it was applied.

2.1.2 Core Optimization

Algorithm 3 Core optimization procedure

```
 1: procedure CoreOpt(S*, R)
 2:     S ← S*, T ← T_0
 3:     ω ← (ω_1, ω_2, ..., ω_|V_c|), ω_i ← ω_base ∀i ∈ V_c
 4:     γ ← (γ_0, γ_1, ..., γ_|V_c|), γ_i ← γ_base ∀i ∈ V
 5:     for n ← 1 to Δ_CO do
 6:         S' ← S
 7:         S' ← Shaking(S', R, ω)
 8:         S' ← LocalSearch(S', R)
 9:         if Cost(S') < Cost(S*) then
10:             S* ← S'
11:             ResetSparsificationFactors(γ)
12:         else
13:             UpdateSparsificationFactors(γ)
14:         end if
15:         UpdateShakingParameters(ω, S', S, R)
16:         if AcceptNeighbor(S, S', T) then S ← S'
17:         T ← c · T
18:     end for
19:     return S*
20: end procedure
```

Input parameters: solution S^*, random engine $\mathcal{R}$

The core optimization, whose pseudocode is shown in Algorithm 3, is an iterative procedure performed for a number Δ_{CO} iterations that is the main actor of the solution improvement in FILO.

As an initial setup step, the working solution S is set to be equal to current best solution S^* defined by the initial construction phase and possibly optimized by the route minimization procedure (line 2). Then, during every iteration, a neighbor solution S' is generated through the application of shaking and local search procedures to solution S (lines 6 to 8). Solution S^* is replaced by S' if the latter has a lower cost (line 10). Moreover, S' may replace S based on a classical simulated annealing criterion (line 16).

The shaking procedure is performed in a ruin-and-recreate fashion. More precisely, the ruin step removes a number ω_i of customers, starting from a randomly selected seed customer i. This is achieved by performing a random walk rooted in i, of length ω_i on the partial graph representing the solution and by removing every visited customer. The recreate step reinserts the removed customers in the position that minimizes their insertion cost after they have been sorted according to certain criteria (i.e., either by increasing or decreasing distance from the depot, or by decreasing demand value). The vector $\boldsymbol{\omega} = (\omega_1, \ldots, \omega_{|V_c|})$ defines a per-customer ruin intensity, whose value is initially set to a $\omega_{base} = \lceil \ln |V| \rceil$ and then iteratively updated, for all customers i involved in the ruin step, according to the quality of the resulting neighbor solution S' with respect to the source solution S.

The SVC is empty at beginning of every iteration before the ruin application (line 6) since no action has been performed to S' yet. The shaking application causes some vertices involved in the ruin-and-recreate procedures to be inserted into the SVC. The actual cached vertices are defined by the order in which vertices are processed and by the SVC maximum capacity. Because the explored local search moves are identified by move generators (i, j) such that

at least a vertex between i or j is included in the SVC, the local search will mainly optimize the solution area affected by the shaking. This makes every core optimization iteration extremely efficient and largely independent from the actual instance size.

The local search intensity is guided by a vertex-wise sparsification level $\boldsymbol{\gamma} = (\gamma_0, \ldots, \gamma_{|V_c|})$ that is a percentage determining, for each vertex $i \in V$, the number of move generators used during a local search application involving i. The complete set of move generators is defined to be $T = \cup_{i \in V}\{(i,j),(j,i) \in E : j \in \mathcal{N}_i^{n_{gs}}(V \setminus \{i\})\}$, that is, for every vertex i, the arcs connecting i to the n_{gs} neighbors j. Parameters γ_i are used to select a fraction, in increasing arc cost, of these move generators for every vertex i. At the beginning of the procedure, these parameters are initialized to $\gamma_i = \gamma_{base}, i \in V$. Whenever a solution S' is found to improve the best known solution S^*, parameters γ_i for vertices i involved the last local search application are reset to γ_{base} (line 11). On the other hand, if a vertex i is involved in a certain number of consecutive unsuccessful local search applications, the value of γ_i is set to be $\gamma_i = min\{\gamma_i \cdot \lambda, 1\}$, where λ is an increment factor (line 13). Vertices i for which γ_i is updated are those into the SVC after the local search application.

2.2 The FILO2 Metaheuristic

Algorithm FILO2 extends FILO to support XXL instances. In particular, it mostly removes the nonlinear (and some linear) procedures still existing within the original algorithm, and limits the memory occupation of its data structures. The following sections describe in detail the implemented changes, by highlighting the differences between FILO and FILO2.

2.2.1 Cost Matrix Storage

Before starting the actual optimization process, the original FILO algorithm performs an initial instance preprocessing during which some data structures are computed once, to be then used throughout the whole algorithm. In particular, during this initial instance preprocessing, all costs c_{ij} for $i, j \in V$ are computed and stored in the cost matrix $\boldsymbol{c}$. Such a quadratic memory occupation is no longer feasible when moving to XXL instances. For this reason in FILO2, the cost matrix $\boldsymbol{c}$ is never explicitly stored but costs are computed on-demand whenever required.

In addition, every solution object keeps track of the costs of the arcs composing it, i.e., if j is served after i, the cost c_{ij} is associated with vertex j and identified by $\hat{c}_j$. This allows to access in constant-time most of the costs used during the algorithm procedures. As an example, consider a relocate local search move removing vertex i from its current position and inserting it before vertex j. By denoting with π_i and σ_i the predecessor and successor of vertex i respectively, the solution cost variation of this change is given by $c_{\pi_i\sigma_i} - c_{\pi_i i} - c_{i\sigma_i} + c_{\pi_j i} + c_{ij} - c_{\pi_j j}$. Half of the required costs, and more precisely those describing pairs of consecutive vertices $c_{\pi_i i}$, $c_{i\sigma_i}$, and $c_{\pi_j j}$ can be retrieved in constant-time from the current solution by accessing $\hat{c}_i$, $\hat{c}_{\sigma_i}$, and $\hat{c}_j$. The decision of associating the cost c_{ij} to vertex j is completely arbitrary, indeed c_{ij} could have been associated with i as well. Moreover, associating to each vertex i the cost $c_{\pi_i i}$ and not also the cost $c_{i\sigma_i}$ is enough for the access pattern of FILO2.

Furthermore, since every local search operator is designed according to the GN paradigm, a local search move identified by move generator (i,j) will always try to insert arc (i,j) in solution. The arc costs c_{ij} can thus be associated with move generator (i,j) allowing for an additional constant-time cost retrieval. Thus, in the above relocate example, only two out of six costs need to be computed in practice. Finally, since the CVRP is associated with a symmetric cost matrix, a single copy between c_{ij} and c_{ji} can be associated with both move generators (i,j) and (j,i) halving the memory requirements.

2.2.2 Neighbor Lists Storage

In FILO, during the above-mentioned initial preprocessing, also neighbor lists $\mathcal{N}_i(V)$ for every $i \in V$ are fully populated, i.e., for every vertex $i \in V$, vertices $j \in V$ are sorted according to their distance to i and stored in increasing order. As for the cost matrix storage, storing all neighbors for all vertices would require a quadratic memory occupation and it is thus not feasible from a memory perspective when solving XXL instances. Moreover,

completely sorting all vertices to populate these lists is also not acceptable from a computing time perspective since it would require $|V|^2 log|V|$ operations.

We shall, however, note that such neighbor lists are very important data structures that are used in key procedures of the algorithm. In particular, they support the following ones.

- **The construction phase based on the savings algorithm.** As mentioned in Section 2.1, not all savings are computed, but only a limited number of them. More precisely, for every customer $i \in V_c$, only savings s_{ij} with $j \in \mathcal{N}_i^{n_{cw}}(V_c)$ are computed. Therefore, for every customer i at least the n_{cw} nearest customers j need to be known to maintain the same construction algorithm proposed in FILO. The parameter n_{cw} was originally set to 100 because the experimental analysis showed that such value achieves a good trade-off between computing time and initial solution quality.
- **The ruin step of the core optimization and route minimization shaking procedures.** During the ruin step of the core optimization procedure, a random walk of prefixed length is performed by starting from a random seed customer. Whenever a customer is visited during the walk, it is removed from the solution to be later reinserted during the recreate step (see Section 2.2.3). A random walk ending at customer i of route r_i may be extended either by moving within the same route, thus visiting the predecessor or the successor of i, or by jumping to a neighbor route possibly different from r_i. Such a neighbor route is identified by scanning customers $j \in \mathcal{N}_i(V_c)$ and searching for a customer j served by route r_j possibly such that $r_i \neq r_j$.
  Similarly, during the ruin step of the route minimization procedure, two close routes are selected, destroyed, and all their customers removed from the current solution. The first route is identified by randomly selecting a seed customer i, then the second route is again identified by searching customers $j \in \mathcal{N}_i(V_c)$ for one such that $r_i \neq r_j$.
- **The move generators definition for GNs.** The set of move generators T used to perform the local search procedures is defined to contain all arcs connecting a vertex $i \in V$ to its $n_{gs} = 25$ nearest vertices, i.e., $T = \cup_{i \in V}\{(i,j),(j,i) \in E : j \in \mathcal{N}_i^{n_{gs}}(V \setminus \{i\})\}$.

In FILO2, the neighbor lists are restricted to contain a limited number n_{nn} of vertices, i.e., for every vertex i, only $\mathcal{N}_i^{n_{nn}}(V)$ is computed and stored during the initial instance preprocessing. These vertices are efficiently identified by exploiting a *kd*-tree data structure built on top of vertex coordinates (see Bentley (1990)). As described in Section 3.2 and analyzed in Section 4.1, the value for n_{nn} is set to be $n_{gs} < n_{cw} \leq n_{nn} \ll |V|$ to keep both the construction phase and the move generators definition as in the original algorithm. As far as the ruin step of the route minimization and core optimization procedures is concerned, the value of n_{nn} is large enough to make sure these procedures are almost always able to find nearest vertices satisfying the required conditions, e.g., belonging to a route different from the current one. In the unlikely case in which this is not possible, the ruin procedure is simply prematurely concluded, e.g., the random walk is not extended in the core optimization procedure or a single route is removed in the route minimization procedure.

2.2.3 Recreate Strategy

In the original FILO algorithm, the recreate step of the core optimization and route minimization shaking procedures takes all the customers removed by the ruin step, sorts them according to a certain criterion (distance from the depot or customer demand), and greedily reinserts them in the best possible position, i.e., the position that currently minimizes the insertion cost. When such a position cannot be found in the existing routes because all candidate positions would violate the capacity constraints, a new single-customer route is created in the core optimization procedure or the customer may probabilistically remain unserved in the route minimization procedure.

When moving to XXL instances, fully scanning a partial solution searching for the best insertion position is computationally too expensive. Thus, in FILO2, for every removed customer i, only routes r_j that serve neighbor customers $j \in \mathcal{N}_i^{n_{nn}}(V)$ are scanned searching for candidate insertion positions.

2.2.4 Arc Cost Sampling

The search trajectory of FILO is ruled by a neighbor acceptance criterion based on the SA paradigm. In particular, during the core optimization procedure shaking and local search are applied to the current working solution S to generate a neighbor solution S'. Solution S' will replace solution S in the subsequent core optimization iteration if $\text{COST}(S') < \text{COST}(S) + \mathcal{T} \cdot \ln U(0,1)$, where $\mathcal{T}$ is the current temperature. The value of $\mathcal{T}$ is bounded between an initial and final temperature $\mathcal{T}_0$ and $\mathcal{T}_f$, respectively, and it is updated at every core optimization iteration according to $\mathcal{T} = c \cdot \mathcal{T}$, where $c = (\mathcal{T}_f/\mathcal{T}_0)^{(1/\Delta_{CO})}$ and Δ_{CO} is the total number of core optimization phase iterations. The initial and final temperatures $\mathcal{T}_0$ and $\mathcal{T}_f$ are defined to be proportional to the average cost d of an arc of the instance. More precisely $\mathcal{T}_0 = 0.1 \cdot d$ with $d = \sum_{i,j \in V: i<j} c_{ij}/(|V| \cdot (|V|-1)/2)$, and $\mathcal{T}_f = 0.01 \cdot \mathcal{T}_0$. This was done to adapt the algorithm to instances having considerably different arc costs.

