Alma Mater Studiorum - Università di Bologna

DOTTORATO DI RICERCA IN

INGEGNERIA BIOMEDICA, ELETTRICA E DEI SISTEMI

Ciclo 34

Settore Concorsuale: 01/A6 - RICERCA OPERATIVA

Settore Scientifico Disciplinare: MAT/09 - RICERCA OPERATIVA

INNOVATIVE HYBRID APPROACHES FOR VEHICLE ROUTING PROBLEMS

Presentata da: Luca Accorsi

Coordinatore Dottorato

Michele Monaci

Supervisore

Daniele Vigo

Co-supervisore

Michele Lombardi

Esame finale anno 2022

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Abstract

Dipartimento di Ingegneria dell'Energia Elettrica e dell'Informazione
"Guglielmo Marconi"

Ingegneria Biomedica, Elettrica e dei Sistemi
(Curriculum Ricerca Operativa)

**Innovative Hybrid Approaches for
Vehicle Routing Problems**

by Luca ACCORSI

This thesis deals with the efficient resolution of Vehicle Routing Problems (VRPs). The routing of vehicles is a core activity in logistics and with the rise and continuous growth of e-commerce shopping, being able to tackle problem instances having larger and larger sizes is becoming increasingly important. The first chapter of the thesis deals with the archetype of all VRPs: the Capacitated Vehicle Routing Problem (CVRP). Despite having being introduced more than 60 years ago, it still remains an extremely challenging problem. In this chapter I design a Fast Iterated-Local-Search Localized Optimization algorithm for the CVRP, shortened to FILO. The simplicity of the CVRP definition allowed me to experiment with advanced local search acceleration and pruning techniques that have eventually became the core optimization engine of FILO. Thanks to them, FILO experimentally shown to be extremely scalable and able to solve very large scale instances of the CVRP in a fraction of the computing time compared to existing state-of-the-art methods, still obtaining competitive solutions in terms of their quality. The chapter also contains a detailed analysis of all components of FILO, which I hope can be useful for designing scalable algorithms for other VRP variants as well as other combinatorial optimization problems. The second chapter deals with a novel extension of the CVRP called the Extended Single Truck and Trailer Vehicle Routing Problem, or simply XSTTRP. The XSTTRP models a broad class of VRPs in which a single vehicle, composed of a truck and a detachable trailer, has to serve a set of customers with accessibility constraints making some of them not reachable by using the entire vehicle. This problem moves towards VRPs including more realistic constraints and it models scenarios such as parcel deliveries in crowded city centers or rural areas, where maneuvering a large vehicle is forbidden or dangerous. The XSTTRP generalizes several well known VRPs such as the Multiple Depot VRP and the Location Routing Problem. For its solution I developed an hybrid metaheuristic which combines a fast heuristic optimization with a polishing phase based on the resolution of a limited set partitioning problem. Finally, the thesis includes a final "reference chapter" to guide the computational study section of new approaches to VRPs proposed by the machine learning community.

Acknowledgements

I want to thank my supervisor Prof. Daniele Vigo for supporting me during this Ph.D. journey and for giving me the opportunity to work on exciting and challenging topics. Without his guidance this work would have not been the same. Many thanks to my co-supervisor Prof. Michele Lombardi for sharing with me ideas and possible interesting research paths.

I would also like to thank the co-authors of my scientific publications: Prof. Andrea Lodi, Prof. Michele Monaci and Francesco Cavaliere. Working with them has been a pleasure.

I also want to thank all the other professors of the Operations Research group of the Department of Electrical, Electronic and Information Engineering "Guglielmo Marconi" of the University of Bologna: Prof. Valentina Cacchiani, Prof. Enrico Malaguri, Prof. Silvano Martello, Prof. Paolo Toth I would also thank all friends and colleagues for the nice time were shared together: Dr. Carlos Emilio Contreras Bolton, Francesco Cavaliere, Silvia Anna Cordieri, Cristiano Fabbri, Naga Venkata Chaitanya Gudapati, Henri Lefebvre, Federico Naldini, Alan Osorio Mora, Dr. Paolo Paronuzzi, Antonio Punzo, Carlos Rodrigo Rey Barra.

Finally, a special thanks goes to my parents without whom none of this could have happened, and to my girlfriend Daniela for supporting me during these years.

Contents

Abstract **i**

Acknowledgements **iii**

1 Introduction **1**
- 1.1 Vehicle Routing Problems 1
- 1.2 Solution of Vehicle Routing Problems 3
- 1.3 Research Contribution 4

2 Heuristic Solution of Large-Scale CVRPs **9**
- 2.1 Introduction 9
- 2.2 Solution Approach 12
  - 2.2.1 Construction 12
  - 2.2.2 Local Search Engine 13
    - Hierarchical Randomized Variable Neighborhood Descent 14
    - Move Generators and Granular Neighborhoods 15
    - Static Move Descriptors 16
    - Selective Vertex Caching 17
  - 2.2.3 Improvement 19
    - Route Minimization 19
    - Core Optimization 21
- 2.3 Computational Results 26
  - 2.3.1 Implementation and Experimental Environment 26
  - 2.3.2 Parameter Tuning 26
  - 2.3.3 Testing on $\mathbb{X}$ Instances 28
  - 2.3.4 Testing on Very Large-Scale Instances 30
- 2.4 Algorithmic Components Analysis 33
  - 2.4.1 Initial Solution Definition 33
  - 2.4.2 Local Search 34
  - 2.4.3 Shaking Guiding Strategy 36
  - 2.4.4 Move Generators and Granular Neighborhoods 37
  - 2.4.5 Selective Vertex Caching 39
  - 2.4.6 Extreme Runs 39
  - 2.4.7 Simplified Versions of FILO 41
- 2.5 Conclusions 42
- 2.6 Acknowledgments 43
- A Algorithmic Components Analysis: Additional Material 44
  - A.1 Initial Solution Definition 44
  - A.2 Local Search 44
  - A.3 Extreme Runs 46
  - A.4 Computations with Limited Memory Footprint 46
- B Move Generators and Static Move Descriptors 48

B.1 Move Generators: Storage and Management . . . 48
B.2 An Abstract SMD-based Local Search Operator . . . 49
Dynamic Vertex-wise Move Generators for SMD-based Local Search Operators . . . 51
B.3 SMD Implementation Details of Local Search Operators . . . 52
TWOPT . . . 52
SPLIT . . . 53
TAILS . . . 53
n0EX with $n \geq 1$ . . . 54
n0REX with $n \geq 2$ . . . 54
nmEX with $1 \leq m \leq n$ . . . 55
nmREX (and nmREX*) with $1 \leq m \leq n$ . . . 56
EJCH . . . 57
C Computational details for $\mathbb{X}$ instances . . . 60
D Computational details for very large-scale instances . . . 66

3 Heuristic Solution of STTRPs **71**
1 Introduction . . . 71
2 Literature review . . . 72
3 Problem description . . . 74
3.1 Comparative analysis of related problems . . . 75
4 Solution approach . . . 76
4.1 Assignment . . . 78
4.2 Construction . . . 79
4.3 Improvement . . . 79
Granular neighborhoods. . . . 81
4.4 Polishing . . . 82
4.5 Extensions to handle specific sub-problems . . . 83
5 Computational results . . . 84
5.1 Implementation and experimental environment . . . 85
5.2 Parameter definitions . . . 85
5.3 Testing on MDVRP instances . . . 86
5.4 Testing on LRP instances . . . 87
5.5 Testing on STTRPSD instances . . . 89
5.6 Testing on STTRP instances . . . 91
5.7 Testing on XSTTRP instances . . . 91
6 Algorithm components analysis . . . 93
6.1 Assignment fitness functions . . . 95
6.2 Local search . . . 96
6.3 Granular neighborhoods adoption . . . 97
6.4 Set-partitioning post-optimization . . . 98
7 Conclusions . . . 98
8 Acknowledgments . . . 99

4 Guidelines for the Computational Testing of ML approaches to VRPs **101**
1 Introduction . . . 101
2 Guidelines for a Computational Testing . . . 102
2.1 Benchmark Instances and Problem Definition . . . 102
2.2 Baseline Algorithms . . . 103
2.3 Algorithms Comparison . . . 104
3 Examples of a Computational Studies . . . 107

3.1 Scenario 1: all algorithms source codes are available 107
3.2 Scenario 2: at least one algorithm source code is not available 107
3.3 Statistical validation of the results 108
4 Conclusions 109
5 Acknowledgments 110

5 Conclusions **113**

Bibliography **115**

Dedicated to my parents, Daniela and Chicco

Chapter 1

Introduction

1.1 Vehicle Routing Problems

Vehicle routing problems (VRPs) are among the most studied combinatorial optimization problems and are nowadays very relevant due to their huge impact in real-world applications.

Transportation of goods or people is, in fact, a pervasive theme in our daily life. As an example, more and more people are making use of e-commerce shopping. The COVID-19 pandemic started in the late 2019 and peaked in 2020 caused online retail, which was already greatly used, to grew massively especially during lockdown periods. Moreover, given the convenience for the final customers, this trend is expected to continue over the next decades. Indeed, the ability of e-commerce retailers to readily process and deliver goods make them very appealing to users that, without moving from home, can with a few clicks order an item and receive it during the next few hours. Planning of delivery routes is a fundamental step in this process. Moreover, being able to handle larger and larger volumes of requests is already of great importance nowadays and it will become crucial in future given the increasing usage of these platforms. The need for highly efficient solvers able to find, in short computing time, routes serving effectively thousand of customers is thus becoming increasingly relevant.

Most VRPs rely on a graph-based definition. More formally, given a graph $G = (V, E)$ where V is the set of vertices representing locations of interest (for example customers and depot locations), and E is a set of edges connecting these vertices. A value c_{ij} with $i, j \in V$ is associated with each arc $(i, j) \in E$ and represents the cost for going from i to j. Moreover, these costs are symmetric, that is $c_{ij} = c_{ji}$. Basic VRPs consider G to be complete, thus having an edge between every pair of vertices, and undirected, since each edge can be traversed in both directions. The cost of a VRP solution $\mathcal{S}$ is given by the sum of the cost of all edges defining it, that is, $\textsc{Cost}(\mathcal{S}) = \sum_{(i,j)\in\mathcal{S}} c_{ij}$. The objective consists in finding a solution with the minimum cost.

The above description provides a common base for several highly-relevant VRPs. On top of this base, problem-specific constraints can be added to model particular scenarios casting this abstract and general problem into a precise one.

Traveling Salesman Problem The most well known combinatorial optimization problem arising in the VRP area is probably the Traveling Salesman Problem (TSP).

The TSP requires to find the routing for a salesman who starts from a origin vertex in G, visits the remaining vertices, and returns to the origin one in such a way that each vertex is visited exactly once and the total distance traveled is a minimum.

Applications of the TSP, however, go far beyond the route planning problem of a traveling salesman. They span, in fact, over several areas of knowledge from mathematics and computer science to genetics and electronics (see Gutin and Punnen (2006) for a comprehensive description of the TSP and its applications).

The TSP, despite not solved directly in its pure abstract definition, is among the fundamental subproblems contained into the core of routing problems faced in this thesis.

Capacitated Vehicle Routing Problem In 1959, Dantzig and Ramser introduced the *truck dispatching problem* as a generalization of the TSP with the aim of modeling a real-world application concerning the delivery of gasoline to gas stations.

The TSP is generalized by attaching an additional label q_i to each vertex $i \in V$ and considering one of the vertex in G, say node $0 \in V$ to be the origin vertex with a label $q_0 = 0$. Vertex 0 is called *depot* and the remaining vertices $i \in V \setminus \{0\}$ are said *customers*. The value q_i represents the quantity of a certain product required by customer i. Moreover, the vehicle, which is initially located at the depot, has a maximum capacity Q. When $Q \geq \sum_{i \in V} q_i$, all customers can be served by a single route starting from the depot, visiting all customers and coming back to the depot (and thus the problem can be reduced to a TSP). On the other hand, when $Q \ll \sum_{i \in V} q_i$, more than one vehicle is required to serve all the customers and multiple routes must be defined.

The truck dispatching problem is nowadays known as the Capacitated Vehicle Routing Problem (CVRP) and since the 1959 has been among the most studied VRPs. In Chapter 2 I will develop a solution approach specifically tailored for CVRPs having several thousand of customers.

The CVRP, which has primarily an academic relevance, belongs to what is known as the *VRP family*. This family contains a great variety of problems that, by extending the standard CVRP, model scenarios having a more concrete applicability in the real world. Among the most studied variants we have the VRP with Time Windows, in which customers have to be visited during specific time intervals, the Heterogeneous VRP, in which vehicles possibly have different capacities and a fixed cost for their usage, and finally, the VRP with Pickup and Deliveries, where goods or passengers have to be transported from different origins to different destinations. Constraints and objectives arising from real-world applications are studied in the so-called multi-attribute VRPs (see e.g., Vidal et al. (2013)).

The problem described in the following section goes in the direction of a more realistic VRP.

Truck and Trailer Routing Problem By considering vehicles composed of a truck and a trailer, and introducing incompatibilities between customers and (part of) vehicles, Chao (2002) introduced the family of Truck and Trailer Routing Problem (TTRP) as a generalization of the CVRP.

In the TTRP, some customers cannot be reached by the whole vehicle, for example because located in crowded city centers where maneuvering the complete vehicle is difficult or forbidden. It is thus necessary to park the trailer on appropriate parking locations, serve one or more customers and return to retrieve the parked trailer.

In Chapter 3 I deal with a variation of the TTRP, called the Extended Single TTRP (XSTTRP) which directly generalizes the CVRP by introducing parking locations, where the trailer can be possibly detached, and specializing the role of customers in G in truck customers and vehicle customers with and without parking location. Truck customers can be visited by the truck only, while vehicle customers can be visited both by the whole vehicle as well as by the truck only. Moreover, some vehicle customers can be used as parking locations. Thanks to the flexibility offered by the XSTTRP, several well-known VRPs, such as the Multiple Depot VRP (MDVRP, Cordeau, Gendreau, and Laporte (1997)) and the Location Routing Problem (LRP, Schneider and Drexl (2017)), can be modeled within this framework.