In FILO2, we replace d with an estimation $\tilde{d}$ based on a limited random sampling of a number of $|V|$ arcs. We observed a minor difference between the exact and estimated average arc cost for most of the instances. Moreover, this difference becomes even smaller for the SA initial temperature since the average arc cost is down-scaled by a factor of ten.

2.2.5 Solution Copies

Every core optimization iteration of FILO starts from an initial solution S and produces a neighbor solution S' obtained by the shaking and local search applications. As described in Section 2.2.4, solution S' could replace the current working solution S if the SA acceptance criterion is satisfied (line 10 of Algorithm 3). Moreover, S' could replace the best solution S^* in case it improves it (line 16 of Algorithm 3). Because shaking and local search applications are extremely efficient, the simple copy of solutions for XXL instances, despite being a linear operation in the instance size, is an expensive procedure that when performed hundreds of thousands of times during the algorithm execution, becomes a critical bottleneck. In addition, given the locality of the optimization of FILO, the differences between S and S' are typically minimal if the instance is large enough and the SVC maximum capacity is limited.

For this reason, to avoid complete solution copies, in FILO2 we adopt a technique based on do/undo-lists. This technique assumes that before starting the core optimization procedure, the above mentioned three solutions S, S' and S^* are identical. In particular, we have that S^* is the solution obtained by the construction phase and $S \leftarrow S' \leftarrow S^*$. This condition can be obtained by performing two full solution copies once only.

At the beginning of every core optimization iteration, an invariant states that S is equal to S'. Next, shaking and local search applications are applied to S' to obtain a solution which is an actual neighbor of S. During these applications, every change applied to S' is recorded into a do-list $\mathcal{D}$, and likewise, the inverse change is recorded into an undo-list $\mathcal{U}$. List $\mathcal{D}$ thus contains all changes that should be applied to S to obtain S', and similarly list $\mathcal{U}$ allows to obtain S from S' by applying the stored changes in reverse order.

Whenever S' is accepted to replace S because of the SA criterion, all changes stored in $\mathcal{D}$ are applied in chronological order to S. Moreover, these changes are also inserted into an additional do-list $\mathcal{D}^*$, that will possibly at some point be used to convert the best solution S^* into S'. If, on the other hand, S' is not accepted to replace S, the undo-list $\mathcal{U}$ is applied to S' in reverse chronological order. Finally, in both cases, lists $\mathcal{D}$ and $\mathcal{U}$ are cleared.

To conclude, whenever S' is improving with respect to the best solution S^*, and thus S' should replace S^*, changes in the do-list $\mathcal{D}^*$ are applied to S^*, and then $\mathcal{D}^*$ is cleared.

We shall note that converting S into S', through the application of $\mathcal{D}$ to S when the SA acceptance criterion is satisfied, can in practice be avoided since S can always be recovered from the application of $\mathcal{U}$ to S'. This makes the existence of S redundant, and in practice not needed. Pseudocode for the updated FILO2 core optimization procedure is shown in Algorithm 4.

A similar approach is used in the route minimization procedure to keep the current working solution and the final solution synchronized during the algorithm execution.

Algorithm 4 FILO2 core optimization procedure

procedure CoreOpt($S^*, \mathcal{R}$)
  $S' \leftarrow S^*, \mathcal{T} \leftarrow \mathcal{T}_0, c_S \leftarrow \text{Cost}(S')$
  $\boldsymbol{\omega} \leftarrow (\omega_1, \omega_2, \ldots, \omega_{|V_c|}), \omega_i \leftarrow \omega_{base}\ \forall i \in V_c$
  $\boldsymbol{\gamma} \leftarrow (\gamma_0, \gamma_1, \ldots, \gamma_{|V_c|}), \gamma_i \leftarrow \gamma_{base}\ \forall i \in V$
  $\mathcal{D} \leftarrow \mathcal{U} \leftarrow \mathcal{D}^* \leftarrow []$
  for $n \leftarrow 1$ to Δ_{CO} **do**
    $S' \leftarrow \text{Shaking}(S', \mathcal{R}, \boldsymbol{\omega}, \mathcal{D}, \mathcal{U})$
    $S' \leftarrow \text{LocalSearch}(S', \mathcal{R}, \mathcal{D}, \mathcal{U})$
    if $\text{Cost}(S') < \text{Cost}(S^*)$ **then**
      $S^* \leftarrow \text{ApplyDoList}(\mathcal{D}^*, S^*)$
      $\mathcal{D}^* \leftarrow []$
      ResetSparsificationFactors($\boldsymbol{\gamma}$)
    else
      UpdateSparsificationFactors($\boldsymbol{\gamma}$)
    end if
    UpdateShakingParameters($\boldsymbol{\omega}, S', c_S, \mathcal{R}$)
    if AcceptNeighbor($c_S, S', \mathcal{T}$) **then**
      $c_S \leftarrow \text{Cost}(S')$
      $\mathcal{D}^* \leftarrow \mathcal{D}^* + \mathcal{D}$
    else
      $S' \leftarrow \text{ApplyUndoList}(\mathcal{U}, S')$
    end if
    $\mathcal{U} \leftarrow \mathcal{D} \leftarrow []$
    $\mathcal{T} \leftarrow c \cdot \mathcal{T}$
  end for
  return S^*
end procedure

Input parameters: solution S^*, random engine $\mathcal{R}$

In practical terms, the primitive changes supported by this technique are the solution operations happening during the shaking and local search steps, i.e., vertex removal and insertion, empty route creation and removal, one-customer route creation and removal, and the reverse of a contiguous sequence of vertices in a route (required by the intra-route 2-opt local search operator).

By using do/undo-lists, the synchronization between solutions is no longer a linear operation in the instance size. However, it is still bounded by the actual number of changes performed by the shaking and the local search applications which is limited by the SVC and mostly instance size independent.

2.2.6 Lazy Preprocessing for TAIL and SPLIT Local Search Operators

The local search engine of FILO employs, among others, two inter-route adaptations of the 2-opt operator (see Reinelt and Rinaldi (1994)) called TAIL and SPLIT, both working on two different routes at a time.

By denoting as head, a path of vertices belonging to the initial part of a route, and as tail, a path of vertices belonging to the final part of a route, TAIL swaps the tails 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. The execution of such moves, and in particular the check of their feasibility, benefits from the precomputation, at the beginning of the neighborhood exploration of the cumulative load up to q_i^{up} and from q_i^{from} any customer $i \in V_c$ included, i.e., $q_i^{up} = q_i + q_{\pi_i^1} + q_{\pi_i^2} + \ldots + q_0$ and $q_i^{from} = q_i + q_{\sigma_i^1} + q_{\sigma_i^2} + \ldots + q_0$ where π_i^n and σ_i^n identifies the n-th predecessor and successor of i, respectively.

Knowing q_i^{up} and q_i^{from} for any customer $i \in V_c$, allows to perform feasibility checks for TAIL and SPLIT in constant-time. Therefore, in FILO these values are precomputed for all customers $i \in V_c$ at the beginning of both the TAIL and SPLIT neighborhood explorations. However, since local search applications are very limited in their scope,

precomputing these values for all possible customers is unlikely to be useful and most certainly very time consuming for XXL instances. As a consequence, in FILO2, q_i^{up} and q_i^{from} for customers i belonging to route r are computed on-demand only when required, and cached until route r is not changed. Whenever some edit is performed to r, q_i^{up} and q_i^{from} for $i \in r$ are marked as no longer valid.

2.2.7 Hierarchical Randomized Variable Neighborhood Descent

In the original FILO algorithm, local search operators are organized according to the Hierarchical Randomized Variable Neighborhood Descent (HRVND) principle. More precisely, operators of the same cardinality are grouped together in tiers. Each tier can be seen as a compound operator in which operators are applied according to the Randomized Variable Neighborhood (RVND) principle, that is the order of application of the operators is randomly chosen before each tier application and whenever after a complete application of all the operators an improvement is found, all the operators are re-considered (inner RVND loop), see Algorithm 5 (left). Tiers are then linked together in increasing computational complexity according to the VND principle. Again, whenever after all tiers application, an improvement is found, the complete local search is repeated (outer VND loop).

Algorithm 5 FILO (left) and FILO2 (right) HRVND tier application

```
1: procedure TierApplication(S, O, R)
2:     O ← Shuffle(O, R)
3:     e ← 0, c ← 0
4:     repeat
5:         S' ← Apply(O_c, S)
6:         if Cost(S') < Cost(S) then S ← S', e ← c
7:         c ← (c + 1) mod Length(O)
8:     until c ≠ e
9:     return S
10: end procedure
```

```
1: procedure TierApplication(S, O, R)
2:     O ← Shuffle(O, R)
3:     for o ∈ O do
4:         S ← Apply(o, S)
5:     end for
6:     return S
7: end procedure
```

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

Novel extensive experimental tests performed with FILO2 have shown that the inner RVND loop is seldom able to bring significant improvements given that the outer VND loop is already performed. In FILO2 we thus simplified the HRVND by removing the inner RVND loop, that is, operators within each tier are executed only once per outer VND loop, see Algorithm 5 (right). This allows to save some costly operations without significantly decreasing the quality of final solutions.

3 Computational Testing

The computational testing aims at verifying that the above introduced changes to the original algorithm still allow FILO2 to provide state-of-the-art results on well-known literature benchmark instances as well as they make it applicable to the new dataset of XXL instances. To accomplish the first goal, we test FILO2 on the $\mathbb{X}$ and $\mathbb{B}$ instances proposed by Uchoa et al. (2017) and Arnold, Gendreau, and Sörensen (2019), respectively. The computational results of FILO2 are compared to what was previously obtained by FILO. Once assessed that FILO2 provides results comparable to those obtained by FILO, we move on to the real target of this research, that is, showing that FILO2 can easily manage much larger instances. To this end, we test the proposed approach on a novel $\mathbb{I}$ dataset containing XXL instances with up to one million customers.

3.1 Implementation and Experimental Environment

The proposed algorithm was implemented in C++ and compiled using g++ 11.2. All the experiments were performed using a single thread on a 64-bit GNU/Linux Ubuntu 22.04 mini computer equipped with an AMD Ryzen 5 PRO 4650GE CPU, running at 3.3 GHz, and 16GB of RAM.

The source code of FILO2 is available at https://github.com/acco93/filo2. In the computations we considered a standard version of FILO2, performing 10^5 core optimization iterations, and a longer version, called FILO2 (long),

		FILO		FILO2		FILO (long)		FILO2 (long)	
Size	Vertices	Avg	t	Avg	t	Avg	t	Avg	t
S	101 - 247	0.18	78	0.17	75	0.08	827	0.08	807
M	251 - 491	0.39	73	0.36	73	0.25	771	0.23	769
L	502 - 1001	0.53	75	0.50	82	0.32	763	0.29	831
All	101 - 1001	0.37	75	0.34	76	0.22	786	0.20	801

Table 2: Comparison of FILO and FILO2 on $\mathbb{X}$ instances.

which performs 10^6 iterations. For every problem instance, the algorithms were executed for a symbolic number of ten runs.

We did similarly for the original algorithm FILO, whose source code is available at https://github.com/acco93/filo. The results of FILO have been obtained by running the source code on the above mentioned computing system. Furthermore, because of their randomized nature, all the analysis on solution quality and computing time described in the following sections refer to the average behavior of the algorithms. The best solution values (BKS) for $\mathbb{X}$ and $\mathbb{B}$ instances have been taken from CVRPLIB (2022) at the time of writing, whereas BKS for the new $\mathbb{I}$ instances are the costs of the best solutions we found during the overall experimentation. Finally, gaps are computed as $100 \cdot (z - \text{BKS})/\text{BKS}$ where z is the final solution value obtained by the algorithms, the computing time is always reported in seconds, and the tables in the subsequent paragraphs always show the average percentage gap obtained by the algorithm with respect to the BKS (Avg).

3.2 Parameter Tuning

Algorithm FILO2 inherits all the parameters and their values from FILO (see Table 1 for the main ones). Moreover, FILO2 adds an additional parameter n_{nn}, first introduced in Section 2.2.2, that identifies the number of nearest neighbors that are computed, stored and used during the algorithm execution. Whenever $n_{nn} = |V|$, FILO2 would behave similarly to FILO, however, given the storage and computing time requirements it would require, this is not possible for XXL instances.

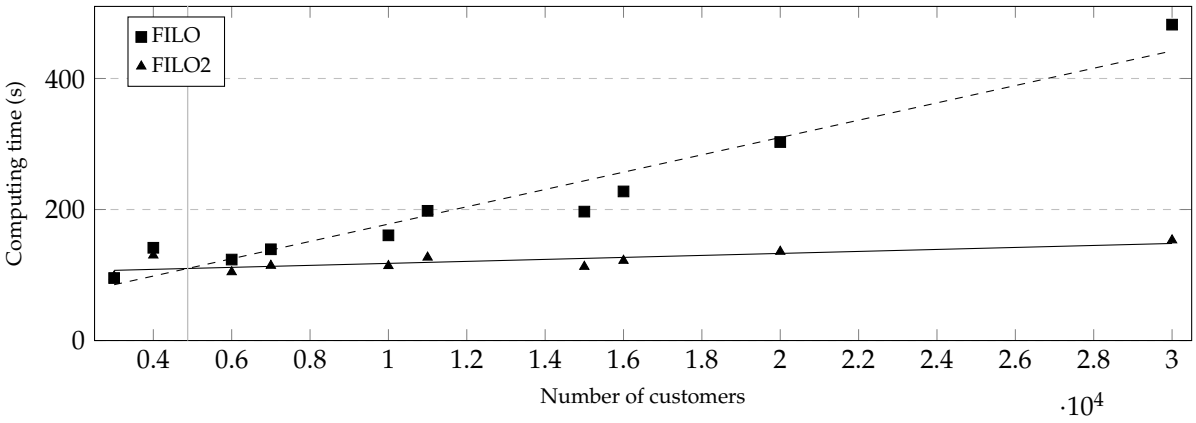
The value of n_{nn} was selected with a simple uniform sampling of reasonable values so as to obtain results of FILO2 comparable with those obtained by FILO on the $\mathbb{X}$ and $\mathbb{B}$ literature instances. At the same time, as it is described in Section 4, it was also set to obtain a good tradeoff between solution quality and computing time on the newly proposed $\mathbb{I}$ instances. As a deliberate design choice, all parameters inherited from FILO are set at their original value to both simplify the tuning process and to keep the resulting FILO2 algorithm behaving as similar as possible to the original one.

3.3 Testing on $\mathbb{X}$ Instances

The $\mathbb{X}$ dataset proposed by Uchoa et al. (2017) is the main reference literature dataset for the CVRP containing a hundred medium to large instances with a variety of customer demand distributions and vertex layouts. Table 2 compares the results of the proposed approach with those of the original one.

As can be seen from the table, the solution quality obtained by both the standard and longer version of FILO2 is comparable with that obtained by FILO. More precisely, we have that FILO and FILO2, as well as their longer version, are not significantly different in the quality of final solutions on the instances with S size. On the other hand, FILO2 always finds statistically better solutions than FILO on instances with M and L sizes (see Appendix B.1 for more details). As far as the computing time is concerned, the new approach behaves in average very similarly to the original one. Interestingly, FILO2 seems to require a longer computing time on L sized instances. However, given the results on much larger instances (see Section 3.4), we believe this to be mainly related to the slightly different search trajectory of the two algorithms, rather than because of obvious inefficiencies introduced in FILO2.

Figure 1: Computing time growth of FILO and FILO2 on the 𝔹 dataset. The dashed and the continuous lines show a linear function fitting the existing data for FILO and FILO2, respectively. The two linear functions met when the number of customers is approximately 4876.



3.4 Testing on 𝔹 Instances

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, and thus much longer routes are required to visit the customers. Table 3 compares FILO and FILO2 configurations and Figure 1 provides some hints on the computing time growth of the two algorithms.

The results show that both FILO2 and FILO2 (long) are on average better than the original approach. Moreover, statistical analysis shows that the final solution quality of FILO2 and FILO is not significantly different. On the other hand, the quality of final solutions generated by FILO2 (long) is significantly better than that of solutions generated by FILO (long). For more details see Appendix B.2. In addition, Figure 1 shows that the computing time of FILO2 increases at a much slower rate compared to that of FILO. More precisely, it seems that the newly introduced changes are more convenient in terms of computing time on very large-scale instances with at least 5000 customers, making the overall algorithm considerably more scalable when much larger instances are solved. Finally, from the actual observations, FILO2 results to be slower than FILO on the smallest L1 instance only.

		FILO		FILO2		FILO (long)		FILO2 (long)	
ID ($\lvert V_c \rvert$)	BKS	Avg	t	Avg	t	Avg	t	Avg	t
L1 (3000)	192848	0.53	95	0.42	98	0.26	958	0.20	992
L2 (4000)	111391	1.03	142	0.85	130	0.49	1515	0.32	1479
A1 (6000)	477277	0.61	124	0.52	105	0.29	1247	0.25	1073
A2 (7000)	291350	1.10	139	1.08	114	0.44	1505	0.40	1226
G1 (10000)	469531	0.74	161	0.66	114	0.30	1602	0.27	1180
G2 (11000)	257748	1.51	198	1.47	127	0.44	2284	0.41	1505
B1 (15000)	501719	1.08	197	1.02	113	0.42	2006	0.39	1206
B2 (16000)	345481	2.01	228	1.91	122	0.58	2564	0.55	1430
F1 (20000)	7240124	0.81	303	0.73	136	0.35	3276	0.31	1575
F2 (30000)	4373320	2.11	482	2.13	153	0.68	6194	0.62	2040
Mean		1.15	207	1.08	121	0.42	2315	0.37	1371

Table 3: Comparison of FILO and FILO2 on the 𝔹 dataset.

					FILO2		FILO2 (long)	
ID ($\lvert V_c \rvert \cdot 10^3$)	Depot	Q	BKS	Routes	Avg	t	Avg	t
Valle-D-Aosta (20)	C	50	21679514	800	0.28	192	0.13	2620
Molise (50)	C	50	111184982	2007	0.41	146	0.16	1901
Trentino-Alto-Adige (100)	S	150	102063181	1338	0.92	194	0.40	2292
Basilicata (150)	W	150	175623919	2007	0.74	185	0.31	2722
Umbria (200)	C	50	545507981	8006	0.29	248	0.12	2124
Abruzzo (250)	NW	200	311712556	2500	0.73	278	0.28	2613
Friuli-Venezia-Giulia (300)	SE	200	415805616	3004	0.60	393	0.25	4116
Liguria (320)	C	50	1426389867	12801	0.20	332	0.07	2964
Calabria (380)	C	50	1964651530	15201	0.40	292	0.13	2299
Marche (420)	E	200	420484426	4205	0.69	408	0.26	3634
Sardegna (470)	S	200	827934149	4732	1.61	316	0.78	2690
Campania (500)	SW	200	391859276	5003	1.01	420	0.37	4417
Piemonte (600)	C	50	2627446164	23992	0.32	404	0.15	2721
Toscana (700)	N	150	1084417188	9336	0.77	488	0.33	5793
Puglia (750)	E	200	1464797603	7538	1.49	436	0.74	2957
Sicilia (800)	NW	200	1774262462	8044	1.35	469	0.63	3298
Veneto (850)	E	200	1050488613	8499	0.71	600	0.29	5131
Emilia-Romagna (900)	C	50	5405446715	36001	0.24	501	0.09	3263
Lombardia (950)	SW	150	1339900081	12669	0.75	562	0.34	7990
Lazio (1000)	C	50	3145381332	39982	0.40	532	0.14	3938
Mean					0.70	370	0.30	3474

Table 4: Detailed results obtained by FILO2 on the new $\mathbb{I}$ instances.

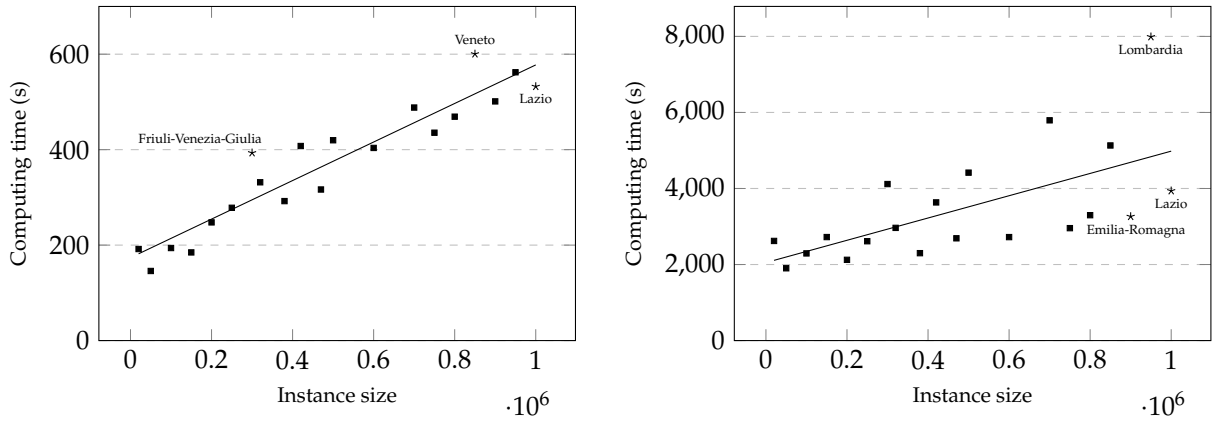
3.5 Testing on $\mathbb{I}$ Instances

We propose the new $\mathbb{I}$ dataset containing twenty XXL instances having a number of customers ranging from $20,000$ to $1,000,000$. These instances are obtained by projecting to the Euclidean plane, geographical coordinates of randomly sampled addresses in various Italian regions. The addresses are based on OpenAddresses (2022) data. Similarly to the $\mathbb{B}$ instances, customers have a load that is uniformly randomly selected in $[1, 3]$. The depot is located on the regional capital city position and the vehicle capacity is $Q = 50$ when the depot is located centrally in the region. Otherwise, the capacity is $Q = 150$ when the depot is eccentric, and $Q = 200$ when very close to the region boundary, respectively. As a consequence, the dataset contains half of the instances where we can expect relatively short routes, and half where longer routes are required. The dataset can be downloaded from https://github.com/acco93/filo2. Finally, the new instances follow the same conventions followed by the $\mathbb{X}$ and $\mathbb{B}$ datasets, that is, vertex coordinates are integer and the arc costs are rounded to the nearest integer.

Computational results of FILO2 and some instance statistics are shown in Table 4. In particular, for each instance we report its name and size in thousands of customers, the depot location in cardinal points with the addition of C to denote a central depot, and the vehicle capacity Q. Furthermore, the table reports also the cost and number of routes of the best known solution obtained during our entire experimentation as well as the average gap and computing time in seconds for both FILO2 and FILO2 (long). Detailed results are available in Appendix B.3.

To better determine the behavior of the approach, Figure 2 shows the computing time growth of the algorithm configurations on this new dataset. As can be seen from the charts both variants exhibit a linear growth of their computing time. On the one hand, FILO2 seems to generate results with generally less variance of the computing time with respect to the predicted value given by the linear regression on the observed data. On the other hand, the distance of points from the regression line seems to increase with the instance size for FILO2 (long). Highly influential points, i.e., points that if removed would considerably change the regreession line, can be identified by computing the Cooks's distance (see Cook (1977)). The three most influential points, in decreasing order of influence, for both algorithms, are Veneto (0.247811), Friuli-Venezia-Giulia (0.174067), Lazio (0.135176) for FILO2 and Lombardia (0.850441), Emilia-Romagna (0.141266), and Lazio (0.114841) for FILO2 (long).

Figure 2: Computing time growth of FILO2 (left) and FILO2 (long) (right) on the 𝕀 dataset. Marks represent the algorithms computing time, while the line linearly fits the data. The most three most influential observations according to the Cook's distance are marked with a star.



It is difficult to determine the reasons for the abnormally larger computing time of FILO2 (long) on instance Lombardia. We can however observe that the running time of the standard FILO2 version on such instance is in line with those of the other instances of similar size.

An additional interpretation could be that it is not instance Lombardia requiring a long computing time when solved by FILO2 (long) but rather that instances with similar sizes are indeed solved faster by FILO2 (long) compared to what we could expect by a tenfold increase in the number of iterations of FILO2. In fact, contrarily to what is happening for the 𝕏 and 𝔹 datasets, the average computing time of FILO2 (long) on the 𝕀 instances is smaller than ten times the computing time of FILO2. More specifically, this is happening for 12 instances out of 20 sometimes with unexpectably low values (e.g., see instance Emilia-Romagna).