1.2 Solution of Vehicle Routing Problems

The TSP belongs to the class of $\mathcal{NP}$-hard problems. It is thus very unlikely that a polynomial time algorithm exists which is able to solve a generic instance. The CVRP as well as the XSTTRP, containing the TSP as a subproblem, are also $\mathcal{NP}$-hard.

Even though problems are classified as $\mathcal{NP}$-hard, it does not mean that finding a proven optimal solution is impossible. Indeed, enormous advances have been made in the area of exact solution approaches since the introduction of the truck dispatching problem by Dantzig and Ramser (1959). State-of-the-art exact solvers consists of branch-cut-and-price algorithms, which combine column and cut generation to effectively explore the space of all possible solutions. These approaches can consistently solve CVRP instances with up to 250 customers and in some cases even larger instances (Baldacci, Toth, and Vigo (2007), Pecin et al. (2017), Pessoa et al. (2020), and Pecin et al. (2014)). Moreover, specialized routing solvers such as VRPSolver by Pessoa et al. (2020), containing cutting-edge algorithms, are freely available for research purposes providing a great value and opportunities for the entire VRP community.

In order to solve VRPs having a larger size within a reasonable computational effort one must however rely on heuristic approaches. More precisely, the evolution of VRP heuristics over the past years has mostly taken place within the context of meta and math heuristics. A metaheuristic (see Talbi (2009)) balances intensification and diversification phases to explore promising regions of the search space trying to escape from local optima. On the other hand, a matheuristic (see Boschetti et al. (2009)) sees the interoperation of metaheuristics and mathematical programming techniques combining the flexibility of the former with the effectiveness of the latter.

As I have briefly mentioned above, most of the research in the solution of large scale VRPs consists in meta and math heuristics. Among the most successfully used paradigms we find Genetic Algorithms (Vidal et al. (2012) and Vidal (2022)), Iterated Local Search (Subramanian, Uchoa, and Ochi (2013)), and Guided Local Search (Arnold and Sörensen (2019)). The common theme in the above approaches is the usage of local search that when coupled with effective diversification techniques is able to find high-quality solutions. However, among the notable exceptions to local-search based approaches, we find algorithm SISR (*Slack Induction by String Removals*) proposed by Christiaens and Vanden Berghe (2020). Indeed, SISR combines an extremely well-designed ruin-and-recreate procedure with a diversification mechanism based on the simulated annealing paradigm. SISR achieves several desirable properties of metaheuristics (see Chapter 4 of Toth and Vigo (2014)) such as flexibility, conceptual and implementation simplicity, and effectiveness into an unique solution framework able to tackle a great variety of VRPs.

Moreover, I want to mention a recently ri-discovered trend consisting of so-called polishing algorithms based on the POPMUSIC paradigm (Taillard and Voss (2002)) that, starting from near-optimal solutions, explore very large neighborhoods through the resolution of carefully selected subproblem typically by means of exact methods (Toth and Tramontani (2008), Queiroga, Sadykov, and Uchoa (2020), and Cavaliere, Bendotti, and Fischetti (2020a)).

The intensification phase of most state-of-the-art metaheuristics for VRPs includes local search procedures as the fundamental ingredient to readily find high-quality solutions.

The trend of the last few years is that of moving to very large scale instances with several thousand of customers. In these scenarios, even standard local search procedures such as 2-opt or relocate exploring neighborhoods of quadratic cardinality may become the bottleneck of an algorithm if naively designed.

In this cases one may rely on the so-called local search *acceleration* and *pruning* techniques as dedicated tools to speed up local search execution.

In particular, an acceleration technique provides design guidelines on the efficient implementation of local search operators without reducing the scope of the search. Among the most popular acceleration techniques we have Static Move Descriptors, introduced by Zachariadis and Kiranoudis (2010) that replace the standard nested for-loop exploration of local search neighborhoods with specialized data structures and procedures, that avoid unnecessary neighbors re-evaluation by exploiting the locality of a local search move application. Alternatively, the Sequential Search proposed by Irnich, Funke, and Grünert (2006) breaks a local search move into basic blocks called partial moves. The execution of a sequence of partial moves can be aborted as soon as certain conditions are met, thus pruning in advance a nonpromising local search move.

A (heuristic) pruning technique, on the other hand, typically improves the execution speed possibly by reducing the scope of the search. It is thus crucial to find a good tradeoff between the efficiency and effectiveness of a local search operator making use of such techniques. Granular Neighborhoods (GN) introduced by Toth and Vigo (2003) is a popular pruning technique widely and successfully adopted in VRP algorithms. A GN is a restricted local search neighborhood in which special arcs associated with the instance, called move generators, are used to identify promising local search moves in which the edge associated with the move generator is inserted into the solution. Move generators are defined by a so-called sparsification rule. As an example, a simple sparsification rule considers as move generators all arcs (i, j) and (j, i) associated with an edge (i, j) such that c_{ij} is lower than a given threshold γ. The value of γ can also be changed during the algorithm execution, to implement a dynamic intensification of local search procedures.

An alternative way to deal with very large scale instances consists in considering smaller subproblems derived from the original one. A promising direction considers the usage of decomposition techniques in which statically defined portions of the instance are defined and solved independently (see, e.g., Santini et al. (2021)). Such an approach would also allow for a parallel optimization of the different areas, possibly reaching an horizontal algorithm scalability.

In Chapter 2, I will develop a simple, yet effective, technique called Selective Vertex Caching (SVC) that combines heuristic pruning and decomposition techniques and can be thought as a sort of GN counterpart, but for vertices. The idea behind SVC is to localize local search applications to well defined areas instead of optimizing the whole solution. In particular, each solution is designed to keep track of a limited set of interesting vertices. In our design, a vertex is considered to have some interest if it belongs to a solution area that recently underwent some changes. In Chapter 2, the SVC, which seamlessly integrates with GNs and SMDs, provides a dynamic and soft instance decomposition.

As will be clear to the reader that will go through the forthcoming chapters, the design of efficient local search procedures (through the local search acceleration and pruning techniques) has been among the main themes of my research activity.

1.3 Research Contribution

In this thesis I focused on basic and fundamental combinatorial optimization problems arising in the vehicle routing area. I purposely abstracted away real-world constraints to concentrate on the core of routing problems. This has allowed me to experiment and push to their limits the different techniques used, without the need to complicate their design and implementation to cope with additional constraints.

I believe, however, that most of the positive results obtained on the basic VRPs studied in this thesis could be reproduced on more realistic variants possibly even though that may require some non-negligible design and implementation development which is thus left as a possible future research direction.

Finally, since the research I did was mainly of experimental nature, I made available the software produced during these years that can be freely used for research purposes.

The thesis is structured as follows.

Chapter 2 We study local search acceleration and pruning techniques for the Capacitated Vehicle Routing Problem. As a result, we design a fast and scalable, yet effective, solution approach able to achieve state-of-the-art results on several well-known benchmark instances having a variety of sizes ranging from few hundreds to tens of thousands of customers.

This work has been done in collaboration with my supervisor prof. Daniele Vigo and it has eventually been published in Transportation Science with the title "A Fast and Scalable Heuristic for the Solution of Large-Scale Capacitated Vehicle Routing Problems".

Chapter 3 We define a problem that describes a broad class of vehicle routing problems that use a single vehicle, composed of a truck and a detachable trailer, to serve a set of customers having specific accessibility constraints. We then propose a matheuristic solution approach achieving good quality results also on very well-known and studied special cases of the general problem.

This work has been done in collaboration with my supervisor prof. Daniele Vigo and it has eventually been published in Transportation Science with the title "A Hybrid Metaheuristic for Single Truck and Trailer Routing Problems".

Chapter 4 Vehicle routing problems are increasingly receiving attention in the machine learning community and several very promising results have already been published. Despite of the relevance of the problems, the increasing attention it is receiving in the ML community and the promising results achieved so far, these techniques have not yet become widespread in the vehicle routing community. This may be because of their novelty and the great changes required to adopt them by the OR community that typically has a different background. However, at the same time, the computational testing performed by most of the proposed machine learning methods may not be very convincing with respect to the standard practices in the vehicle routing community. The current work deals with the second aspect by highlighting important points to consider during the computational testing, providing guidelines and methodologies that we believe are relevant from an operations research perspective.

This work has been done in collaboration with my supervisor prof. Daniele Vigo and prof. Andrea Lodi.

List of Publications and Working Papers

- Accorsi, Luca and Daniele Vigo (2020). *A Hybrid Metaheuristic for Single Truck and Trailer Routing Problems*. Transportation Science 2020 54:5, 1351-1371.

- Accorsi, Luca and Daniele Vigo (2021). *A Fast and Scalable Heuristic for the Solution of Large-Scale Capacitated Vehicle Routing Problems*. Transportation Science 2021 55:4, 832-856.
- Accorsi, Luca and Francesco Cavaliere and Michele Monaci and Daniele Vigo (2021). *Daily Planning of Acquisitions and Scheduling of Dynamic Downlinks for the PLATiNO-1 Satellite*, submitted to European Journal of Operational Research. Not included in this thesis.
- Accorsi, Luca and Andrea Lodi and Daniele Vigo (2021). *Guidelines for the Computational Testing of Machine Learning approaches to Vehicle Routing Problems*, submitted to Operations Research Letters.

Chapter 2

A Fast and Scalable Heuristic for the Solution of Large-Scale Capacitated Vehicle Routing Problems[1]

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

2.1 Introduction

The Capacitated Vehicle Routing Problem (CVRP) has been studied for several decades but still remains a challenging problem to solve in practice. Recently, several new benchmark instances having large and very large scale of this fundamental problem (see $\mathbb{X}$ dataset, Uchoa et al. (2017) and $\mathbb{B}$ dataset, Arnold, Gendreau, and Sörensen (2019)) have brought it into focus again in the vehicle routing scene.

The careful design and implementation of solution algorithms becomes of primary importance when dealing with large instances. Failing to find the best tradeoff between effectiveness and efficiency has dramatic effects which are much more noticeable than when dealing with smaller instances. While computer memory capacity is a less pressing problem every year, finding a solution of satisfactory quality within a reasonable computing time still remains the real challenge: algorithm designers cannot rely on continuous increments in the working frequency of future processors. Processing units have, in fact, almost hit their maximum physical speed. New chips are moving towards massive parallelization (see Etiemble (2018)), possibly initiating an age of new algorithms that make use of concurrent decomposition techniques.

[1]The results of this chapter appears in: L. Accorsi and D. Vigo, "A Fast and Scalable Heuristic for the Solution of Large-Scale Capacitated Vehicle Routing Problems", *Transportation Science*, 2021, 55:4, 832-856

The CVRP can be described by using an undirected graph $G = (V, E)$ where V is the vertex set and E is the edge set. The vertex set V is partitioned into $V = \{0\} \cup V_c$ where 0 is the depot and $V_c = \{1, \dots, N\}$ is a set of N customers. A cost c_{ij} is associated with each edge $(i, j) \in E$. Moreover, we assume that the cost matrix $\boldsymbol{c}$ satisfies the triangle inequality. For a vertex $i \in V$ and a subset of vertices $V' \in V$, we identify with $\mathcal{N}_i^k(V')$ the set of the k nearest neighbor vertices $j \in V'$ of i with respect to the cost matrix $\boldsymbol{c}$. The set $\mathcal{N}_i^k(V')$ is shortened to $\mathcal{N}_i(V')$ for the case $k = |V'|$. Each customer $i \in V_c$ requires an integer quantity $q_i > 0$ from the depot, and $q_0 = 0$. An unlimited fleet of homogeneous vehicles with capacity Q is located at the depot available to serve the customers.

Recalling that a Hamiltonian circuit is a closed cycle visiting a set of customers exactly once, a CVRP solution S is composed of a number $|S|$ of Hamiltonian circuits, called *routes*, starting from the depot, visiting a subset of customers and coming back to the depot. We identify with r_i the route of load q_{r_i} serving customer $i \in V_c$. A solution is feasible if all customers are visited exactly once and none of the vehicles exceed its capacity. The cost of a solution S is given by the sum of the cost of the edges defining the routes of S. Finally, the CVRP goal is to find the feasible solution with the minimum cost.

An analysis of the current best state-of-the-art CVRP algorithms having a termination criterion based on the number of iterations (namely, HGSADC by Vidal et al. (2012), ILS-SP by Subramanian, Uchoa, and Ochi (2013), and SISR by Christiaens and Vanden Berghe (2020)) shows that they often exhibit a non-linear growth of the computing time that may, in their current state, undermine their applicability to large-scale instances within a reasonable computational effort. Clearly, as described in Chapter 4 of Toth and Vigo (2014), computing time is just one among many, seldom conflicting, dimensions characterizing heuristic solution approaches. The quality of the solutions is often another of the most obvious criteria used to assess algorithm quality. In addition, scalability with respect to the instance size is another very valuable quality, especially when moving to very large-scale instances.

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

In fact, local search acceleration techniques represent a very promising direction in the design of scalable algorithms that are efficient but still retain their effectiveness. The local search component, for a local search-based solution method, is typically one of the most time-consuming. Naive implementations, e.g., those built on full neighbor enumeration, fail to be competitive even for medium-sized instances. Among the most popular acceleration techniques, Granular Neighborhoods (GNs), proposed by Toth and Vigo (2003), define a heuristic filtering of less promising neighbors. This approach has been experimentally shown to provide an excellent compromise between computing time and solution quality; see, e.g., Toth and Vigo (2003), Zachariadis and Kiranoudis (2010), Schneider, Schwahn, and Vigo (2017), and Accorsi and Vigo (2020). The Sequential Search proposed by Irnich, Funke, and Grünert (2006) breaks a local search move into basic blocks called partial moves. The execution of those partial moves can be aborted if certain conditions are met, thus pruning in advance a non-promising local search move. Finally, Static Move Descriptors (SMD), introduced by Zachariadis and Kiranoudis (2010) and later improved by Beek et al. (2018), provide a data-oriented approach to the local search execution that avoids unnecessary evaluations by exploiting the locality of a local search move application.