4 Algorithmic Components Analysis

This section provides some insights on the effectiveness of the newly introduced changes in FILO2. If not stated otherwise, the analysis are performed on the 𝕀 instances by running the the standard FILO2 configuration for 10^5 core optimization iterations. Computational results always refer to the average of ten distinct runs.

4.1 Number of Neighbors

When moving to XXL instances, the initial computation of the limited neighbor lists $\mathcal{N}_i^{n_{nn}}(V), i \in V$ may take a significant portion of the overall algorithm execution time, especially when relatively few core optimization iterations are performed, as in the standard FILO2 version. As an example, for the largest instance (Lazio), the initial preprocessing performed by FILO2 takes approximately 48% of the total computing time. Table 5 shows, for every instance in the 𝕀 dataset, the running time spent to compute the neighbor lists and the percentage of the overall algorithm computing time for both FILO2 and FILO2 (long).

Luckily this procedure could in practice be easily parallelized, thus the process could be sped up by employing more CPU cores at the same time. Finally, in scenarios in which parallelization is not allowed but initial solutions are required to be generated as fast as possible, e.g., the DIMACS (2022) competition where algorithm were evaluated according to the primal integral metric proposed in Achterberg, Berthold, and Hendel (2012), one could employ more sophisticated approaches based on a lazy computation of neighbors for a vertex as they are required by the algorithm procedures, e.g., initially only $n_{cw} < n_{nn}$ neighbors are computed so that they can be used during the initial construction phase based on the limited savings algorithm (see Section 2.1), then the list of neighbors for

ID	$t(s)$	FILO2 (%)	FILO2 (long) (%)
Valle-D-Aosta	4.00	2.09	0.15
Molise	10.00	6.86	0.53
Trentino-Alto-Adige	23.00	11.86	1.00
Basilicata	33.00	17.87	1.21
Umbria	49.00	19.79	2.31
Abruzzo	61.00	21.93	2.33
Friuli-Venezia-Giulia	75.00	19.07	1.82
Liguria	78.00	23.52	2.63
Calabria	92.00	31.49	4.00
Marche	103.00	25.27	2.83
Sardegna	113.10	35.73	4.20
Campania	127.00	30.26	2.88
Piemonte	153.00	37.91	5.62
Toscana	177.60	36.38	3.07
Puglia	178.10	40.90	6.02
Sicilia	197.90	42.19	6.00
Veneto	222.20	37.00	4.33
Emilia-Romagna	234.60	46.82	7.19
Lombardia	241.90	43.04	3.03
Lazio	258.60	48.58	6.57
Mean	121.60	32.89	3.50

Table 5: Preprocessing time used to compute the neighbor lists and the percentage of the total computing time of FILO2 and FILO2 (long) spent performing this initial instance preprocessing.

specific vertices i is enlarged as soon as it is required, e.g., when a recreate application tries to reinsert a customer i back into the solution.

As mentioned in Section 3.2, the computing time for the definition of neighbor lists is affected by parameter n_{nn}. This not only affects the initial preprocessing time but also the overall algorithm computing time, the memory requirements as well as the final solution quality. Figure 3 shows the overall computing time as well as the final solution quality for several value of n_{nn}. Values larger than 2500 would require, for the largest $\mathbb{I}$ instances, more memory than that available in the computing system used for experimentation.

As can be seen from the chart, most of the quality improvement happens for n_{nn} lower than 1250, then the improvement ratio decreases even though it still remains improving while the number of neighbors is increased. The selected parameter $n_{nn} = 1500$ provides a satisfying tradeoff between computing time and final solution quality.

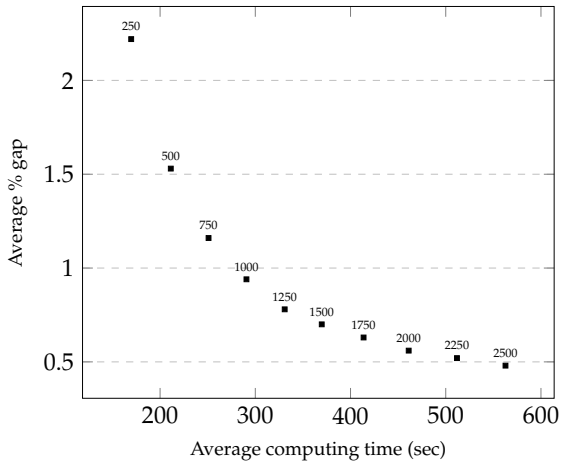
4.1.1 Approximated Simulated Annealing Temperature

The initial and final SA temperatures $\mathcal{T}_0$ and $\mathcal{T}_f$ are defined to be proportional to the average arc cost d of the instance under examination, and more precisely $\mathcal{T}_0 = 0.1 \cdot d$ and $\mathcal{T}_f = 0.01 \cdot \mathcal{T}_0$. As mentioned in Section 2.2.4, computing d may be too time consuming for XXL instances. In FILO2 we thus use an approximation $\tilde{d}$ defined as the average arc cost obtained by randomly sampling $|V|$ instance arcs. In this section, we compare the quality and computing time of this approximation with respect to the exact computation.

For the $\mathbb{X}$ and $\mathbb{B}$ datasets, the time to compute d is negligible, and in particular, it is about 3 seconds for the largest F2 instance. The value of $\mathcal{T}_0$ is 47.50 and 186.45 for the $\mathbb{X}$ and $\mathbb{B}$ datasets averaged over all instances when computing with d, and on average 47.27 and 186.46 when computed with $\tilde{d}$.

The difference in computing time increases considerably for the $\mathbb{I}$ instances. More precisely, the average running time for computing d is more than 1000 seconds while $\tilde{d}$ is computed in less than 30 milliseconds. The value of $\mathcal{T}_0$ is 6451.97 when computed with d, and on average 6452.09 when computed with $\tilde{d}$. Table 6 provides more details.

Figure 3: FILO2 average computing time and final solution quality on the 𝕀 dataset while varying parameter n_{nn}.



	Exact ($\mathcal{T}_0 = 0.1 \cdot d$)		Approximated ($\mathcal{T}_0 = 0.1 \cdot \tilde{d}$)	
ID	$\mathcal{T}_0$	$t(s)$	$\mathcal{T}_0$	$t(ms)$
Valle-D-Aosta	1784.73	1.32	1780.85	0.00
Molise	3558.74	8.27	3553.99	0.00
Trentino-Alto-Adige	4809.19	33.11	4810.42	1.20
Basilicata	5586.32	74.51	5588.64	3.20
Umbria	4354.79	132.48	4355.77	6.00
Abruzzo	5452.96	207.02	5456.59	11.00
Friuli-Venezia-Giulia	4463.20	298.18	4462.78	15.10
Liguria	7347.70	339.24	7346.42	16.90
Calabria	8155.82	478.45	8156.91	21.70
Marche	5599.47	584.64	5602.36	24.20
Sardegna	8681.42	732.39	8683.78	28.40
Campania	5047.63	828.64	5049.10	30.40
Piemonte	7377.79	1193.74	7375.64	37.90
Toscana	6825.34	1624.52	6823.56	47.20
Puglia	11562.96	1864.79	11561.90	48.70
Sicilia	11035.42	2122.36	11038.61	52.40
Veneto	6388.52	2395.48	6389.44	55.60
Emilia-Romagna	8527.99	2686.10	8526.11	59.20
Lombardia	6767.92	2993.95	6768.97	65.50
Lazio	5711.48	3315.50	5709.98	65.70
Mean	6451.97	1095.73	6452.09	29.51

Table 6: Comparison of exact and approximate SA initial temperature $\mathcal{T}_0$. Note that for the exact approach the computing time is in seconds, while for the approximated one the time is given in milliseconds.

Finally, because of its heuristic usage, we consider $\tilde{d}$ to be a good estimation of d with an error of about 0.001%, and we believe the two values to be close enough for FILO2 to explore values of SA temperatures similar to those explored by FILO.

4.1.2 Accessing Arc Costs

Given that it is widely used in most of the algorithm procedures, retrieving arc costs is a critical operation that if not properly handled may become one of the main algorithm bottleneck. As described in Section 2.2.1, memory constraints impose that the cost matrix cannot be explicitly stored, but costs need to be computed on-demand or retrieved from a limited-size cache. Table 7 compares the computing time of FILO2 when different approaches are used to handle cost retrieval. In particular, it compares the algorithm performance when costs are computed on-demand or cached as described in Bentley (1990). Moreover, for each possibility we consider the effect of using $O(|V| + |T|)$ additional space, where V is the set of vertices and T is the set of move generators, to store frequently used costs in the solution and in move generators to allow for some constant-time cost retrieval. In the table these versions are identified by the $^+$ symbol.

As one would expect, the $^+$ versions using additional space to allow some constant-time retrieval of costs are always preferable compared to the respective version not exploiting this possibility. More surprisingly, the versions in which costs are computed on-demand are considerably more efficient than the versions using an additional caching mechanism.

The cache implementation follows the suggestions described in Bentley (1990). In particular, we implemented a simple linear cache composed of a signature and a value vectors, respectively sig and val, having a size equal to the smallest power of two greater than or equal to the instance size $|V|$.

Given an arc (i, j) normalized such that $i \leq j$, first a hash index h is computed as $h = i \oplus j$ where $\oplus$ represents a bitwise xor operation. The caching algorithm then checks the signature entry $sig[h]$ and if $sig[h] \neq i$ it recomputes the distance c_{ij}, sets $val[h] = c_{ij}$, and updates $sig[h] = i$. On the other hand if $sig[h] = i$, it immediately returns $val[h]$ which will contain the correct cost of arc (i, j). This works since $sig[h]$ represents a signature of the arc (i, j) given that $h = i \oplus j$, thus $j = h \oplus i$.

In Bentley (1990), the author states that the cache will prove effective if the underlying algorithm displays a great locality in the computation of the cost values. Given its optimization pattern, FILO2 should definitely have this locality property. In fact, Table 8 shows that the average cache hit ratio is considerably high for both versions using cached costs. The better performance of the on-demand versions might thus be related to hardware advancements rather than to a low usage of the cache. As a final note, we specify that, despite all datasets used in this study consider integer arc costs, in FILO2 arc costs are stored and used as floating point numbers with double precision.

4.1.3 Recreate Strategy

Part of the recreate strategy tuning is described in Section 4.1 where the final solution quality and computing time is analyzed while varying parameter n_{nn} that is also affecting the breadth of the recreate procedure as described in Section 2.2.3.

In this section, we analyze what happens if the recreate of the original FILO algorithm, which greedily reinserts

Configuration	t
Cached	471
Cached$^+$	419
On-demand	409
On-demand$^+$	370

Table 7: Average FILO2 computing time for different cost retrieval strategies.
$^+$Some costs are retrieved in constant-time from the current solution and from move generators.

Table 8: Average cache percentage hit ratio.

ID	Cached	Cached$^+$
Valle-D-Aosta	91.24	84.37
Molise	93.18	87.34
Trentino-Alto-Adige	90.17	82.34
Basilicata	89.77	81.65
Umbria	93.74	88.20
Abruzzo	89.62	81.49
Friuli-Venezia-Giulia	89.64	81.44
Liguria	93.46	87.71
Calabria	93.75	88.19
Marche	89.89	81.88
Sardegna	87.94	78.83
Campania	89.03	80.54
Piemonte	93.62	88.01
Toscana	90.52	82.92
Puglia	89.15	80.72
Sicilia	89.00	80.51
Veneto	89.75	81.74
Emilia-Romagna	93.28	87.46
Lombardia	89.59	81.53
Lazio	92.97	86.93
Mean	90.99	83.67

a removed customer in its best possible position considering the whole solution, is employed to solve the new $\mathbb{I}$ instances.

The total average computing time increases to 1224 seconds compared to the 370 seconds of the restricted approach. Moreover, the average solution quality decreases to 1.02% compared to 0.70% of the current FILO2 approach. This experimentally shows that carefully restricting reinsertions to close points for large enough instances does not necessarily harm the final solution quality but, on the other hand, may positively influence the search trajectory.

4.1.4 Solution Copies

The larger the instance the more convenient becomes the synchronization between solutions by using the do/undo-lists described in Section 2.2.5.