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

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

The main design contributions, embedded into the proposed solution algorithm, are the following:

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

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

The chapter is structured as follows. Section 4 describes the details of our solution approach. Section 2.3 provides the experimental results, and Section 6 offers an experimental analysis of the algorithm components. Finally, the Appendix contains supplemental material covering implementation details.

2.2 Solution Approach

The metaheuristic we propose, whose high-level pseudocode is shown in Algorithm 1, consists of a *construction* phase (Line 3), which builds an initial feasible solution using a restricted version of the savings algorithm (see Clarke and Wright (1964a)). Then, it follows an *improvement* phase (Lines 4-6) aimed at further enhancing the initial solution quality.

More precisely, the improvement phase may first employ a *route minimization* procedure, to possibly reduce the number of routes in the initial solution when it is considered to be using more routes than necessary. A *core optimization* procedure, which is the central part of our algorithm, then uses an iterative, and localized optimization scheme to further improve the solution quality. Both route minimization and core optimization follow the Iterated Local Search paradigm (ILS, see Lourenço, Martin, and Stützle (2003)) in which shakings, performed in a ruin-and-recreate fashion (see Schrimpf et al. (2000a)), and local search applications interleave for a prefixed number of iterations. The result is a Fast ILS Localized Optimization algorithm, shortened to FILO.

The following paragraphs provide a detailed description of the algorithm, which is the outcome of an iterative design process based on the analyses described in Section 6. In particular, we first describe the construction phase, then the local search engine, which is a central component of the improvement phase, and finally, the section ends with the definition of the improvement procedures.

Algorithm 1 High-level FILO structure

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

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

2.2.1 Construction

The initial solution is built using an adaptation of the well-known savings algorithm by Clarke and Wright (1964a). As was already shown by other authors (see, e.g., Arnold, Gendreau, and Sörensen (2019)), the savings computation, which is quadratic in nature, can be linearized by considering for each customer $i \in V_c$ a restricted number n_{cw} of neighbors $j \in \mathcal{N}_i^{n_{cw}}(V_c)$ for which the saving value is computed. By limiting the number of savings, one can speed up the construction process without significantly harming the quality and compactness (i.e., the number of routes) of initial solutions especially when working with very large-scale instances. Note in addition that, since the construction phase is executed only once per run, over-optimizing it does not substantially contribute to the efficiency of the whole algorithm. As suggested in Arnold, Gendreau, and Sörensen (2019), we set $n_{cw} = 100$, and we compute the savings values for the arcs connecting each customer i to its n_{cw} neighbor customers j using a lexicographic order for the customers so as to avoid symmetries. More precisely, this set is given by the arcs $\{(i,j) : i \in V_c, j \in \mathcal{N}_i^{n_{cw}}(\{j \in V_c : i < j\})\}$. In fact, as reported in Section 2.4.1, larger n_{cw} values did not create significant differences either in the quality or compactness of the solutions.

2.2.2 Local Search Engine

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

- an exchange of a contiguous sequence, or *path*, of n vertices with another path of m vertices belonging either to the same or a different route, see CROSS-exchange, Taillard et al. (1997a). In the following this exchange is referred to as nmEX. For example, 21EX identifies the case in which $n = 2$ and $m = 1$. In particular, we implement the neighborhoods associated with $n, m = 0, \ldots, 3$ such that $n \geq m$ and nmEX is equivalent to mnEX;
- variants for 20EX, 21EX, 22EX, 30EX, 31EX, 32EX and 33EX, in which the first path of n vertices is reversed before being exchanged, called nmREX;
- variants for 22EX, 32EX and 33EX, in which both paths are reversed before being exchanged, called nmREX*;
- an intra-route 2-opt procedure, called TWOPT, as it is designed for the Traveling Salesman Problem, see Jünger, Reinelt, and Rinaldi (1995);
- two inter-route adaptations of the 2-opt procedure, called TAIL and SPLIT, both working on two different routes at a time. By denoting with head, a path of vertices belonging to the initial part of a route, and with tail, a path of vertices belonging to the final part of a route, TAIL swaps the tail of the two involved routes at some point, whereas SPLIT cuts the two routes at some point, then it replaces the tail of the first route with the reversed head of the second route and the head of the second route with the reversed tail of the first route;
- finally, an ejection-chain procedure, called EJCH, implementing the first improving sequence of 10EX, found by exploring a restricted number of sequences. That is, starting from an initial 10EX, a tree of at most n_{EC} nodes representing partial sequences is built. The sequence with the most improving value is always explored first, and a number of relocations (10EX) are generated by ejecting customers that restore the feasibility of the current route sequence endpoint. A 10EX may visit the same route more than once and no limit on the length of a sequence is imposed. As soon as a sequence is found to restore the feasibility of the target route, the associated 10EX exchanges are implemented. For more details, see Appendix B.3.

The above operators are structured in a Hierarchical Randomized Variable Neighborhood Descent (see Section 2.2.2), whose aim is to define a balanced combination of intensification-diversification, likelihood of escaping from local optima, and execution efficiency.

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

Hierarchical Randomized Variable Neighborhood Descent

We propose a local search architecture based on a combination of the Variable Neighborhood Descent (VND, see, e.g., Duarte et al. (2018)) and the Randomized VND (RVND, see, e.g., Subramanian, Uchoa, and Ochi (2013)) principles.

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

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

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

The Hierarchical RVND (HRVND) we propose mixes the two principles by defining a slightly more structured neighborhood exploration strategy, in which the operator order is neither completely random nor fixed a priori. More precisely, local search operators are arranged in *tiers*. Each tier is a compound operator that applies its subset of local search operators by following the RVND principles.

The overall HRVND links the tiers together once they have been ordered according to the criteria defined by the VND, such as the overall computational complexity of the operators involved in the tier.

The HRVND can thus be seen as a standard VND in which each tier is a compound local search operator and where successively more expensive tiers are used to escape from the local optima of the previous ones.

The proposed HRVND local search applies the operators described in Section 2.2.2 organized in the following two tiers: (i) 10EX, 11EX, SPLIT, TAILS, TWOPT, 20EX, 21EX, 22EX, 20REX, 21REX, 22REX, 22REX*, 30EX, 31EX, 32EX, 33EX, 30REX, 31REX, 32REX, 33REX, 32REX* and 33REX*, and (ii) EJCH.

The first tier contains operators defining neighborhoods of quadratic cardinality and having very similar execution times; whereas the second tier contains the most expensive operator employed by the local search engine. More details about computing times and improving power are provided in the analysis in Section 2.4.2 and in Section A.2 of the Appendix.

Each tier stores its operators in a circular list which is shuffled before the application (see Algorithm 2). The neighborhood associated with an operator is completely explored and all the improvements are applied before moving to the next operator on the list. More details about the neighborhood exploration are given in Section 2.2.2 and in Section B.2 of the Appendix. The

Algorithm 2 HRVND tier application

```
1: procedure TIERAPPLICATION($S, \mathcal{O}, \mathcal{R}$)
2:     $\mathcal{O} \leftarrow$ SHUFFLE($\mathcal{O}, \mathcal{R}$)
3:     $e \leftarrow 0, c \leftarrow 0$
4:     repeat
5:         $S' \leftarrow$ APPLY($\mathcal{O}_c, S$)
6:         if COST($S'$) $<$ COST($S$) then $S \leftarrow S', e \leftarrow c$
7:         $c \leftarrow (c+1)$ mod LENGTH($\mathcal{O}$)
8:     until $c \neq e$
9:     return $S$
10: end procedure
```

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

next tier is only applied when the solution is a local optimum for the previous tiers. Moreover, as in the standard VND setting, if the solution improved after a complete tier execution, the search is restarted from the initial tier (see Algorithm 3).

Algorithm 3 HRVND application

```
1: procedure HRVND($S, \boldsymbol{T}, \mathcal{R}$)
2:     $e \leftarrow 0$
3:     repeat
4:         $S' \leftarrow$ TIERAPPLICATION($S, \boldsymbol{T}_e, \mathcal{R}$)
5:         if COST($S'$) $<$ COST($S$) $\wedge e > 0$ then
6:             $S \leftarrow S', e \leftarrow 0$
7:         else
8:             $e \leftarrow e + 1$
9:         end if
10:    until $e <$ LENGTH($\boldsymbol{T}$)
11:    return $S$
12: end procedure
```

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

Move Generators and Granular Neighborhoods

A *move generator* $(i, j) \in E$ is an arc that, as the name suggests, is used to generate and identify a *unique* move in a local search context. In Toth and Vigo (2003), a set T of move generators is used to define a restricted local search neighborhood, also known as a *granular neighborhood* (GN), containing a subset of the possible moves associated with a local search operator.

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

In our approach, to speed up the local search execution, all neighborhoods induced by local search operators are implemented as GNs. We define the set T of move generators to contain all arcs connecting a vertex $i \in V$ to its $n_{gs} = 25$ nearest vertices. More precisely, the set $T = \cup_{i \in V}\{(i, j), (j, i) \in E : j \in \mathcal{N}_i^{n_{gs}}(V \setminus \{i\})\}$. Note that move generators are described by arcs instead of edges. In fact, a GN is said to be asymmetric if the move induced by (i, j) is different from that induced by (j, i), and symmetric otherwise.

All the local search operators we considered, with the exception of TWOPT and SPLIT, identify asymmetric GNs. The defined sparsification rule comes directly from the original GN definition by Toth and Vigo (2003), where the emphasis was on trying to replace long edges with short ones. However, in their work the sparsification is based on a cost rule selecting all edges having a cost lower than a given threshold, while here we adopt a nearest-neighbor rule that ensures a minimum number of move generators involving any vertex. A cost rule may in fact not be well suited when the standard deviation among arc costs is high, e.g., in clustered instances, and may cause several vertices not to have any move generator associated with them when the threshold is very low.

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

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

Finally, several papers (e.g., Schneider, Schwahn, and Vigo (2017)), support the inclusion of all the edges connecting the depot to customers in the set of move generators. However, in our experience, when scaling to large-scale instances, even the dynamic management of these move generators becomes very challenging, causing a significant increase in computing time without guaranteeing any improvements to the solution quality. More details are given in Section 2.4.4.

Static Move Descriptors

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

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

A local search operator whose neighborhood is designed according to the SMD principles requires four stages. First, an *initialization* stage, executed once at the beginning of the neighborhood exploration, computes the δ-tag for the available SMDs. Then, a sequence of *search*, *execution*, and *update* stages is performed until no more improving solutions are available in the neighborhood.

The search stage looks for a *feasible and improving* SMD - that is, an SMD associated with a move that generates a feasible and improving neighbor solution. Moreover, the SMD might also be required to meet some additional criteria (e.g., the SMD associated with the most improving feasible move might be sought). Once found, the move associated with the SMD that meets the criteria is executed during the execution stage. Because a local search application causes only a local change in a solution, most of the SMDs will still hold a correct δ-tag even after (part of) the solution is changed. An operator-specific list of SMDs requiring an update to their δ-tag can thus be considered during the update stage. Finally, the neighborhood exploration ends when the search stage is no longer able to identify a feasible and improving SMD.

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

Finally, to speed up the initialization stage we coupled it with the selective vertex caching strategy described in Section 2.2.2 that forces the local search to focus on recently changed areas of the solution.

Selective Vertex Caching

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

In our implementation, $\bar{V}_S$ consists of the set of vertices directly belonging to solution areas that recently underwent some changes. Without loss of generality, changes in a solution S, seen from a vertex perspective, can be subdivided into insertions and removals. For example, consider the relocation of vertex i from its original position to another one between vertex j and its predecessor π_j. The removal *directly* affects vertex i itself, its successor σ_i, and its predecessor π_i, while the subsequent insertion affects vertices i, j and π_j. Within this setting, after the move execution, we say that i, π_i, σ_i, j, and π_j are *cached* for S: i.e., they are inserted into $\bar{V}_S$.

This strategy allows us to easily keep track of solution areas that were recently changed. However, not all changes have the same importance; more recent ones are more likely to be relevant to a forthcoming decision. This aspect is captured by imposing a limit, for a solution S, on the maximum number of vertices that can be cached at the same time equal to a constant value C, by imposing $|\bar{V}_S| \leq C$ and adopting a *least recently used* strategy to maintain $\bar{V}_S$.

SVC to Restrict Local Search Execution. Vertices can be selectively cached in order to identify a kernel of relevant vertices to be used in local search procedures. As mentioned at the end of Section 2.2.2, we used this method as a heuristic acceleration and filtering technique for the initialization stage of the SMDs, which as shown in the analysis of Section 2.4.5, may also have

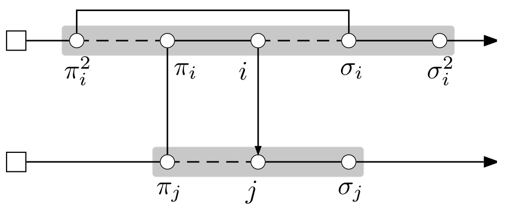


Figure 2.1: A 2OEX application induced by move generator (i, j) relocating path $(\pi_i - i)$ between π_j and j. SMDs involving vertices in the gray area require an update to their δ-tag after the move execution.

a significant influence on subsequent SMD stages and, ultimately, on the overall local search execution.

During the optimization of a solution S, the SMD initialization stage considers, for the heap insertion, the restricted subset of move generators $(i, j) \in \bar{T}^\gamma(S) \subseteq T^\gamma$, such that at least one of the endpoints i or j belongs to the subset of cached vertices $\bar{V}_S$. More precisely, $\bar{T}^\gamma(S) = \cup_{i \in \bar{V}_S}\{(i,j),(j,i) \in E : j \in \mathcal{N}_i^{k_i}(V \setminus \{i\})\}$ with $k_i = \lfloor \gamma_i \cdot n_{gs} \rceil$.

During subsequent SMD update stages, additional move generators might, however, be added to the heap due to the search incrementally extending to vertices not belonging to the selective cache and whose SMDs require an update although they have not been directly involved in a change of the solution.

To better understand this, consider the scenario shown in Figure 2.1 in which, during a 2OEX neighborhood exploration, a move induced by move generator (i, j) is executed, causing the relocation of path $(\pi_i - i)$ between π_j and j. Once the move is executed, vertices $\pi_i^2, \pi_i, i, \sigma_i, \pi_j$ and j will be marked as cached. However, as shown by the gray overlay, two additional vertices, namely σ_i^2 and σ_j are also (indirectly) affected by the move execution. In particular, active move generators involving σ_i^2, i.e. $\{(\sigma_i^2, j) : j \in V\} \cap T^\gamma$, require an update because the predecessor of σ_i changes from i to π_i^2. Similar reasoning applies to some move generators involving σ_j. More details about update lists associated with different local search operators are given in Section B.3 of the Appendix. Note that σ_i^2 and σ_j do not belong to the cache, but their move generators will be updated and, if improving, inserted into the heap and considered during subsequent SMD search stages.

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

A possible scenario in any of the improvement procedures, analyzed from the perspective of the number of distinct vertices either cached or (directly and indirectly) reached by the local search execution, is depicted in Figure 2.2. Improvement procedures, at the beginning of each iteration, work with solutions S having no cached vertices, i.e., $\bar{V}_S = \emptyset$. As mentioned in Section 2.2.2, both procedures make use of a shaking performed in a ruin (R^-) and recreate (R^+) fashion. The vertices involved in the disruptive effects of the shaking applied to solution S are added to the set $\bar{V}_S$. This set is the kernel of vertices used to identify the area where the optimization of the subsequent local search execution is focused. In particular, each SMD initialization stage (I) considers the current kernel of cached vertices. Then, a sequence of SMD search (S) and execution and update (X) stages might cause the search to reach far more vertices than those cached (dashed line), potentially covering all vertices. However, as discussed in

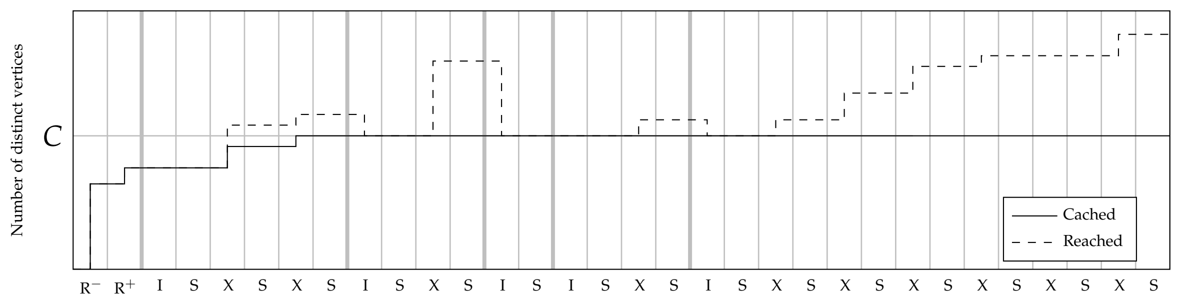


Figure 2.2: Evolution of an improvement procedure iteration in terms of number of distinct vertices simultaneously cached and considered during a neighborhood exploration (Reached). The number of vertices is analyzed after the ruin (R^-), recreate (R^+), and local search operator applications. Each local search operator application is partitioned into an SMD initialization (I) and a sequence of search (S) and execute and update (X) stages.

Section 2.4.5, the maximum size C of this kernel indirectly affects the exploration power as well as the computing time of the local search.

The overall result is an implicit dynamic instance decomposition, induced by a very focused and localized neighborhood exploration strategy which mainly considers the areas of the solution that are more likely to require further optimization because they were more recently changed.

2.2.3 Improvement

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

Route Minimization

The CVRP typically does not impose any limit on the number of routes that solutions may have. However, there is often a positive correlation between the number of routes in a solution and its cost. Moreover, many simple construction algorithms, such as the one we use, typically produce solutions using far more routes than those found in good-quality solutions. We thus include an optional route minimization procedure that may be executed right after the initial solution construction.

This procedure is applied to a solution S built by the initial construction phase whenever the number $|S|$ of its routes is found to be greater than an ideal estimated number of routes k. The value k is computed by heuristically solving a bin-packing problem with an item of weight q_i for each customer $i \in V_c$ and bins of capacity Q through a simple greedy first-fit algorithm (see, e.g., Chapter 8 of Martello and Toth (1990)). The route minimization procedure, whose pseudocode is shown in Algorithm 4, starts by setting the best found solution S^* to be equal to the initial solution S generated by the construction phase. During each iteration a pair (r_i, r_j) of routes

Algorithm 4 Route minimization procedure

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

belonging to S is selected and their customers removed from the solution and placed into a list of unrouted customers L (Lines 5-7). The first route r_i is chosen as the route containing a random customer seed i. The second route r_j is identified by considering customer neighbors $j \in \mathcal{N}_i(V_c)$ in increasing c_{ij} cost until a customer j belonging to a route $r_j \neq r_i$ is found. Customers in L are, with equal probability, either randomly shuffled or sorted according to their demand, in decreasing order. Then, for each unrouted customer $i \in L$, a position in the current set of existing routes is searched, such that with the insertion of i the target route remains feasible and the insertion cost is minimized (Lines 10-21). When such a position cannot be found (i.e., when inserting i violates the capacity constraints of all existing routes), and consequently a new route should be created to accommodate customer i, an action is selected according to the current number of routes $|S|$. If $|S|$ is lower than the estimate k, a new single-customer route with i is created. Otherwise, the single-customer route serving i is created only when a random number drawn from a uniform real distribution in the interval $[0, 1]$ is greater than a threshold $\mathcal{P}$, set to $\mathcal{P} = 1$ at the beginning of the route minimization procedure. When the random number is not greater than $\mathcal{P}$, i is inserted into an additional list $\bar{L}$.

Once all customers in L have been considered, the list L is set to $L = \bar{L}$ and the threshold $\mathcal{P}$ is lowered according to an exponential schedule $\mathcal{P} = z \cdot \mathcal{P}$ with $z = (\mathcal{P}_f/\mathcal{P}_0)^{(1/\Delta_{RM})}$, $\mathcal{P}_f = 0.01$, and $\mathcal{P}_0 = 1$. Where $\mathcal{P}_f$ and $\mathcal{P}_0$ are the final and initial probabilities of not creating an additional single-customer route, respectively.

Next, a restricted HRVND, consisting of the first tier only but using all the available move generators, i.e., $\gamma_i = 1, i \in V$, is applied to the (possibly partial) current solution. We restrict the HRVND to the first tier because we noticed it was sufficient to obtain good-quality solutions and resulted in a considerable computing time saving. Moreover, we set $\gamma_i = 1, i \in V$ to avoid a complex management of move generators in this secondary improvement phase which, as shown in the parameters Table 2.1, is executed for a small number $\Delta_{RM} = 1000$ of iterations.

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

Before proceeding to the next iteration, a partial or feasible solution S having a cost higher than the current best solution S^* is reset to the latter; i.e., $S = S^*$. Finally, solution S^* is returned after Δ_{RM} iterations if the early stopping condition is not met throughout the route minimization execution.

Core Optimization

The core optimization procedure, whose pseudocode is shown in Algorithm 5, is iterative and randomized. By making use of an adaptive shaking strategy and the local search engine, it implements a powerful localized solution improvement. First, the best solution S^* is set equal to solution S, generated by the construction phase and possibly improved by the route minimization procedure.

A shaking application, whose pseudocode is shown in Algorithm 6, performs a ruin step, removing a number of customers in the customer sub-graph by means of a random walk. Then, a simple greedy recreate step defines the new position for the previously removed customers. More precisely, the ruin step, starting from a randomly selected seed customer $i \in V_c$, identifies a random walk of length ω_i in the sub-graph $G' = (V'_c, E'_c)$ where $V'_c = V_c$ and

Algorithm 5 Core optimization procedure

```
1: procedure CoreOpt($S, \mathcal{R}$)
2:     $\boldsymbol{\omega} \leftarrow (\omega_0, \omega_1, \dots, \omega_{|V_c|}), \omega_i \leftarrow \omega_{base}\ \forall i \in V$
3:     $\boldsymbol{\gamma} \leftarrow (\gamma_0, \gamma_1, \dots, \gamma_{|V_c|}), \gamma_i \leftarrow \gamma_{base}\ \forall i \in V$
4:     $S^* \leftarrow S, \mathcal{T} \leftarrow \mathcal{T}_0$
5:     for $n \leftarrow 1$ to $\Delta_{CO}$ do
6:         $\bar{V}_S \leftarrow \emptyset$
7:         $\hat{S}, i' \leftarrow$ Shake($S, \mathcal{R}, \boldsymbol{\omega}$), $\mathcal{S} \leftarrow \bar{V}_{\hat{S}} \setminus \{0\}$
8:         $S' \leftarrow$ HRVND($\hat{S}, \mathcal{R}$), $\mathcal{L} \leftarrow \bar{V}_{S'}$
9:         if Cost($S'$) < Cost($S^*$) then
10:            $S^* \leftarrow S'$
11:            ResetSparsificationFactors($\boldsymbol{\gamma}, \mathcal{L}$)
12:        else
13:            UpdateSparsificationFactors($\boldsymbol{\gamma}, \mathcal{L}$)
14:        end if
15:        UpdateShakingParameters($\boldsymbol{\omega}, S', S, i', \mathcal{S}, \mathcal{R}$)
16:        if AcceptNeighbor($S, S', \mathcal{T}$) then $S \leftarrow S'$
17:        $\mathcal{T} \leftarrow c \cdot \mathcal{T}$
18:    end for
19:    return $S^*$
20: end procedure
```

$E'_c = \{(i,j) : i,j \in V'_c\}$ is the set of arcs connecting customers. When a customer i is visited, it is removed from the solution and the sub-graph G' is updated accordingly by setting $V'_c = V'_c \setminus \{i\}$ and $E'_c = E'_c \setminus \{(i,j),(j,i) : j \in V'_c\}$. A partial walk ending at customer i is extended by moving either forward or backward within the same route r_i, or by jumping to a neighbor route, which can be any route or a not yet visited one (Lines 7-20). First, whether to move along the same route or jump to another is decided; then the possible options associated with that choice are considered. At every step, the choices are considered with equal probability. When a jump to a neighbor route is selected to extend a walk ending at $i \in V'_c$, customers $j \in \mathcal{N}_i(V'_c)$ are examined in increasing c_{ij} cost until a route r_j satisfying the appropriate requirements, i.e., either any route (including $r_j = r_i$) or a not yet visited one, is found. The r_j, identified by scanning the routes to which neighbor customers j of i belong, is considered to be a neighbor route of r_i. In the unlikely case that such a route cannot be found, the ruin procedure is prematurely aborted (Line 18). Note that a jump to a neighbor route is always selected when the current route r_i contains customer i only (Line 7).

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

The ruin intensity is controlled by a meta-procedure (described in Paragraph 2.2.3) that iteratively adapts the random walk length ω_i from a seed customer $i \in V_c$ to identify a disruptive action that best suits the instance and solution under examination.

The HRVND is then applied to the shaken solution S to identify a local optimum S'. Whether to accept S' as the next point in the search trajectory is determined by a simulated annealing acceptance strategy; see Kirkpatrick, Gelatt, and Vecchi (1983). In particular, S' is accepted if $c(S') < c(S) + \mathcal{T} \cdot \ln U(0,1)$. The value of $\mathcal{T}$ is initially set to $\mathcal{T} = \mathcal{T}_0$ and lowered at the end of each core optimization iteration by performing $\mathcal{T} = c \cdot \mathcal{T}$ with $c = (\mathcal{T}_f / \mathcal{T}_0)^{(1/\Delta_{CO})}$; Δ_{CO} is the number of core optimization iterations.

The core optimization makes full use of vertex-wise dynamic move generators. In particular, at the beginning of the procedure, sparsification parameters γ_i are set to a base value γ_{base}. Whenever $(\delta \cdot \Delta_{CO} \cdot \text{AVERAGE}(|\bar{V}_S|))/|V|$ nonimproving iterations involving a vertex i are performed,

Algorithm 6 Adaptive shaking procedure

```
1: procedure SHAKE($S, \mathcal{R}, \omega$)
2:     $i' \leftarrow$ RANDOMCUSTOMER($\mathcal{R}$)
3:     $i \leftarrow i', e \leftarrow 0, C = [\,], R = \{\}$
4:     $\mathcal{B} \leftarrow$ RANDOMBOOLEAN($\mathcal{R}$)
5:     repeat
6:         $C \leftarrow [C, i], R \leftarrow R \cup \{r_i\}$
7:         if $|S| > 1 \land \mathcal{B}(\,)$ then
8:             if $\mathcal{B}(\,)$ then
9:                 $j \leftarrow$ NEXTCUSTOMERINROUTE($i$)
10:            else
11:                $j \leftarrow$ PREVCUSTOMERINROUTE($i$)
12:            end if
13:        else
14:            if $\mathcal{B}(\,)$ then
15:                $j \leftarrow$ NEARESTSERVEDCUSTOMER($S, i$)
16:            else
17:                $j \leftarrow$ NEARESTSERVEDCUSTOMER($S, i$) such that $r_j \notin R$
18:                if $j = none$ then $e \leftarrow \infty$
19:            end if
20:        end if
21:        $S \leftarrow$ REMOVECUSTOMER($S, i$)
22:        $i \leftarrow j$
23:    until $e < \omega_{i'}$
24:    $C \leftarrow$ DEFINEORDER($C, \mathcal{R}$)
25:    for $i \in C$ do
26:        $p \leftarrow$ BESTINSERTIONPOSITIONINEXISTINGROUTES($i, S$)
27:        if $p \neq none$ then
28:            $S \leftarrow$ INSERT($i, p, S$)
29:        else
30:            $S \leftarrow$ BUILDSINGLECUSTOMERROUTE($i, S$)
31:        end if
32:    end for
33:    return $S, i'$
34: end procedure
```

the value is updated as $\gamma_i = \min\{\gamma_i \cdot \lambda, 1\}$ where $\delta \in [0,1]$ is a reduction factor, $\textsc{average}(|\bar{V}_S|)$ denotes the average number of vertices cached after previous local search executions, and λ is an increment factor. However, the value of γ_i is reset to γ_{base} whenever a solution S improving S^* is found during the execution of a local search involving i (i.e., $i \in \bar{V}_S$ after the HRVND application). This update rule is a possible vertex-wise implementation of the standard way of handling dynamic move generators described in Schneider, Schwahn, and Vigo (2017) in which the total number of core optimization iterations is partitioned over restricted working areas identified by cached vertices. Note that, when the cache size C is smaller than the total number of instance vertices, i.e., $C < |V|$, some vertex i may not be considered for the γ_i update even if it is involved in a change during the HRVND execution because, due to the limit imposed by C, it is no longer cached after the optimization. However, as shown in the analysis in Section 2.4.5, an accurate selection of C, which might heuristically filter out some vertices from the update, does not prevent good solutions from being found much faster than in the scenario in which $C = |V|$ and the SVC is completely disabled.