The average computing time when performing standard solution copies is 2011 seconds, that is more than 5 times larger than the computing time of the algorithms when using the do/undo-lists approach.

4.1.5 Lazy Updates for TAIL and SPLIT Local Search Operators

It's easy to see that precomputing all q_i^{up} and q_i^{from} for every customer $i \in V_c$ as it is described in Section 2.2.6 cannot be beneficial in any way for very large instances. It is in fact incredily expensive since these computations are performed whenever a TAIL or SPLIT neighborhood is explored (possibly several times in a VND loop). In practice, the computing time of the algorithm when these values are always precomputed for all customers is 15843 seconds, i.e., more than 40 times more expensive than the version using lazy updates.

4.1.6 Hierarchical Randomized Variable Neighborhood Descent

The average gap of FILO2 with the standard RVND loop (see Section 2.2.7) is 0.69% while when only a single scan of the operators is performed the average gap is 0.70%. On the other hand, the average computing time of the first version is 413 seconds, while the computing time of the second one is 370. Finally, if we analyze the performance of FILO2 (long) we have that for both versions the average gap is 0.30% but the computing time for the version

with the standard RVND loop is 4085 versus 3474 for the single-scan one. We can conclude that looping through the operators multiple times provides some limited quality advantage on short runs but it does not result on average beneficial on longer runs.

5 Conclusions

This paper introduces the 𝕀 dataset for CVRP containing twenty XXL instances with a number of customers ranging from 20,000 to 1,000,000. The design of extremely efficient algorithms is crucial when dealing with such extremely large instance sizes which are intractable for most existing algorithms for CVRP. We believe these new instances may foster fresh research interest in this direction. The 𝕀 instances are solved with FILO2, an evolution of the FILO algorithm proposed in Accorsi and Vigo (2021). FILO2 is able to scale to these sizes still keeping, and sometimes improving, the performance of the original approach on existing well-known literature instances. Computational results are obtained with an ordinary computing system in a considerably short computing time, showing the effectiveness of the proposed changes when combined with the techniques already introduced in Accorsi and Vigo (2021).

6 Acknowledgments

We are kindly grateful to Francesco Cavaliere for several discussions about this work and for his contribution to part of the source code that was used as a base for the proposed approach. The research of Daniele Vigo is partially supported by the U.S. Air Force Office of Scientific Research [Award FA8655-21-1-7046].

A Instances Layout

Figures 4 and 5 show the layout of the new instances composing the 𝕀 dataset. In particular, the black dots represent customers, while the depot is the empty square.

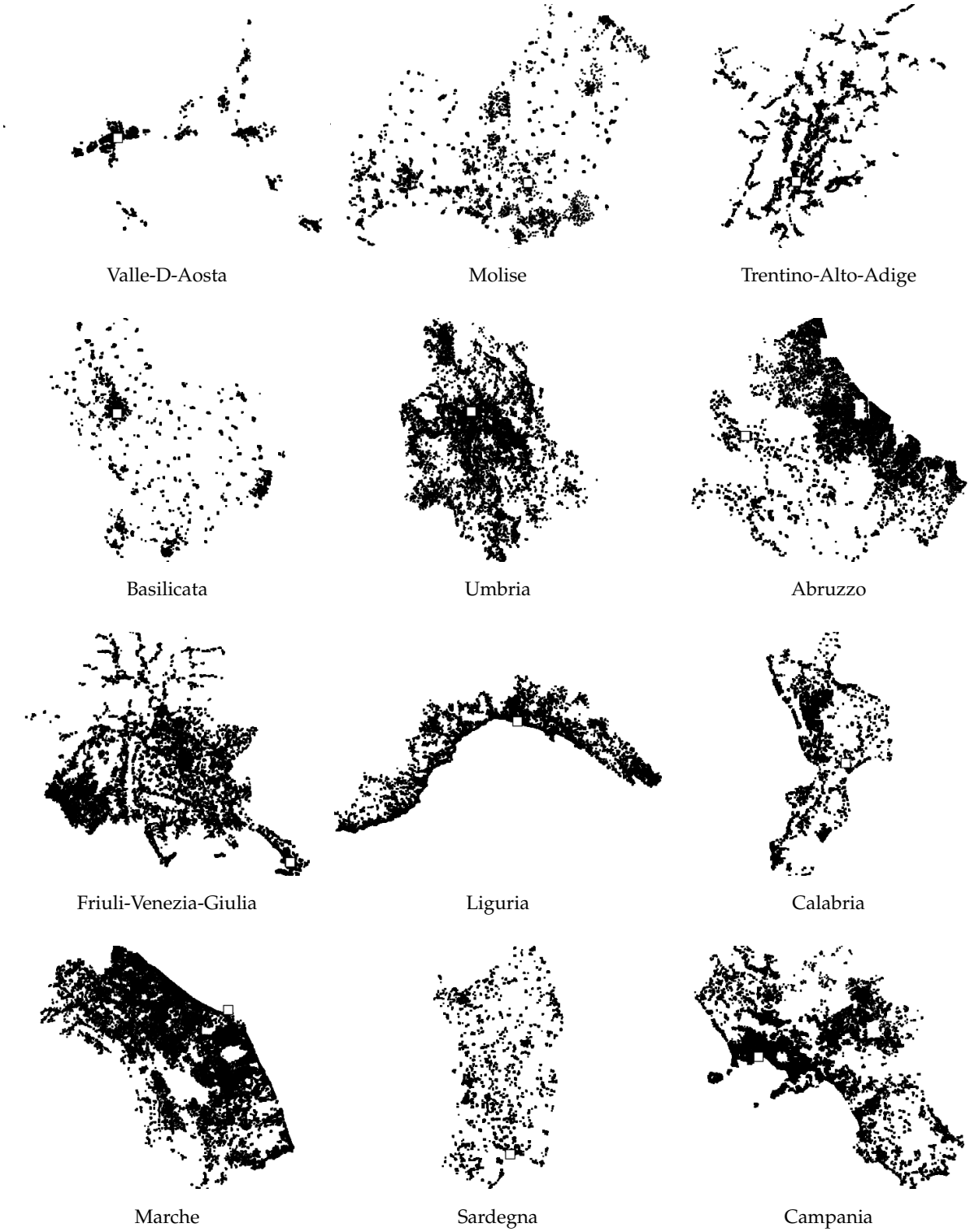


Figure 4: 𝕀 instances layout

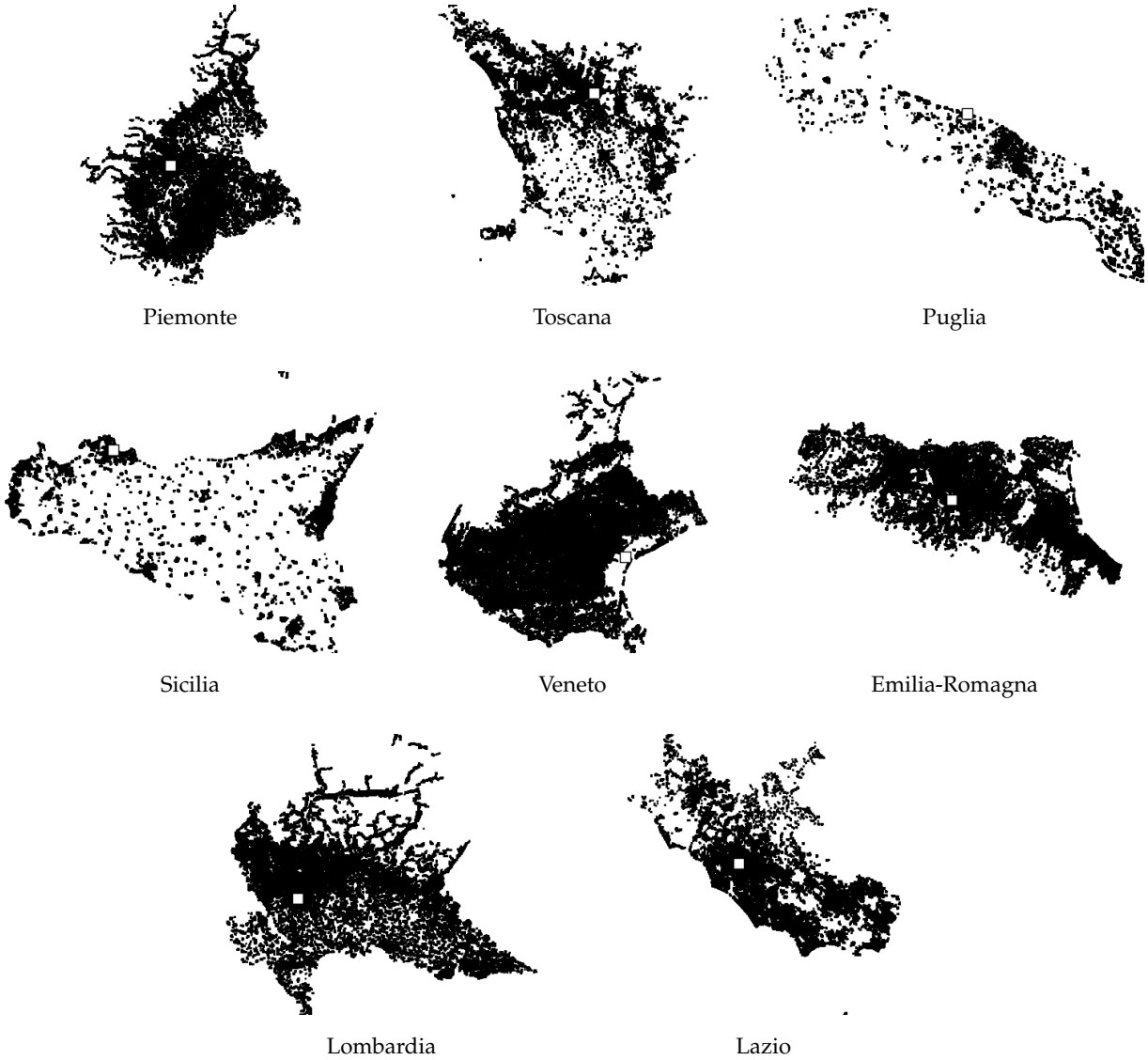


Figure 5: $\mathbb{I}$ instances layout

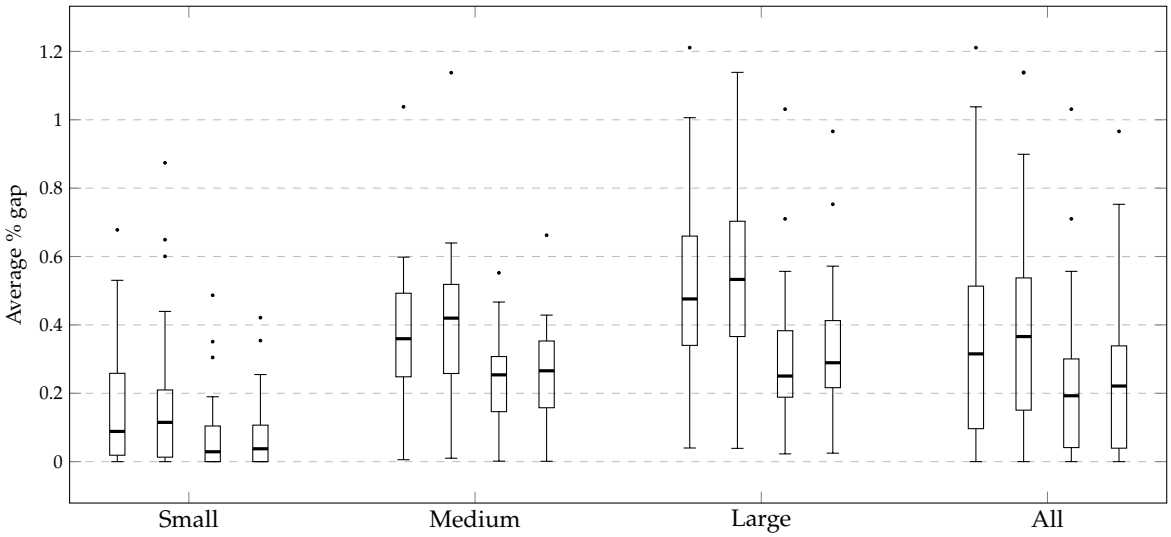


Figure 6: Comparison of average gaps obtained by the FILO2 and FILO versions on the $\mathbb{X}$ dataset. For each group, boxplots, from left to right, are associated with: FILO2, FILO, FILO2 (long), and FILO (long).