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

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

From the above observations, we can derive a simple update rule for the shaking parameters ω_i that is executed at every core optimization iteration (Line 15 of Algorithm 5). Denoting by $\tilde{\omega} = \omega_{i'}$ the shaking parameter value associated with the current seed customer i'

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

where $\mathcal{S} = \bar{V}_{\hat{S}} \setminus \{0\}$ is the set of vertices cached in the shaken solution $\hat{S}$ right after the shaking execution (excluding the depot, which is never considered in the ruin execution); see Line 7 of Algorithm 5. Shaking parameters $\omega_i, i \in \mathcal{S}$ are moved towards the new value for $\tilde{\omega}$, without exceeding it.

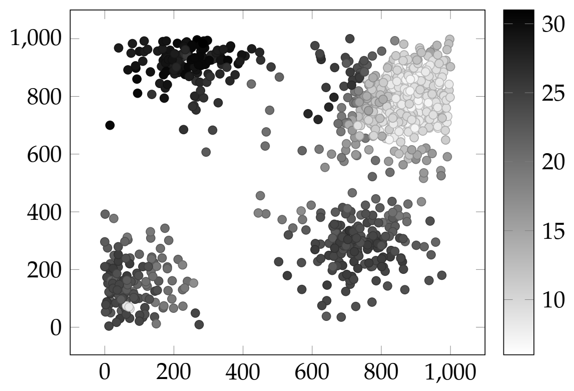


Figure 2.3: Shaking parameters values at the end of the core optimization procedure for instance X-n979-k58 of the $\mathbb{X}$ dataset. Each circle represents a customer i in its x_i and y_i coordinates and the color denotes the shaking parameter ω_i value.

This limit prevents situations in which a single vertex j surrounded by a set of vertices $\tilde{V}$, all having a very small (respectively, large) shaking parameter value has its ω_j indirectly incremented (respectively, decremented) to very large (respectively, small) values due to updates involving some $i \in \tilde{V}$. Furthermore, update rule (iii) describes the scenario in which S' is improving with respect to S or the shaking was of the appropriate strength; i.e, $\Omega_{LB} \leq \textsc{Cost}(S') - \textsc{Cost}(S) \leq \Omega_{UB}$.

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

As an example, consider Figure 2.3, showing shaking parameter values $\omega_i, i \in V$ for instance X-n979-k58 of the $\mathbb{X}$ benchmark after the core optimization procedure.

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

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

2.3 Computational Results

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

2.3.1 Implementation and Experimental Environment

The algorithm was implemented in C++ and compiled using g++ 8.3.0. The experiments were performed on a 64-bit desktop computer with an Intel Xeon CPU E3-1245 v5 central processing unit (CPU), running at 3.5 GHz and with 16 GB of RAM on a GNU/Linux Ubuntu 18.04 operating system. The algorithm source code, together with a library of reusable components, can be downloaded from `https://acco93.github.io/filo/`; detailed instructions are given to accurately reproduce our results. In all the computational testing, we considered a standard version of FILO and a longer version, called FILO (long), which performs ten times more core optimization iterations than the standard version. Because of the randomized nature of the algorithm, for every experiment, we executed a symbolic number of fifty runs for each instance, defining the seed of the pseudorandom engine (the Mersenne twister of Matsumoto and Nishimura (1998)) as equal to the run counter minus one. Moreover, to mitigate the impact of small time-variations due to the overhead of the operating system, we used a clock function that reports running times with the minimum recordable run time set to one second. To better compare our results with other algorithms executed on different hardware configurations, for which no source code was available, we used the single-thread rating defined by PassMark ®Software (2020). At the time of writing, a score of 2285 was assigned to our CPU. Competing methods' CPU times are scaled to match our CPU score; their normalized time is identified by $\hat{t} = t \cdot (P_A/P_B)$, where P_A is the competing method's CPU single-thread rating, P_B is our CPU rating, and t is the raw computing time. All times refer to an average run and are reported in minutes. For randomized algorithms we report, when available, the best (Best), average (Avg) and worst (Worst) gaps of the solution found by the algorithm, with respect to the best known solution value (BKS). Gaps are computed as $100 \cdot (\textsc{Cost}(S) - \text{BKS})/\text{BKS}$, where S is the final solution. For deterministic algorithms, we report the gap (Gap) of the solution found by a single run.

2.3.2 Parameter Tuning

Crafting algorithm FILO and tuning its parameters followed an iterative process whose key decisions are detailed in Section 6.

Parameters are summarized in Table 2.1; their tuning, whose hidden interactions and interconnected effects might be very challenging to analyze, followed a straightforward sequential strategy aimed at keeping the tuning effort low but still able to identify good performing values for each parameter considered individually.

Initial solution definition – Described in Sections 2.2.1 and 2.2.3 – Analyzed in Section 2.4.1	
$n_{cw} = 100$	Number of neighbors considered in the savings computation.
$\Delta_{RM} = 10^3$	Maximum number of route minimization iterations.
Granular neighborhood – Described in Section 2.2.2 – Analyzed in Section 2.4.4	
$n_{gs} = 25$	Number of neighbors considered by the sparsification rule.
$\gamma_{base} = 0.25$	Base sparsification factor.
$\delta = 0.5$	Reduction factor used in the definition of the fraction of non-improving iterations performed before increasing a sparsification parameter.
$\lambda = 2$	Sparsification increment factor.
Core optimization – Described in Sections 2.2.2 and 2.2.3 – Analyzed in Sections 2.4.2, 2.4.5 and 2.4.3	
$\Delta_{CO} = 10^5, 10^6$	Number of core optimization iterations for a short and a long run.
$\mathcal{T}_0, \mathcal{T}_f$	Initial and final simulated annealing temperature.
$C = 50$	Maximum number of cached vertices.
$\omega_{base} = \lceil \ln \lvert V \rvert \rceil$	Initial shaking intensity.
$n_{EC} = 25$	Maximum number of sequences explored by EJCH for each move generator.
$I_{LB} = 0.375, I_{UB} = 0.85$	Shaking factors.

Table 2.1: Parameters of Algorithm FILO.

First, reasonable values were identified using the authors' judgment and experience, along with a trial-and-error approach. Then we evaluated the algorithm's behavior while changing the value of one parameter at a time (keeping the others fixed). A new value was kept when it allowed an improvement in quality without increasing the computing time. This process was iterated several times until satisfactory results were obtained. In fact, we noticed that the iterated sequential tuning of individual parameters, without a prefixed order, was enough to reach good local optima without exploring all possible combinations of values.

The parameter tuning, as well as the algorithm design, was mainly performed by considering the largest $\mathbb{X}$ instances - in particular, those with more than five hundred vertices. The resulting tuned algorithm was then used for all our computational testing.

We briefly summarize, in the following, the key choices made during the parameter tuning procedure, referring to Section 6 for more details. The number of customers n_{cw} for which the savings are computed in the construction phase and the maximum number of route minimization iterations Δ_{RM} are both low impact parameters. Once they are set to reasonable values, small variations do not significantly change the outcome of the procedures in which they are employed. We set them approximately to one of the smallest values able to provide results of a quality comparable to larger values, which may, however, have required a longer computing time. Conversely, the value of n_{gs}, γ_{base}, δ and λ heavily affect the algorithm performance.

The number of neighbors n_{gs}, the base sparsification factor γ_{base} and the reduction factor δ were set so as to obtain the best trade-off between computing time and solution quality; see Section 2.4.4. In particular, we noticed that a milder sparsification (obtained, for example, by increasing n_{gs}, γ_{base} or both; or by decreasing δ) may sometimes provide slightly better solutions, but with an unacceptable increment in the computing time. This is particularly noticeable in instances for which very good-quality (or near-optimal) solutions are found early in the search, and for which the sparsification is just steadily decreased by increasing the $\gamma_i, i \in V$ and never reset; see, e.g., the computing time for instance X-n219-k73 of the $\mathbb{X}$ dataset in Table 2.9 (Section C of the Appendix). We set the the value for λ as in the original proposal of Toth and Vigo (2003), leaving the other granular parameters depending on it. On average, we found a very aggressive sparsification associated with an large number of optimization iterations to be preferable to a very accurate local search execution performed with fewer iterations.

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

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

The size of the selective cache C was chosen to be the value that identified a good trade-off between computing time and solution quality; see Section 2.4.5.

As mentioned in Section 2.2.2, both small and large-scale instances are not significantly affected by the choice of the initial shaking intensity ω_{base}; in fact, comparable quality and computing time are obtained by setting $\omega_{base} = 1$. However, when considering very large-scale instances, the difference in the final outcome is much more noticeable. We found that setting ω_{base} as a logarithmic function of the number of vertices provides a reasonable starting point. The shaking procedure then actually performs some fruitful iterations in large instances before reaching better-performing values for the shaking parameters $\omega_i, i \in V_c$.

We set the number n_{EC} of EJCH sequences explored from every move generator, to the minimum value that could provide reasonably good improvements with a much more limited computing time variability compared to larger values; see Section 2.4.2.

Finally, the shaking factors I_{LB} and I_{UB} can have a major impact on the overall algorithm execution. In fact, assigning them large values allows the guiding meta-strategy to increment the ω_i values, inducing a stronger shaking effect that will eventually require a longer local search re-optimization. On the other hand, values that are too low do not disrupt the current solution sufficiently to allow the re-optimization the possibility of performing some improvements. The identified values, coming from the limited random search analysis (see Bergstra and Bengio (2012)) described in Section 2.4.3, define an associated shaking which is neither too disruptive nor too gentle, resulting once again in a compromise between computing time and final solution quality.

2.3.3 Testing on $\mathbb{X}$ Instances

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

The performance of FILO is compared with current state-of-the-art algorithms on this dataset.

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

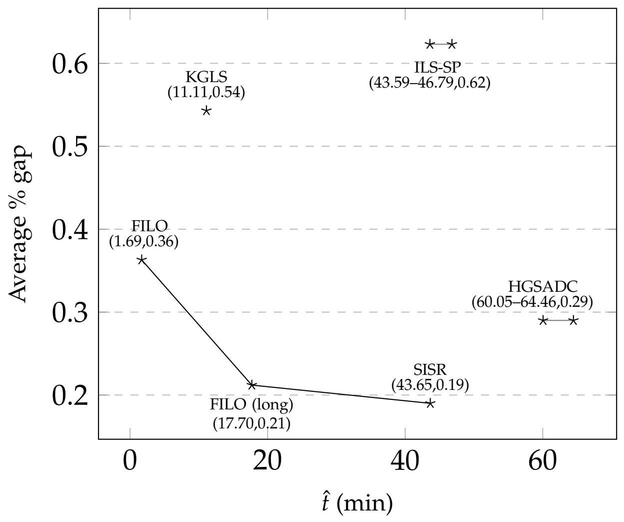


Figure 2.4: State-of-the-art CVRP algorithms performance comparison over the $\mathbb{X}$ instances of Uchoa et al. (2017). A tuple describing the computing time and the average percentage gap is reported below each algorithm name. For ILS-SP and HGSADC we report a range of times since the exact CPU model is not specified.

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

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

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

Figure 2.4 provides a graphic comparison of the algorithms' performances in terms of efficacy and efficiency by reporting the average behavior over 50 runs for ILS-SP, HGSADC, SISR, and FILO, and over a single run for KGLS, since it is a deterministic algorithm. The best known solution values (BKS) used to compute gaps are taken, at the time of writing, from CVRPLIB (2020). FILO compares favorably with the best existing algorithms, achieving an excellent compromise between solution quality and computing time. In fact, FILO finds average solutions that are significantly better than those of ILS-SP and KGLS, and similar to those found by HGSADC. However, SISR outperforms FILO. FILO (long), on the other hand, finds average solutions that are significantly better than those of ILS-SP, HGSADC, and KGLS and similar to those found by SISR. See Section C of the Appendix for full details.

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

Table 2.2: Aggregate computations on $\mathbb{X}$ instances.
[1]obtained by averaging normalized times associated with the fastest and the slowest compatible CPUs.

As mentioned in Section 2.3.1, the average computing time for a single run $\hat{t}$ has been roughly normalized to match our CPU score by using the single-thread rating of PassMark ®Software (2020), which assigns a score in the range of 1389 – 1491 to the compatible Intel Xeon CPUs used by ILS-SP and HGSADC (for which the precise model is not specified in Uchoa et al. (2017)), a score of 2052 to the AMD Ryzen 3 1300X CPU used by KGLS, and a score of 1662 to the Intel Xeon E5-2650 v2 CPU used by SISR.

Table 2.2 provides aggregate computations, grouped by instance size. The table highlights the scalability properties of FILO, by showing that the computing time for the largest instances is very similar to that obtained for smaller ones, yet the solution quality remains comparable to that achieved by other state-of-the-art algorithms. In fact, the computing time of FILO is more related to the number of core optimization iterations rather than to the instance size. Moreover, when the computing times of FILO and FILO (long), which differ by the number of core optimization iterations by a factor of ten, are compared, the increase in computing time is proportional to the increase in the number of iterations.