B Computational Details

Sections B.1, B.2, and B.3 provide additional computational details about the $\mathbb{X}$, $\mathbb{B}$, and $\mathbb{I}$ datasets, respectively. For the $\mathbb{X}$ and $\mathbb{B}$ computations we also provide results of a statistical test to better evaluate whether there is some significant difference between the behavior of FILO2 and FILO. More precisely, similarly to Accorsi and Vigo (2021), we performed a one-tailed Wilcoxon signed-rank test (Wilcoxon, 1945) where the null and alternative hypotheses H_0 and H_1, respectively, are defined as follows.

$$H_0 : \textsc{AverageCost}(FILO2) = \textsc{AverageCost}(FILO)$$

$$H_1 : \textsc{AverageCost}(FILO2) > \textsc{AverageCost}(FILO)$$

A hypothesis is rejected when its associated p-value is lower than a significance level α. Failing to reject H_0 means that the average results of FILO2 and FILO are not statistically different. On the other hand, when H_0 is rejected, the average results obtained by the algorithms are statistically different and the alternative hypothesis H_1 is tested to find whether the average results obtained by FILO2 are statistically greater than those of FILO. Rejecting H_1 implies that FILO2 performs better than FILO.

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.

B.1 Computational Details for $\mathbb{X}$ Instances

Detailed results about computations on the $\mathbb{X}$ dataset can be found in Tables 9 – 11 and Figure 6.

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

		FILO				FILO2				FILO (long)				FILO2 (long)			
ID	BKS	Best	Avg	Worst	t	Best	Avg	Worst	t	Best	Avg	Worst	t	Best	Avg	Worst	t
X-n101-k25	27591	0.00	0.00	0.00	53	0.00	0.00	0.00	49	0.00	0.00	0.00	569	0.00	0.00	0.00	534
X-n106-k14	26362	0.00	0.06	0.10	94	0.00	0.05	0.11	88	0.00	0.02	0.08	922	0.00	0.00	0.03	891
X-n110-k13	14971	0.00	0.00	0.00	76	0.00	0.00	0.00	78	0.00	0.00	0.00	767	0.00	0.00	0.00	782
X-n115-k10	12747	0.00	0.00	0.00	76	0.00	0.00	0.00	72	0.00	0.00	0.00	784	0.00	0.00	0.00	733
X-n120-k6	13332	0.00	0.00	0.00	79	0.00	0.00	0.00	83	0.00	0.00	0.00	813	0.00	0.00	0.00	849
X-n125-k30	55539	0.27	0.60	0.86	58	0.08	0.52	0.86	56	0.00	0.10	0.32	562	0.00	0.10	0.29	523
X-n129-k18	28940	0.03	0.06	0.19	76	0.00	0.07	0.15	68	0.00	0.03	0.05	825	0.00	0.02	0.05	764
X-n134-k13	10916	0.00	0.15	0.26	72	0.00	0.15	0.22	71	0.00	0.06	0.16	776	0.00	0.04	0.16	764
X-n139-k10	13590	0.00	0.00	0.00	90	0.00	0.00	0.00	95	0.00	0.00	0.00	998	0.00	0.00	0.00	998
X-n143-k7	15700	0.00	0.15	0.17	68	0.00	0.15	0.17	68	0.00	0.13	0.17	787	0.00	0.08	0.17	774
X-n148-k46	43448	0.00	0.16	0.38	63	0.00	0.16	0.48	54	0.00	0.06	0.30	703	0.00	0.09	0.31	632
X-n153-k22	21220	0.02	0.21	0.55	79	0.02	0.30	0.69	66	0.02	0.04	0.16	668	0.02	0.02	0.02	586
X-n157-k13	16876	0.00	0.00	0.00	119	0.00	0.00	0.00	112	0.00	0.00	0.00	1311	0.00	0.00	0.00	1334
X-n162-k11	14138	0.08	0.18	0.23	84	0.00	0.09	0.22	75	0.06	0.10	0.13	861	0.00	0.05	0.08	865
X-n167-k10	20557	0.00	0.04	0.13	81	0.00	0.10	0.16	85	0.00	0.00	0.00	935	0.00	0.00	0.00	844
X-n172-k51	45607	0.00	0.01	0.15	59	0.00	0.00	0.00	52	0.00	0.00	0.00	614	0.00	0.00	0.00	583
X-n176-k26	47812	0.03	0.39	0.91	59	0.00	0.50	1.14	54	0.00	0.08	0.28	648	0.00	0.13	0.71	654
X-n181-k23	25569	0.00	0.01	0.08	101	0.00	0.03	0.09	104	0.00	0.00	0.01	1097	0.00	0.00	0.00	1033
X-n186-k15	24145	0.02	0.07	0.16	68	0.01	0.05	0.21	64	0.01	0.03	0.12	711	0.00	0.05	0.14	704
X-n190-k8	16980	0.02	0.08	0.20	78	0.02	0.05	0.11	77	0.00	0.02	0.04	788	0.00	0.02	0.04	809
X-n195-k51	44225	0.06	0.19	0.43	61	0.06	0.22	0.43	52	0.00	0.04	0.19	581	0.00	0.07	0.23	527
X-n200-k36	58578	0.14	0.87	1.93	66	0.45	0.53	0.79	57	0.08	0.35	0.49	750	0.02	0.30	0.49	642
X-n204-k19	19565	0.03	0.07	0.20	63	0.00	0.09	0.68	62	0.00	0.00	0.03	662	0.00	0.00	0.00	693
X-n209-k16	30656	0.07	0.16	0.46	59	0.01	0.09	0.14	58	0.00	0.05	0.09	612	0.00	0.03	0.09	599
X-n214-k11	10856	0.13	0.35	0.60	58	0.05	0.26	0.49	60	0.04	0.18	0.38	633	0.10	0.19	0.30	614
X-n219-k73	117595	0.00	0.00	0.01	227	0.00	0.00	0.00	231	0.00	0.00	0.00	2489	0.00	0.00	0.00	2493
X-n223-k34	40437	0.08	0.21	0.30	55	0.09	0.31	0.61	58	0.00	0.17	0.25	586	0.16	0.18	0.25	549
X-n228-k23	25742	0.00	0.20	0.41	59	0.17	0.22	0.38	60	0.00	0.17	0.21	618	0.00	0.14	0.20	656
X-n233-k16	19230	0.16	0.44	0.56	75	0.00	0.44	0.57	65	0.06	0.25	0.53	748	0.00	0.35	0.53	746
X-n237-k14	27042	0.00	0.01	0.03	90	0.00	0.06	0.36	96	0.00	0.04	0.16	1043	0.00	0.01	0.09	1116
X-n242-k48	82751	0.09	0.30	0.70	77	0.03	0.27	0.41	67	0.09	0.19	0.24	851	0.04	0.15	0.33	778
X-n247-k50	37274	0.07	0.65	0.87	68	0.02	0.68	1.03	66	0.01	0.42	0.75	757	0.03	0.49	0.73	769
Mean		0.04	0.18	0.34	78	0.03	0.17	0.33	75	0.01	0.08	0.16	827	0.01	0.08	0.16	807

Table 10: Computations on medium-sized 𝕏 instances.

		FILO				FILO2				FILO (long)				FILO2 (long)			
ID	BKS	Best	Avg	Worst	t	Best	Avg	Worst	t	Best	Avg	Worst	t	Best	Avg	Worst	t
X-n251-k28	38684	0.04	0.34	0.44	69	0.21	0.37	0.51	74	0.16	0.31	0.40	812	0.17	0.26	0.37	750
X-n256-k16	18839	0.22	0.22	0.22	92	0.22	0.22	0.22	89	0.22	0.22	0.22	994	0.22	0.22	0.22	970
X-n261-k13	26558	0.28	0.48	0.71	64	0.23	0.36	0.72	64	0.01	0.36	0.50	745	0.01	0.27	0.50	692
X-n266-k58	75478	0.21	0.52	0.72	91	0.17	0.36	0.52	76	0.18	0.42	0.54	960	0.19	0.37	0.61	941
X-n270-k35	35291	0.09	0.24	0.35	60	0.05	0.22	0.35	61	0.07	0.16	0.28	657	0.05	0.14	0.27	640
X-n275-k28	21245	0.00	0.06	0.31	81	0.00	0.07	0.29	93	0.00	0.00	0.04	884	0.00	0.01	0.12	994
X-n280-k17	33503	0.29	0.48	0.67	63	0.12	0.28	0.45	60	0.16	0.39	0.53	661	0.06	0.31	0.50	743
X-n284-k15	20215	0.17	0.46	0.82	69	0.34	0.51	0.87	69	0.09	0.22	0.31	684	0.10	0.27	0.46	669
X-n289-k60	95151	0.46	0.59	0.75	86	0.51	0.59	0.86	74	0.28	0.38	0.48	890	0.26	0.39	0.50	847
X-n294-k50	47161	0.22	0.35	0.45	51	0.14	0.30	0.44	50	0.17	0.27	0.35	542	0.15	0.20	0.34	478
X-n298-k31	34231	0.13	0.27	0.39	53	0.01	0.26	0.59	52	0.18	0.23	0.27	553	0.00	0.16	0.26	555
X-n303-k21	21736	0.31	0.48	1.01	53	0.27	0.41	0.68	55	0.10	0.34	0.44	514	0.12	0.32	0.47	539
X-n308-k13	25859	0.02	0.35	1.63	67	0.07	0.60	1.43	68	0.03	0.33	0.72	799	0.04	0.27	0.78	735
X-n313-k71	94043	0.26	0.58	0.80	66	0.38	0.53	0.75	61	0.19	0.29	0.52	683	0.22	0.30	0.49	663
X-n317-k53	78355	0.00	0.01	0.04	135	0.00	0.01	0.03	142	0.00	0.00	0.01	1442	0.00	0.00	0.01	1464
X-n322-k28	29834	0.25	0.53	0.92	54	0.16	0.34	0.69	56	0.05	0.36	0.62	608	0.11	0.35	0.49	598
X-n327-k20	27532	0.20	0.44	0.74	71	0.13	0.39	0.74	74	0.09	0.24	0.37	809	0.00	0.26	0.32	800
X-n331-k15	31102	0.00	0.05	0.31	85	0.00	0.01	0.01	86	0.00	0.01	0.01	919	0.00	0.00	0.01	962
X-n336-k84	139111	0.49	0.63	0.90	67	0.49	0.57	0.72	57	0.20	0.35	0.48	643	0.18	0.35	0.51	595
X-n344-k43	42050	0.34	0.56	0.70	60	0.23	0.49	0.72	61	0.20	0.36	0.52	555	0.16	0.31	0.43	590
X-n351-k40	25896	0.37	0.54	0.76	59	0.44	0.60	0.73	59	0.27	0.43	0.66	567	0.27	0.47	0.61	610
X-n359-k29	51505	0.30	0.47	0.69	64	0.17	0.44	0.84	62	0.09	0.24	0.38	726	0.04	0.15	0.32	683
X-n367-k17	22814	0.00	0.19	0.44	68	0.00	0.12	0.50	70	0.00	0.01	0.06	710	0.00	0.02	0.05	738
X-n376-k94	147713	0.00	0.01	0.03	180	0.00	0.01	0.02	173	0.00	0.00	0.02	1894	0.00	0.00	0.01	1792
X-n384-k52	65928	0.24	0.43	0.72	74	0.27	0.42	0.53	72	0.18	0.32	0.38	835	0.15	0.25	0.42	783
X-n393-k38	38260	0.08	0.28	0.58	59	0.10	0.24	0.60	65	0.08	0.12	0.27	656	0.02	0.09	0.12	653
X-n401-k29	66154	0.11	0.26	0.39	85	0.18	0.25	0.32	86	0.11	0.15	0.19	858	0.09	0.13	0.15	865
X-n411-k19	19712	0.27	0.50	0.98	74	0.26	0.41	0.56	77	0.25	0.35	0.60	795	0.31	0.33	0.36	840
X-n420-k130	107798	0.13	0.22	0.31	53	0.12	0.27	0.36	53	0.09	0.14	0.27	511	0.03	0.15	0.25	540
X-n429-k61	65449	0.28	0.40	0.53	62	0.23	0.41	0.71	61	0.12	0.23	0.36	674	0.15	0.20	0.29	648
X-n439-k37	36391	0.01	0.10	0.24	71	0.01	0.07	0.24	80	0.01	0.01	0.01	767	0.01	0.01	0.04	821
X-n449-k29	55233	0.41	0.64	0.89	61	0.30	0.60	0.92	64	0.24	0.38	0.59	652	0.17	0.29	0.43	634
X-n459-k26	24139	0.22	0.41	0.83	67	0.12	0.26	0.46	69	0.12	0.26	0.42	735	0.14	0.25	0.43	732
X-n469-k138	221824	0.93	1.14	1.31	66	0.76	1.04	1.23	73	0.49	0.66	0.82	669	0.42	0.55	0.68	681
X-n480-k70	89449	0.24	0.39	0.49	71	0.24	0.32	0.43	72	0.05	0.22	0.31	746	0.02	0.18	0.45	799
X-n491-k59	66483	0.28	0.53	0.69	62	0.39	0.53	0.74	62	0.16	0.35	0.49	610	0.25	0.31	0.39	636
Mean		0.22	0.39	0.63	73	0.20	0.36	0.58	73	0.13	0.25	0.37	771	0.11	0.23	0.35	769