More experiments can be found in Section 2.4.6; for the largest instances, we studied the effects of further increasing the number of core optimization iterations which resulted in lowering the average gap to 0.19%.

2.3.4 Testing on Very Large-Scale Instances

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

The $\mathbb{B}$ instances proposed by Arnold, Gendreau, and Sörensen (2019) are a set of ten very large-scale instances containing up to thirty thousand customers and reflecting real-world parcel distribution problems in Belgium. They include a first scenario, in which the depot is located centrally with respect to the customers and relatively short routes are performed, and a second one in which the depot is eccentric with respect to the service zone, thus much longer routes are required to visit the customers. We mainly compared FILO with KGLS$^{\text{XXL}}$ proposed in Arnold, Gendreau, and Sörensen (2019), an adaptation for very large-scale instances of the KGLS algorithm introduced in Section 2.3.3. Like KGLS, KGLS$^{\text{XXL}}$ also defines a time-based termination condition of either three or twelve minutes every 1000 customers for the short or the long version, respectively. KGLS$^{\text{XXL}}$ was executed on the same hardware configuration described for the $\mathbb{X}$ dataset. Moreover, for the sake of completeness, we include results obtained by the LKH-3 algorithm proposed by Helsgaun (2017) in very long computing sessions (up to several days). The best known solution values (BKS) were taken, at the time of writing, from CVRPLIB (2020) and used to compute gaps. We note that for many of those BKS, no citable publication is available; the results are typically the outcome of very long runs. In other cases, as reported us by the authors (see Cavaliere, Bendotti, and Fischetti (2020b) and Uchoa (2020)), the methods

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

Table 2.3: Computations on $\mathbb{B}$ instances.
New best solutions: (L1, 193052);(L2, 111661);(A2, 292303);(G2, 258700);(B2, 347092);(F2, 4400188).

were initialized with previously-known best solutions. Therefore, their achievements cannot be compared with monolithic approaches such as the one we propose.

As can be seen from Table 2.3, FILO is able to successfully find very good quality solutions in a shorter computing time than KGLS^{XXL}. In fact, the average gap of FILO is almost half that of KGLS^{XXL} (long) in just about five minutes of computing time. Furthermore, FILO (long) is able to find several new BKS in less than three hours of computing time, and the computing times remain consistent across instances with different structures. We again note the scalability of FILO by observing that, to solve an instance with ten times more customers, the computing time only increases approximately fivefold.

The $\mathbb{K}$ dataset proposed by Kytöjoki et al. (2007) contains eight very large-scale instances with up to twelve thousand customers. The first four instances (W, E, S, and M) are derived from real-life waste collection problems in Finland, while the remaining instances contain customers randomly and uniformly distributed. Again, we mainly compared FILO with KGLS^{XXL} proposed in Arnold, Gendreau, and Sörensen (2019) but we also include results of the GVNS algorithm introduced in Kytöjoki et al. (2007) (the first method used to solve the $\mathbb{K}$ instances). GVNS was run on an AMD Athlon 64 3000+ having a single thread score of 554 and KGLS^{XXL}

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

Table 2.4: Computations on $\mathbb{K}$ instances.
*taken from Arnold, Gendreau, and Sörensen (2019).
New best solutions: (W, 4101686.00);(E, 4324802.50);(S, 2996254.00);(M, 2964867.25);(R3, 180597.91);(R6, 344407.00);(R9, 507787.66);(R12, 668435.00).

was run on the same hardware configuration as for the $\mathbb{B}$ dataset.

As can be seen from Table 2.4, FILO is able to successfully find very good-quality solutions in a relatively short computing time compared to KGLS$^{\text{XXL}}$. Importantly, both FILO and FILO (long) find new best solutions for all instances. The computing time associated with random instances follows the same trend seen for the $\mathbb{B}$ instances. On the other hand, instances derived from real-life problems require a much longer computing time. By analyzing the structure of the final solutions obtained by FILO (long), we note that they are composed of few very long routes with several hundred customers. Moreover, the shaking intensity at the end of a run $\omega = \sum_{i \in V_c} \omega_i / |V_c|$, averaged over the W, E S and M instances, has a value four times larger than that associated with the $\mathbb{B}$ dataset (91.37 ± 28.86 and 22.42 ± 4.40, respectively). The reason for such a high average value may be related to the shaking procedure. In particular, given the very low number of routes for those instances (on average 15.00 ± 1.83), the shaking procedure might choose to jump to a not-yet-visited neighbor route that is not available. In such cases, the ruin is prematurely aborted, possibly causing a mismatch between the actual shaking intensity and the required one identified by $\omega_i, i \in V_c$ values. The mismatch may cause the average shaking intensity to increase to an abnormally large value. In many cases, the ruin activity is unaffected, because it will be prematurely aborted. However, if the early stop comes after several customers have already been removed, the average ruin activity will be stronger than required, causing a more time-consuming re-optimization.

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

As can be seen from Table 2.5, FILO successfully solved the $\mathbb{Z}$ dataset, finding new best solutions for all four instances in a very short computing time, which is comparable to that associated with instances of similar size of the $\mathbb{B}$ and $\mathbb{K}$ datasets.

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

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

Table 2.5: Computations on $\mathbb{Z}$ instances.
[1]max (normalized) time per run.
New best solutions: (ZK1, 13419.44);(ZK2, 3441.54);(ZK3, 1132.47);(ZK4, 1107.62).

2.4 Algorithmic Components Analysis

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

2.4.1 Initial Solution Definition

The construction phase depends on the parameter n_{cw} to identify, for each customer $i \in V_c$, the number of neighbors $j \in \mathcal{N}_i^{n_{cw}}(\{j \in V_c : i < j\})\}$ involved in the savings computation. Figure 2.5 (left) shows the variation of the solution quality (QUALITY GAP) and compactness (ROUTE GAP) when varying n_{cw}. We focused on the subset of instances, listed in Table 2.6, for which computing an initial solution of good quality is difficult (i.e., instances for which initial solutions have a large gap and use more routes than suggested by the heuristic estimate).

Our findings validate what was already proposed in Arnold, Gendreau, and Sörensen (2019). A value of n_{cw} around 100 provides initial solutions of quality comparable to that of larger values, but in slightly shorter computing times (the differences are, however, in the order of a few tens of milliseconds for the largest n_{cw} values we considered).

As in Section 2.3.1, the QUALITY GAP is defined as $100 \cdot (\textsc{Cost}(S) - \text{BKS})/\text{BKS}$, where S is the solution resulting from the procedure and, similarly, ROUTE GAP is defined as $100 \cdot (|S| - k)/k$, where k is the heuristically found ideal estimated number of routes described in Section 2.2.3.

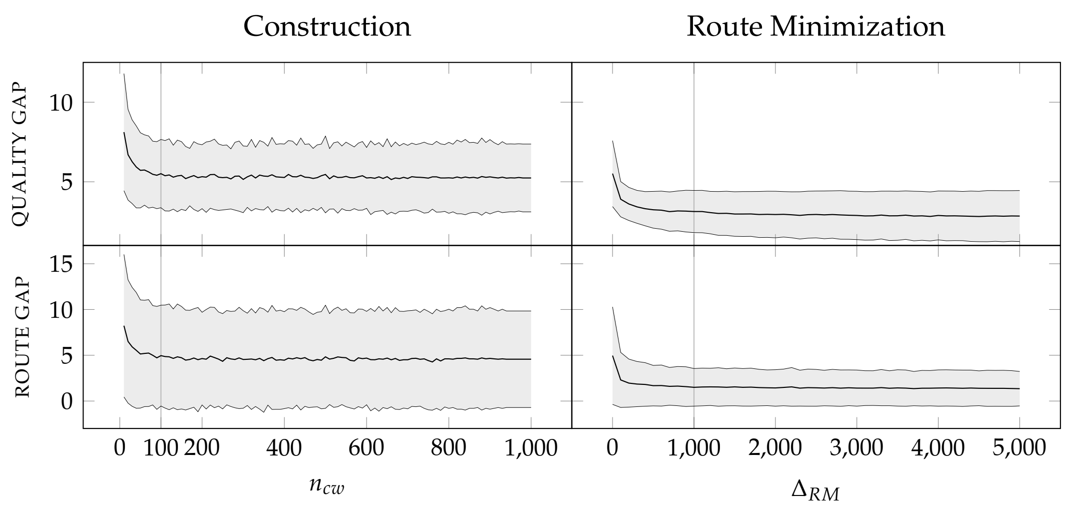


Figure 2.5: Tuning of the n_{cw} and Δ_{RM} parameters based on instances listed in Table 2.6 for which initial solutions have a large gap and use more route than suggested by the heuristic estimate. For each diagram, the middle line represents the average and the grayed area identifies the standard deviation. Selected values for n_{cw} and Δ_{RM} are marked with a vertical line.

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

Table 2.6: Computations with and without route minimization (RM) procedure.
[1]for which the route minimization procedure is executed.

Not surprisingly, using larger n_{cw} values is not sufficient to increase solution compactness. Indeed, the route minimization procedure tackles this strategic aspect of the CVRP, which is more concerned with the assignment of customers rather than with their routing.

Note, however, that, contrarily to most existing route minimization procedures, the proposed one is still quality-oriented. In fact, a solution with a better objective function is always preferred over a solution with a lower number of routes, but the procedure structure is more specifically aimed at reducing the number of routes while also often obtaining the desirable effect of improving the objective function.

Figure 2.5 (right) shows the results of this procedure when it is applied, for different numbers of iterations Δ_{RM}, to a solution S built by the construction phase with $n_{cw} = 100$. The diagrams highlight the procedure's effectiveness, both in improving low-quality initial solutions and in quickly compacting them by (often significantly) reducing the number of routes. As a result, we selected $n_{cw} = 100$ and $\Delta_{RM} = 1000$. The average computing time for largest values of Δ_{RM} is approximately ten seconds.

In addition, Table 2.6 compares the final algorithm outcome when the route minimization procedure is disabled, i.e., $\Delta_{RM} = 0$. Despite not always being crucial for the final solution quality because of the complex interactions among all the algorithm's components, the route minimization procedure provides substantial improvement for those instances containing several customers with small demand and a few customers with relatively large demand, such as X-n670-k130 and X-n936-k151. We can conclude that, in average, the route minimization positively affects the final algorithm's outcome without any significant impact on the computing time.

2.4.2 Local Search

We analyzed the local search operators described in Section 2.2.2 in the context of the core optimization procedure, where all features we propose are fully employed. The effect of a local search operator application is tightly linked to the state of the algorithm in that specific instant. In our approach, randomization plays a major role in selecting the area that, once disrupted by the shaking procedure, is re-optimized, and the evolution of the algorithm affects shaking intensity and sparsification factors. We thus believe that the evaluation of a local search operator should not occur "in a vacuum", but needs to be performed within the algorithm execution. Therefore, we studied the effectiveness of individual operators by sampling the algorithm state throughout the core optimization phase. More specifically, a sample consists of a shaken solution S and its subset of cached vertices $\bar{V}_S$, the shaking vector $\boldsymbol{\omega}$, and the sparsification vector γ. Each sample, derived from the actual algorithm execution, is a relevant snapshot describing the algorithm evolution. Moreover, according to when the sample is taken, it could describe initial or final algorithm states associated with lower or higher quality solutions.

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

As expected, EJCH($\cdot$) are the most effective, yet time-consuming, operators. Their success ratio also suggest that in most shaken solutions, finding an improving ejection-chain is relatively easy. However, when this is not the case, the computing time can be very large especially for the EJCH with the highest n_{EC}.

We also noted that simpler operators, such as 10EX, 11EX, TAILS, and SPLIT have a large success ratio, meaning that they are more likely to be applied, and provide larger gap improvements compared to all other operators with quadratic cardinality. These differences may occur because their feasibility requirements are more easily met, and thus they are more frequently applied. On the other hand, all the remaining operators, despite having a lower success ratio, still do allow better final solutions than when they are disabled. Surprisingly, TWOPT is seldom useful, meaning that the shaking procedure typically generates routes that are almost two-optimal. However, we kept it because of its very short application time. When structuring the HRVND, we grouped all the operators with a comparable application time in the first tier (i.e., all the operators but EJCH($\cdot$)).

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

Finally, as we apply the operators of each tier in an RVND fashion, we may expect a lower success ratio and less gap improvement, as well as a shorter application time, when they are applied on solutions that are already local optima for a number of other operators of the same tier. We can now study how each operator contributes to the total improvement of the defined HRVND structure in more detail. Denoting with $\mathbb{O}$ the set of local search operators we employ, we stored the total gap improvement $D(O)$ achieved in a number of $I(O)$ successful applications for each operator $O \in \mathbb{O}$. Note that a single application consists of a full neighborhood

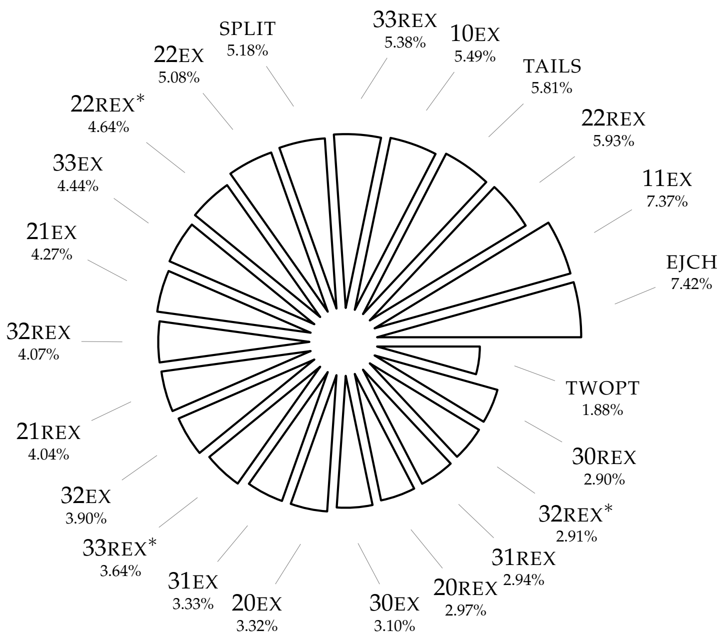


Figure 2.6: Percentage relative neighborhood improvement index for each local search operator in the engine. The index summarizes the contribution of each operator to the total improvement.

exploration. The ratio $R(O) = D(O)/I(O)$ thus identifies the expected improvement that a successful exploration of O would produce on an average solution. We can compute the percentage *Relative Neighborhood Improvement* index of O with respect to the set of the available operators $\mathbb{O}$ as $\text{RNI}(O, \mathbb{O}) = 100 \cdot R(O)/\sum_{O' \in \mathbb{O}} R(O')$, which is shown in Figure 2.6. As shown in the figure, all the operators positively contribute to the overall improvement process.

2.4.3 Shaking Guiding Strategy

The structure-aware and quality-oriented strategy employed by the core optimization procedure to guide the shaking intensity uses two factors to determine whether to increment, reduce, or randomly change the parameters $\omega_i, i \in V_c$ of customers involved in a shaking application. More precisely, I_{LB} determines when shaking parameters are increased, while I_{UB} defines when they are decreased. Together, they identify the range in which the guiding strategy actively operates. To determine reasonably effective values for I_{LB} and I_{UB}, we performed a limited random search, testing a hundred unique combinations for $I_{LB} \in [0.2, 0.4]$ discretized with steps of 0.025, and $I_{UB} \in [0.5, 1.5]$ discretized with steps of 0.05. A graphic representation for the different configurations and their performances is depicted in Figure 2.7.

The proposed ranges are the outcome of an iterative process in which larger ones were narrowed down to focus on combinations producing good-quality solutions in a short computing time. As shown in Figure 2.7, the values in the selected ranges have a minor impact on the solution quality and a slightly greater one on the computing time. In our implementation, we selected configuration 78 ($I_{LB} = 0.375$ and $I_{UB} = 0.85$). We noticed that slightly better solutions can be obtained when using moderately larger values particularly for I_{UB}. However, as shaking parameters reach larger values, we expect the associated re-optimization time to increase and selecting values that are too large may also result in poor quality final solutions.

Finally, once the factors, which are used in combination with the average cost of a solution arc (see Section 2.2.3), are set to reasonable values, the procedure can generalize well to solutions

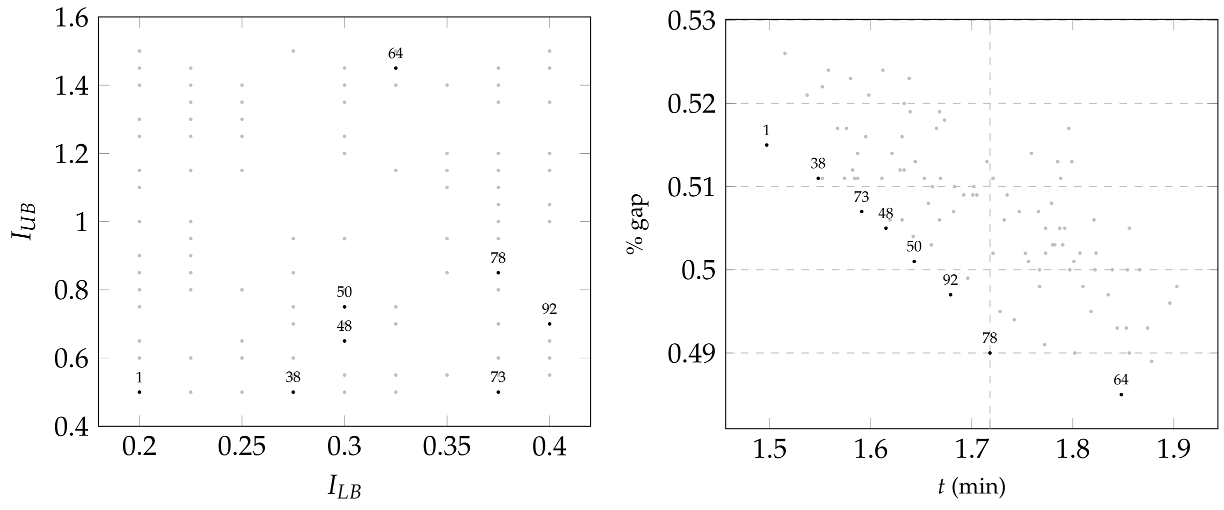


Figure 2.7: Tuning of I_{LB} and I_{UB}. In the left diagram, the configurations we considered in the random search. In the right, the performance associated with each configuration obtained by running FILO with the ones we plot on the left diagram. Only the best configurations are numbered.

and instances with a very different structure, as shown by our computational results.

2.4.4 Move Generators and Granular Neighborhoods

The benefits of GNs are well documented in several papers, such as Toth and Vigo (2003) and Schneider, Schwahn, and Vigo (2017). However, careful tuning is necessary to get the best out of GNs. In fact, different values for the number of neighbors considered in the sparsification rule n_{gs}, the base sparsification factor γ_{base}, and the reduction factor δ affecting the number of nonimproving iterations before increasing a sparsification factor, can dramatically alter the performance of the algorithm and, more specifically, its computing time.

The sparsification increment factor λ may also play a role; however, we fixed it to $\lambda = 2$, as in the original GN definition, and made the other parameters depend on its value.

As for the shaking parameters, we studied the effect of varying n_{gs}, γ_{base} and δ by performing a limited random search among reasonable ranges of values. The selected configurations and the associated performances can be seen in Figure 2.8. In particular, we generated a hundred unique combinations for $n_{gs} \in \{25, 50, 75, 100\}$, $\gamma_{base} \in [0.1, 0.5]$ discretized with steps of 0.025, and $\delta \in [0.1, 1]$ discretized with steps of 0.05. As expected, a larger value for n_{gs} is generally associated with a better final solution quality. However, the associated computing time increment provides only an extremely limited gap improvement.

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

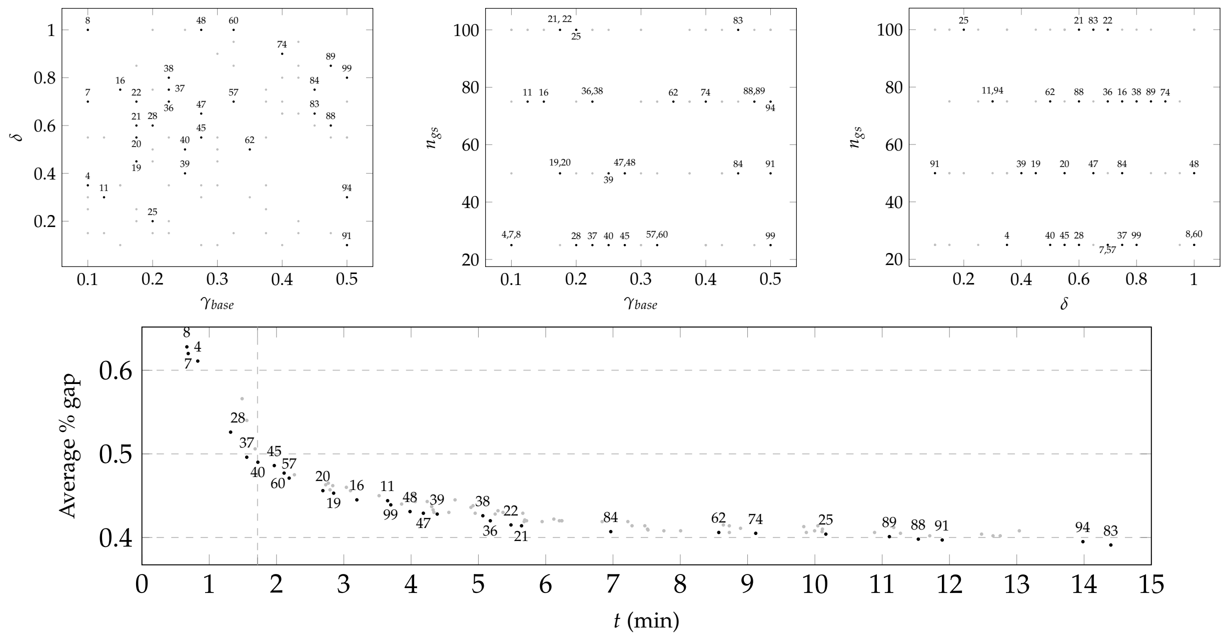


Figure 2.8: Tuning of n_{gs}, γ_{base}, and δ. In the top row, the configurations we considered in the random search. In the bottom row, the performance associated with each configuration obtained by running FILO with the configuration values. Only the best configurations are numbered; seven suboptimal ones, with computing times greater that 15 minutes, are omitted.

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

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

Finally, we compared the vertex-wise management of move generators with the more standard one, in which after a number of $\delta \cdot \Delta_{CO}$ nonimproving iterations the total number of active move generators is doubled; i.e., $\gamma_i = \min\{\gamma_i \cdot \lambda, 1\}, i \in V$. When a solution improving the current best solution is identified, all sparsification factors are reset; i.e., $\gamma_i = \gamma_{base}, i \in V$. By running FILO with move generators managed in the standard way we obtained a gap of 0.51% compared to a gap of 0.49% with the vertex-wise management, and computing times were similar. However, we believe both strategies to be equally effective when properly tuned. To conclude, the proposed vertex-wise management of move generators might be a reasonable and effective alternative generalization of the standard one and better fits the localized optimization design of FILO.

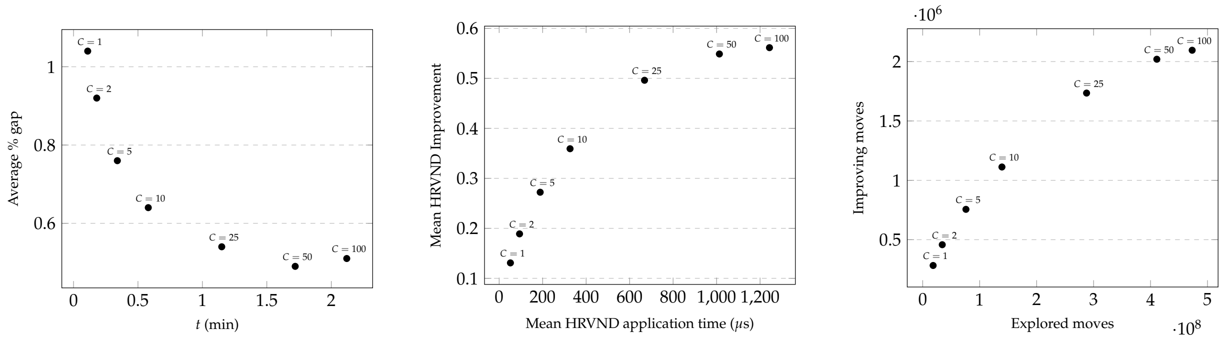


Figure 2.9: Tuning of C. Computational results (left) and local search statistics (center, right) obtained by running FILO with the associated C value.

2.4.5 Selective Vertex Caching

The dimension of the cache C may considerably affect the overall algorithm outcome by indirectly acting on the different components employed: a smaller C value will promote a milder shaking intensity. In fact, most of the customers involved in stronger shakings would not be considered by subsequent local search application because they are no longer cached, due to the limit imposed by C. Furthermore, a low C value would cause the local search to perform fewer improvements per iteration, reducing the likelihood of improving the best solution S^*. As a result, sparsification factors $\gamma_i, i \in V$ might reach larger values compared to scenarios using a larger C. Figure 2.9 illustrates the average performance of FILO and statistics related to the local search execution as C is varied.

In our implementation, we selected $C = 50$ because it produces solutions of a quality comparable to larger values but with shorter computing times. Note, however, that the SVC and shaking guiding strategy are highly interconnected components. In fact, even when $C >> 100$, results and statistics remain comparable to those with $C \approx 100$, because of the limits on the number of ruined customers imposed by the shaking guiding strategy.

Figure 2.10 provides a number of hints about the scalability of FILO.

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

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

2.4.6 Extreme Runs

In this section we investigate whether allowing longer computations is enough to improve the quality of final solutions. This question is partially answered by the computational results of Section 2.3, where we considered the FILO (long) version. In addition we performed an even longer run by setting $\Delta_{CO} = 10^7$. The average gap for the subset of large instances of the $\mathbb{X}$ decreased from the 0.30% with FILO (long) to 0.19% in 183.74 minutes. Moreover, we found three new best solutions. The increment in the computing time remains quite constant: i.e.,

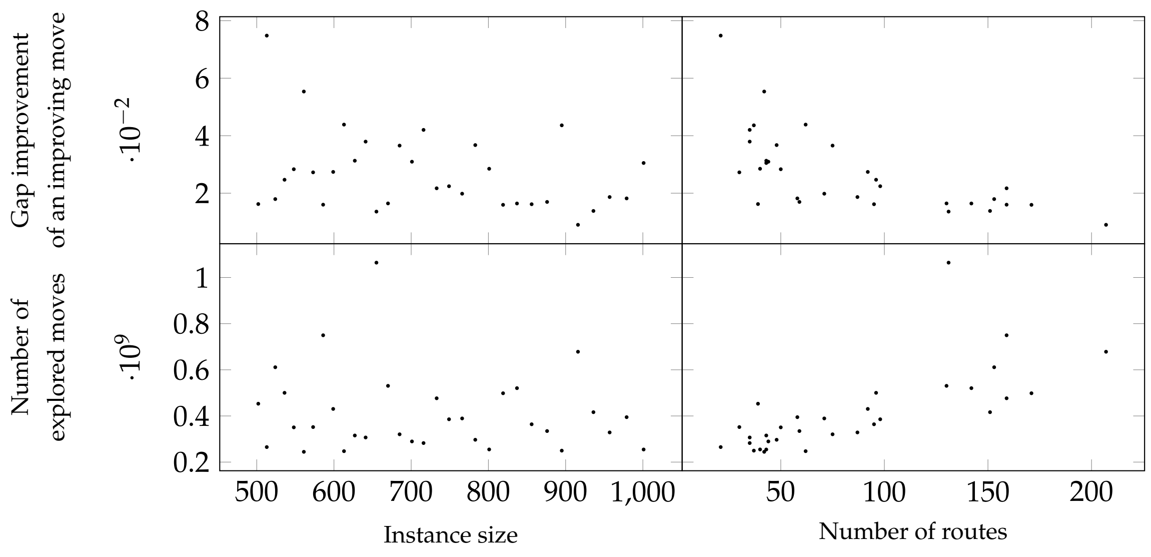


Figure 2.10: Statistics for local search operators while varying the instance size and the nominal number of routes defined in the $\mathbb{X}$ instance name.

increasing the number of iterations by a factor of ten increases the computing time about ten times. Full details are given in Section A.3 of the Appendix.