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

		FILO				FILO2				FILO (long)				FILO2 (long)			
ID	BKS	Best	Avg	Worst	t	Best	Avg	Worst	t	Best	Avg	Worst	t	Best	Avg	Worst	t
X-n502-k39	69226	0.00	0.05	0.09	105	0.01	0.06	0.10	120	0.00	0.03	0.07	1089	0.01	0.04	0.06	1222
X-n513-k21	24201	0.15	0.37	0.69	73	0.00	0.26	0.45	82	0.00	0.13	0.40	762	0.00	0.11	0.32	843
X-n524-k153	154593	0.27	0.46	0.59	62	0.19	0.44	0.97	67	0.04	0.23	0.39	618	0.01	0.20	0.48	670
X-n536-k96	94846	0.69	0.87	0.95	70	0.76	0.85	0.92	70	0.68	0.75	0.80	708	0.56	0.71	0.81	702
X-n548-k50	86700	0.06	0.12	0.19	80	0.00	0.09	0.22	92	0.00	0.04	0.12	828	0.00	0.06	0.11	956
X-n561-k42	42717	0.26	0.45	0.63	57	0.18	0.39	0.62	59	0.16	0.31	0.52	578	0.10	0.22	0.31	615
X-n573-k30	50673	0.24	0.37	0.51	79	0.24	0.35	0.60	88	0.17	0.29	0.48	809	0.14	0.23	0.41	923
X-n586-k159	190316	0.54	0.64	0.89	74	0.38	0.57	0.77	77	0.23	0.38	0.55	790	0.21	0.38	0.47	797
X-n599-k92	108451	0.37	0.54	0.69	72	0.27	0.37	0.54	78	0.24	0.29	0.40	804	0.19	0.27	0.40	831
X-n613-k62	59535	0.36	0.63	0.82	51	0.47	0.68	0.84	54	0.19	0.43	0.66	500	0.08	0.32	0.52	540
X-n627-k43	62164	0.22	0.36	0.52	76	0.31	0.37	0.46	79	0.09	0.23	0.33	796	0.09	0.21	0.26	791
X-n641-k35	63682	0.16	0.42	0.61	79	0.24	0.31	0.44	83	0.20	0.27	0.34	829	0.12	0.23	0.43	812
X-n655-k131	106780	0.02	0.04	0.06	141	0.01	0.04	0.06	162	0.00	0.02	0.04	1479	0.00	0.02	0.04	1633
X-n670-k130	146332	0.65	1.14	1.69	64	0.53	1.21	2.21	69	0.65	0.97	1.35	663	0.62	1.03	1.83	724
X-n685-k75	68205	0.47	0.66	1.05	63	0.36	0.62	0.73	67	0.32	0.46	0.58	593	0.23	0.44	0.60	685
X-n701-k44	81923	0.49	0.54	0.66	67	0.23	0.43	0.57	70	0.15	0.28	0.44	672	0.12	0.22	0.32	710
X-n716-k35	43373	0.60	0.72	0.89	70	0.47	0.62	0.73	75	0.25	0.36	0.54	702	0.16	0.28	0.44	742
X-n733-k159	136187	0.26	0.40	0.60	60	0.23	0.36	0.48	65	0.12	0.22	0.34	572	0.12	0.20	0.29	661
X-n749-k98	77269	0.71	0.80	0.94	65	0.62	0.75	0.89	66	0.46	0.52	0.60	608	0.30	0.44	0.56	647
X-n766-k71	114417	0.43	0.70	0.96	70	0.37	0.58	0.68	74	0.26	0.46	0.62	699	0.19	0.45	0.67	746
X-n783-k48	72386	0.40	0.68	0.81	76	0.23	0.48	0.87	85	0.06	0.29	0.45	794	0.18	0.29	0.40	847
X-n801-k40	73305	0.10	0.22	0.34	74	0.13	0.25	0.44	89	0.09	0.16	0.27	782	0.06	0.12	0.19	893
X-n819-k171	158121	0.70	0.85	1.02	64	0.62	0.76	0.95	66	0.47	0.57	0.70	650	0.48	0.56	0.61	690
X-n837-k142	193737	0.46	0.52	0.63	76	0.40	0.48	0.56	81	0.19	0.28	0.40	823	0.22	0.29	0.36	861
X-n856-k95	88965	0.12	0.18	0.24	80	0.09	0.16	0.25	100	0.06	0.10	0.15	801	0.02	0.07	0.13	1001
X-n876-k59	99299	0.35	0.45	0.57	77	0.38	0.47	0.59	78	0.15	0.26	0.34	744	0.14	0.23	0.35	781
X-n895-k37	53860	0.51	0.76	1.03	74	0.45	0.61	0.80	83	0.18	0.33	0.47	760	0.19	0.33	0.49	852
X-n916-k207	329179	0.56	0.67	0.88	86	0.54	0.66	0.81	85	0.28	0.41	0.56	833	0.29	0.39	0.52	853
X-n936-k151	132715	0.69	0.90	1.05	59	0.70	0.93	1.34	71	0.38	0.54	0.78	581	0.34	0.53	0.75	690
X-n957-k87	85465	0.12	0.17	0.25	81	0.05	0.11	0.21	100	0.04	0.11	0.18	835	0.06	0.09	0.12	1021
X-n979-k58	118976	0.29	0.44	1.15	102	0.80	1.01	1.12	104	0.12	0.20	0.31	999	0.13	0.17	0.20	1064
X-n1001-k43	72355	0.53	0.76	1.07	71	0.50	0.66	0.83	78	0.20	0.30	0.46	717	0.13	0.27	0.45	777
Mean		0.37	0.53	0.72	75	0.34	0.50	0.69	82	0.20	0.32	0.46	763	0.17	0.29	0.43	831

Table 12: Computations on the 𝕏 dataset: p-values for FILO2 vs FILO on the left and FILO2 (long) vs FILO (long) on the right.

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, FILO2 is not statistically different from FILO when H_0 cannot be rejected (Similar), FILO2 is statistically better when both H_0 and H_1 are rejected (Better), and, finally, FILO2 is statistically worse when H_0 is rejected and H_1 is not rejected (Worse).

	FILO2 vs FILO			
	Small	Medium	Large	All
H_0	0.809330	**0.000786**	**0.002750**	**0.000030**
H_1	0.404665	**0.000393**	**0.001375**	**0.000015**
	Similar	Better	Better	Better

	FILO2 (long) vs FILO (long)			
	Small	Medium	Large	All
H_0	0.626246	**0.000600**	**0.000178**	**0.000001**
H_1	0.313123	**0.000300**	**0.000089**	**0.000001**
	Similar	Better	Better	Better

Table 12 shows the p-values of the Wilcoxon test on 𝕏 computations. We considered 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 $\alpha = \alpha_0/8 = 0.003125$. In particular we have that

- FILO2 and FILO behave similarly on the small instances.
- FILO2 performs better than FILO on medium and large instances, and on the overall dataset.
- FILO2 (long) and FILO (long) behave similarly on the small instances.
- FILO2 (long) performs better than FILO (long) on medium and large instances, and on the overall dataset.

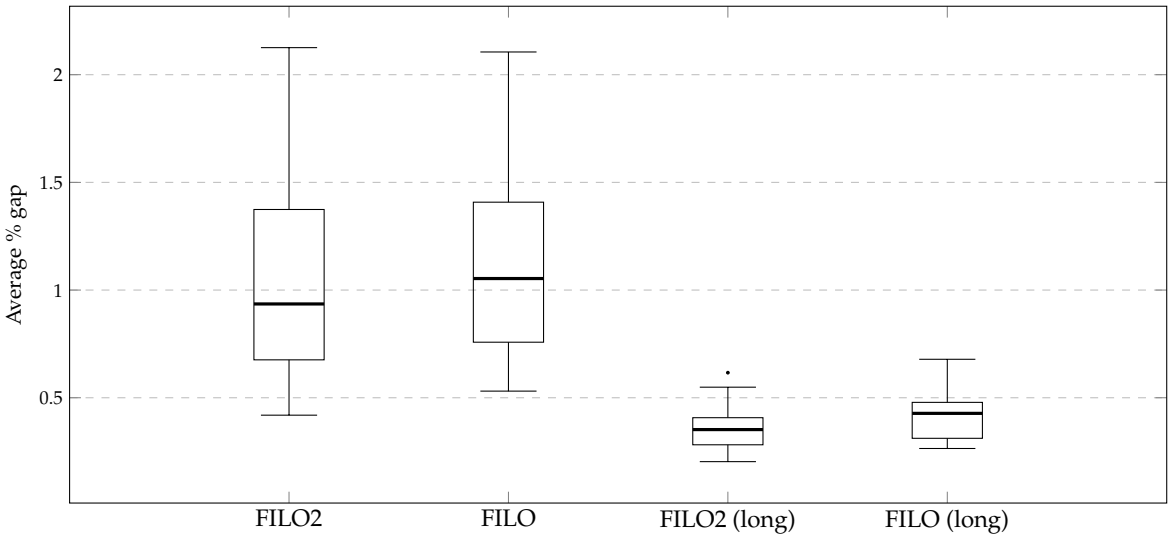


Figure 7: Comparison of average gaps obtained by algorithms on the 𝔹 dataset.

Table 13: Computations on 𝔹 instances.

		FILO				FILO2				FILO (long)				FILO2 (long)			
ID	BKS	Best	Avg	Worst	t	Best	Avg	Worst	t	Best	Avg	Worst	t	Best	Avg	Worst	t
L1	192848	0.47	0.53	0.65	95	0.35	0.42	0.50	98	0.19	0.26	0.31	958	0.16	0.20	0.24	992
L2	111391	0.81	1.03	1.20	142	0.57	0.85	1.34	130	0.40	0.49	0.57	1515	0.17	0.32	0.46	1479
A1	477277	0.54	0.61	0.66	124	0.49	0.52	0.56	105	0.25	0.29	0.33	1247	0.22	0.25	0.29	1073
A2	291350	1.02	1.10	1.22	139	0.96	1.08	1.27	114	0.33	0.44	0.52	1505	0.28	0.40	0.50	1226
G1	469531	0.70	0.74	0.76	161	0.60	0.66	0.72	114	0.27	0.30	0.34	1602	0.25	0.27	0.29	1180
G2	257748	1.42	1.51	1.60	198	1.30	1.47	1.61	127	0.39	0.44	0.53	2284	0.33	0.41	0.49	1505
B1	501719	1.02	1.08	1.13	197	0.94	1.02	1.10	113	0.38	0.42	0.46	2006	0.35	0.39	0.42	1206
B2	345481	1.90	2.01	2.15	228	1.85	1.91	2.01	122	0.52	0.58	0.66	2564	0.46	0.55	0.62	1430
F1	7240124	0.77	0.81	0.85	303	0.69	0.73	0.78	136	0.32	0.35	0.38	3276	0.30	0.31	0.33	1575
F2	4373320	1.99	2.11	2.23	482	1.94	2.13	2.21	153	0.60	0.68	0.83	6194	0.56	0.62	0.72	2040
Mean		1.06	1.15	1.24	207	0.97	1.08	1.21	121	0.36	0.42	0.49	2315	0.31	0.37	0.44	1371

B.2 Computational Details for 𝔹 Instances

Detailed results about computations on the 𝔹 dataset can be found in Table 13 and Figure 7.

Table 14: Computations on the $\mathbb{B}$ dataset: p-values for FILO2 vs FILO on the left and FILO2 (long) vs FILO (long) on the right.

p-values in bold are associated with rejected hypothesis when $\alpha = 0.0125$.

The last row of each group contains a p-value interpretation when $\alpha = 0.0125$. In particular, FILO2 is not statistically different from FILO when H_0 cannot be rejected (Similar), FILO2 is statistically better when both H_0 and H_1 are rejected (Better), and, finally, FILO2 is statistically worse when H_0 is rejected and H_1 is not rejected (Worse).

	FILO2 vs FILO
H_0	0.064453
H_1	0.032227
	Similar

	FILO2 (long) vs FILO (long)
H_0	**0.001953**
H_1	**0.000977**
	Better

Table 14 shows the p-values of the Wilcoxon test on $\mathbb{B}$ computations. We considered a total number of $n = 2$ hypotheses corresponding to H_0 and H_1. Thus, by assuming an initial significance level $\alpha_0 = 0.025$, the adjusted value becomes $\alpha = \alpha_0/2 = 0.0125$. In particular we have that

- FILO2 and FILO behave similarly.
- FILO2 (long) performs better than FILO (long).

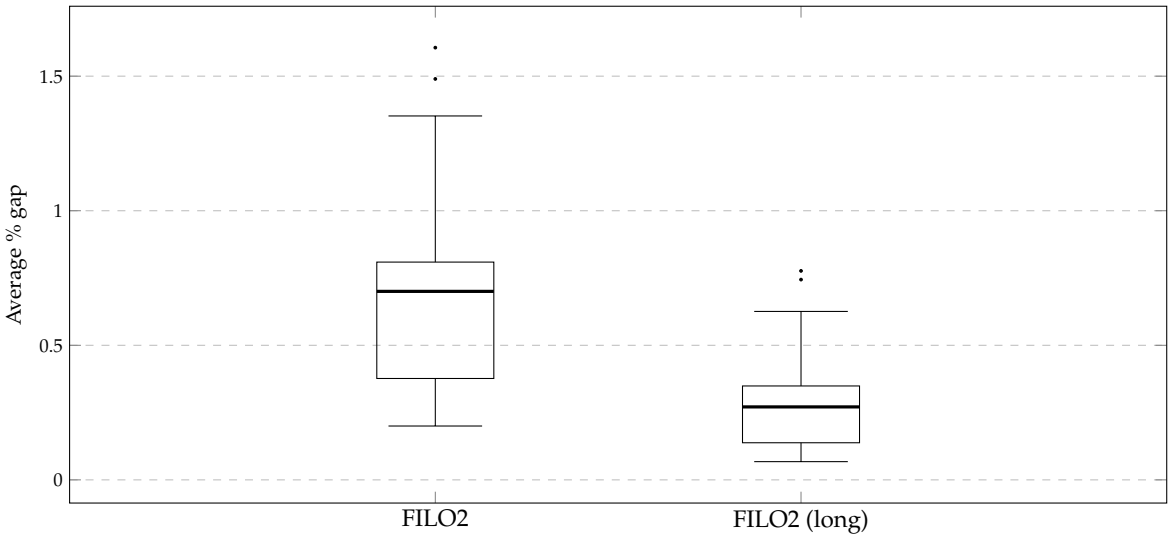


Figure 8: Comparison of average gaps obtained by algorithms on the $\mathbb{I}$ dataset.

B.3 Computational Details for $\mathbb{I}$ Instances

Detailed results about computations on the $\mathbb{I}$ dataset can be found in Table 15 and Figure 8.

Table 15: Computations on $\mathbb{I}$ instances.

		FILO2				FILO2 (long)			
ID	BKS	Best	Avg	Worst	t	Best	Avg	Worst	t
Valle-D-Aosta	21679514	0.22	0.28	0.36	192	0.08	0.13	0.19	2620
Molise	111184982	0.36	0.41	0.47	146	0.13	0.16	0.18	1901
Trentino-Alto-Adige	102063181	0.81	0.92	1.08	194	0.31	0.40	0.49	2292
Basilicata	175623919	0.59	0.74	0.92	185	0.16	0.31	0.45	2722
Umbria	545507981	0.26	0.29	0.31	248	0.11	0.12	0.14	2124
Abruzzo	311712556	0.66	0.73	0.80	278	0.24	0.28	0.34	2613
Friuli-Venezia-Giulia	415805616	0.54	0.60	0.70	393	0.19	0.25	0.32	4116
Liguria	1426389867	0.18	0.20	0.22	332	0.06	0.07	0.07	2964
Calabria	1964651530	0.36	0.40	0.42	292	0.11	0.13	0.17	2299
Marche	420484426	0.65	0.69	0.74	408	0.22	0.26	0.31	3634
Sardegna	827934149	1.43	1.61	1.78	316	0.61	0.78	0.97	2690
Campania	391859276	0.93	1.01	1.10	420	0.33	0.37	0.41	4417
Piemonte	2627446164	0.30	0.32	0.35	404	0.14	0.15	0.17	2721
Toscana	1084417188	0.69	0.77	0.85	488	0.28	0.33	0.40	5793
Puglia	1464797603	1.36	1.49	1.59	436	0.64	0.74	0.89	2957
Sicilia	1774262462	1.23	1.35	1.42	469	0.50	0.63	0.75	3298
Veneto	1050488613	0.62	0.71	0.80	600	0.25	0.29	0.34	5131
Emilia-Romagna	5405446715	0.22	0.24	0.25	501	0.09	0.09	0.10	3263
Lombardia	1339900081	0.70	0.75	0.79	562	0.29	0.34	0.39	7990
Lazio	3145381332	0.37	0.40	0.43	532	0.12	0.14	0.15	3938
Mean		0.62	0.70	0.77	370	0.24	0.30	0.36	3474

References

[1] Luca Accorsi and Daniele Vigo. "A Hybrid Metaheuristic for Single Truck and Trailer Routing Problems". In: *Transportation Science* 54.5 (2020), pp. 1351–1371. eprint: https://doi.org/10.1287/trsc.2019.0943.

[2] Luca Accorsi and Daniele Vigo. "A fast and scalable heuristic for the solution of large-scale capacitated vehicle routing problems". In: *Transportation Science* 55.4 (2021), pp. 832–856.

[3] Tobias Achterberg, Timo Berthold, and Gregor Hendel. "Rounding and Propagation Heuristics for Mixed Integer Programming". In: *Operations Research Proceedings 2011*. Ed. by Diethard Klatte, Hans-Jakob Lüthi, and Karl Schmedders. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 71–76. ISBN: 978-3-642-29210-1.

[4] Florian Arnold, Michel Gendreau, and Kenneth Sörensen. "Efficiently solving very large-scale routing problems". In: *Computers & Operations Research* 107 (2019), pp. 32–42. ISSN: 0305-0548.

[5] Florian Arnold and Kenneth Sörensen. "Knowledge-guided local search for the vehicle routing problem". In: *Computers & Operations Research* 105 (2019), pp. 32–46. ISSN: 0305-0548.

[6] Jon Louis Bentley. "K-d trees for semidynamic point sets". In: *Proceedings of the sixth annual symposium on Computational geometry*. 1990, pp. 187–197.

[7] G. Clarke and J. W. Wright. "Scheduling of Vehicles from a Central Depot to a Number of Delivery Points". In: *Operations Research* 12.4 (1964), pp. 568–581. eprint: https://doi.org/10.1287/opre.12.4.568.

[8] R. Dennis Cook. "Detection of Influential Observation in Linear Regression". In: *Technometrics* 19.1 (1977), pp. 15–18. ISSN: 00401706.

[9] CVRPLIB. *Capacitated Vehicle Routing Problem Library*. visited on 2022-12-22. 2022.

[10] G. B. Dantzig and J. H. Ramser. "The Truck Dispatching Problem". In: *Management Science* 6.1 (1959), pp. 80–91. eprint: https://doi.org/10.1287/mnsc.6.1.80.

[11] DIMACS. *12th DIMACS Implementation Challenge*. visited on 2023-01-19. 2022.

[12] Olive Jean Dunn. "Multiple Comparisons Among Means". In: *Journal of the American Statistical Association* 56.293 (1961), pp. 52–64. ISSN: 01621459.

[13] Fred Glover. "Ejection chains, reference structures and alternating path methods for traveling salesman problems". In: *Discrete Applied Mathematics* 65.1 (1996). First International Colloquium on Graphs and Optimization, pp. 223–253. ISSN: 0166-218X.

[14] Stefan Irnich, Birger Funke, and Tore Grünert. "Sequential search and its application to vehicle-routing problems". In: *Computers & Operations Research* 33.8 (2006), pp. 2405–2429. ISSN: 0305-0548.

[15] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing". In: *Science* 220.4598 (1983), pp. 671–680. ISSN: 0036-8075. eprint: https://science.sciencemag.org/content/220/4598/671.full.pdf.

[16] Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle. "Iterated Local Search". In: *Handbook of Metaheuristics*. Boston, MA: Springer US, 2003, pp. 320–353. ISBN: 978-0-306-48056-0.

[17] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. USA: John Wiley & Sons, Inc., 1990. ISBN: 0471924202.

[18] N. Mladenović and P. Hansen. "Variable neighborhood search". In: *Computers & Operations Research* 24.11 (1997), pp. 1097–1100. ISSN: 0305-0548.

[19] OpenAddresses. *The free and open global address collection*. visited on 2022-12-10. 2022.

[20] Gerhard Reinelt and Giovanni Rinaldi. “The traveling salesman problem”. In: 1994.

[21] Alberto Santini et al. “Decomposition Strategies for Vehicle Routing Heuristics”. In: *INFORMS Journal on Computing* 0.0 (2023), null. eprint: https://doi.org/10.1287/ijoc.2023.1288.

[22] Michael Schneider, Fabian Schwahn, and Daniele Vigo. “Designing granular solution methods for routing problems with time windows”. In: *European Journal of Operational Research* 263.2 (2017), pp. 493–509.

[23] Gerhard Schrimpf et al. “Record Breaking Optimization Results Using the Ruin and Recreate Principle”. In: *J. Comput. Phys.* 159.2 (Apr. 2000), pp. 139–171. ISSN: 0021-9991.

[24] Éric Taillard et al. “A Tabu Search Heuristic for the Vehicle Routing Problem with Soft Time Windows”. In: *Transportation Science* 31.2 (1997), pp. 170–186. eprint: https://doi.org/10.1287/trsc.31.2.170.

[25] Paolo Toth and Daniele Vigo. “The Granular Tabu Search and Its Application to the Vehicle-Routing Problem”. In: *INFORMS Journal on Computing* 15.4 (2003), pp. 333–346.

[26] Paolo Toth and Daniele Vigo. *Vehicle Routing*. Ed. by Daniele Vigo and Paolo Toth. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2014. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611973594.

[27] Eduardo Uchoa et al. “New benchmark instances for the Capacitated Vehicle Routing Problem”. In: *European Journal of Operational Research* 257.3 (2017), pp. 845–858. ISSN: 0377-2217.

[28] Thibaut Vidal. “Hybrid Genetic Search for the CVRP: Open-Source Implementation and SWAP* Neighborhood”. In: *Comput. Oper. Res.* 140.C (Apr. 2022). ISSN: 0305-0548.

[29] Thibaut Vidal et al. “A Hybrid Genetic Algorithm for Multidepot and Periodic Vehicle Routing Problems”. In: *Operations Research* 60.3 (2012), pp. 611–624. eprint: https://doi.org/10.1287/opre.1120.1048.

[30] Frank Wilcoxon. “Individual Comparisons by Ranking Methods”. In: *Biometrics Bulletin* 1.6 (1945), pp. 80–83. ISSN: 00994987.

[31] Emmanouil E. Zachariadis and Chris T. Kiranoudis. “A Strategy for Reducing the Computational Complexity of Local Search-based Methods for the Vehicle Routing Problem”. In: *Comput. Oper. Res.* 37.12 (Dec. 2010), pp. 2089–2105. ISSN: 0305-0548.