Finally, we studied whether performing a larger number of runs per instance drastically changes the average result and computing time. To this end, we compared the results obtained on the large scale $\mathbb{X}$ instances in 50 runs (described in Section 2.3.3) with the results obtained by performing 100 runs. Seeds were selected as described in Section 2.3.1. The average gap (rounded to two decimal places) when performing 100 runs increased to 0.51% and the computing time remained the same; see Figure 2.11.

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

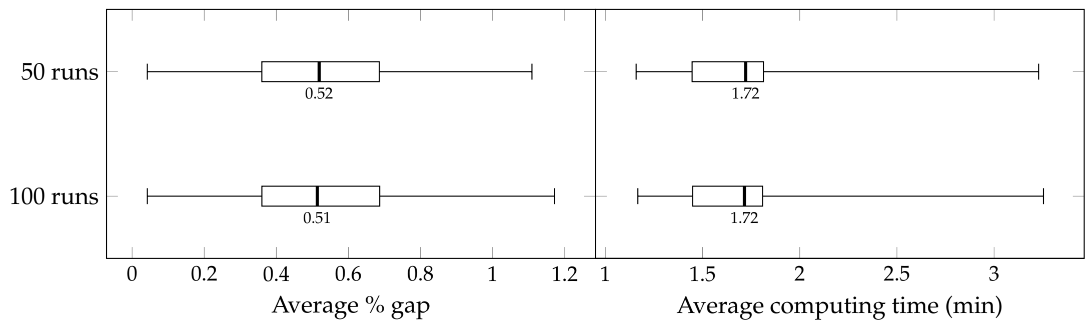


Figure 2.11: Average gap and computing time comparison while varying the number of runs per instance. The median value is shown below each boxplot.

2.4.7 Simplified Versions of FILO

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

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

- The HRVND is replaced by a RVND, containing only simple local search operators, identified as the most effective according to Figure 2.14 of Section A.2 of the Appendix. In particular, we selected 11EX, 10EX, TAILS and SPLIT because they have a high success rate (greater than 45%).
- The selected neighborhoods are explored according to a best-improvement strategy, without exploiting SMDs.
- The route minimization phase is never executed.

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

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

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

The second version, called FILO0, further simplifies FILO1 by retaining only the optimization due to the random walk ruin-and-recreate, without the application of any local search - thus removing all components associated with the local search engine. Each $\mathbb{X}$ and $\mathbb{B}$ instance was solved 50 times with seeds defined as in Section 2.3.1. Parameters for the preserved components have the same tuning as in Section 2.3.2. From the results (shown in Figure 2.12) we can summarize that:

- FILO0, inspired by the effectiveness of the work by Christiaens and Vanden Berghe (2020), confirms that an accurate design of a ruin-and-recreate procedure, coupled with an effective diversification strategy, may be enough to find good-quality solutions for medium to large instances in negligible computing time. However, results on very large-scale instances show that, given the same number of iterations, other strategies are necessary to improve the effectiveness of the method.
- FILO1, given enough iterations, proved to be competitive with FILO on the $\mathbb{X}$ dataset. However, it is not always Pareto-optimal on the largest $\mathbb{B}$ instances. The reduced number of local search operators thus has a significant impact for the proposed algorithm when moving to very large-scale instances. Finally, the SVC and GNs allow us to limit the computing time of FILO1 considerably, because the filtering of GNs is fairly aggressive with the proposed parametrizations. We strongly suggest, however, the adoption of SMDs

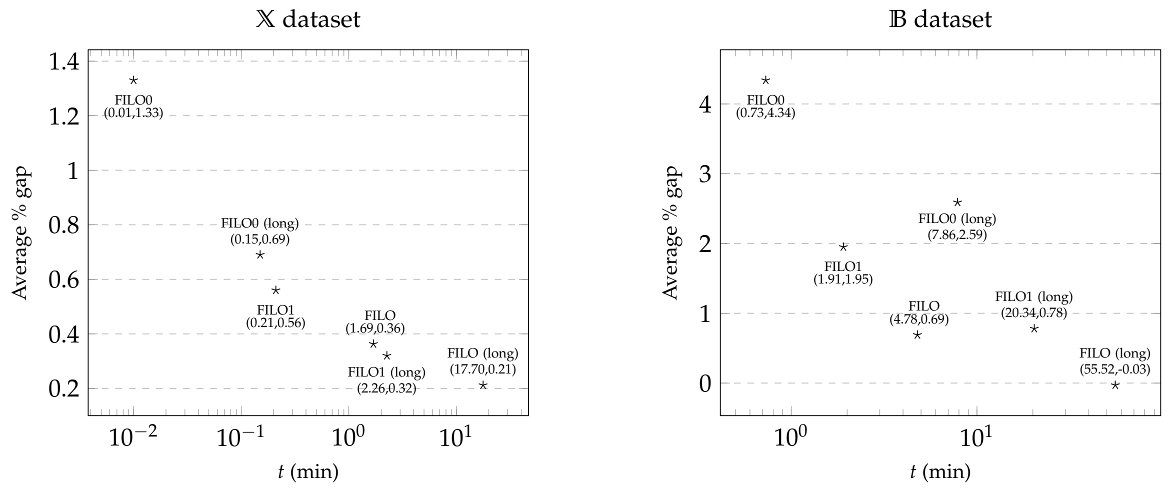


Figure 2.12: Performance comparison of FILO variants on the $\mathbb{X}$ (left) and $\mathbb{B}$ (right) instances.

in VND settings, which would allow considerable savings in the computational effort (see, Zachariadis and Kiranoudis (2010) and Beek et al. (2018)). In fact, the computing time of FILO1 is reduced in average of about 10% when the implementation exploits SMDs even if using few local search operators and very aggressive neighborhood filtering strategies.

2.5 Conclusions

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

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

Despite the exploration of several neighborhoods, the method is still very fast, thanks to an accurate design; it also greatly benefits from the above-mentioned acceleration and pruning techniques. Moreover, FILO exhibits a computing time that grows linearly with respect to the instance size, making it very suitable for solving very large-scale instances without sacrificing the quality of the final solutions. In fact, it was able to find several new best solutions for very large-scale instances and two new best solutions for the well-studied $\mathbb{X}$ dataset proposed by Uchoa et al. (2017).

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

2.6 Acknowledgments

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

A Algorithmic Components Analysis: Additional Material

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

A.1 Initial Solution Definition

Consider the scenario depicted in Figure 2.13, showing a single run for instance X-n936-k136. This example illustrates the evolution of the route minimization procedure with $\Delta_{RM} = 1000$ on the left and of the core optimization procedure with $\Delta_{CO} = 5000$ on the right. Both procedures ran for about three seconds and were applied to the same starting solution generated by the construction phase. By moving into the infeasible space, the route minimization procedure is very effective in quickly improving and compacting trivially bad initial solutions. However, since its structure is specifically designed to reduce the number of routes, the improvements vanish after a few hundred iterations.

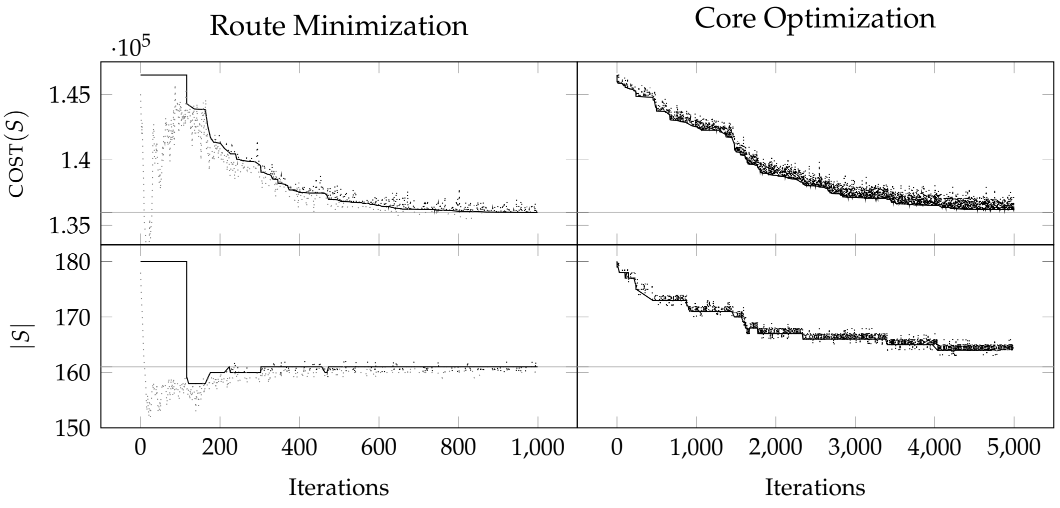


Figure 2.13: Example of improvement procedures evolution when applied to the same initial solution for instance X-n936-k151. The continuous line shows the cost (top) and number of routes (bottom) associated with the best solution (in terms of cost) found up to that iteration; dots represents the current search trajectory. Gray and black dots are associated with infeasible and feasible solutions, respectively.

A.2 Local Search

Figure 2.14 shows, for each local search operator applied to a shaken solution, the gap improvement when successfully applied, the application time in 10^{-6} seconds, and the success ratio computed as the number of improving applications over the total number of attempts. Similarly, Figure 2.15 shows the effect of EJCH$(\cdot)$ when applied to solutions that are already a local optimum for the first HRVND tier.

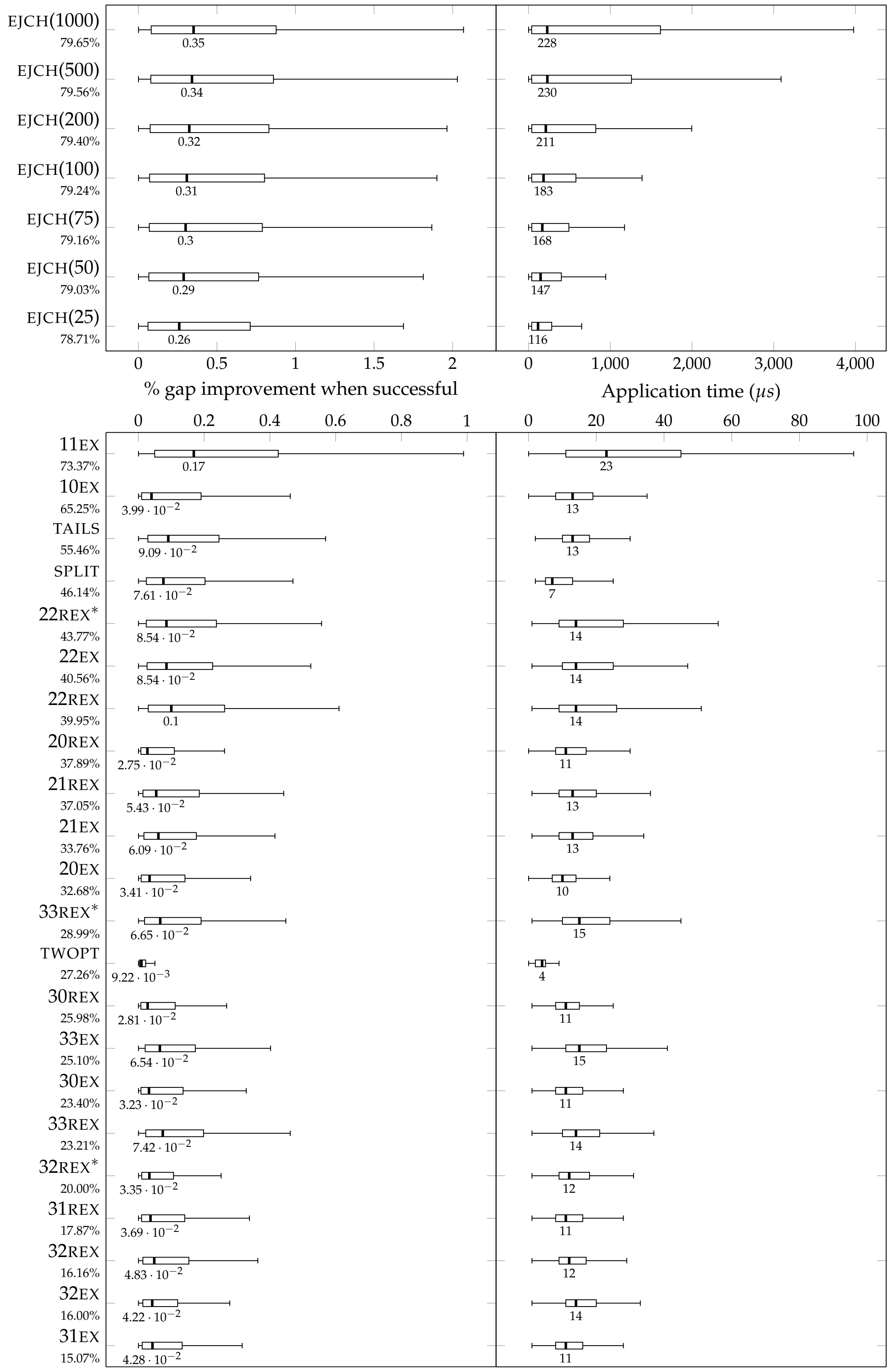


Figure 2.14: Statistics for local search operators when applied to shaken solutions. For each operator, we report the expected gap improvement when successfully applied (left), the total exploration time (right), and the success ratio (below each operator's name). The median value is shown below each boxplot.

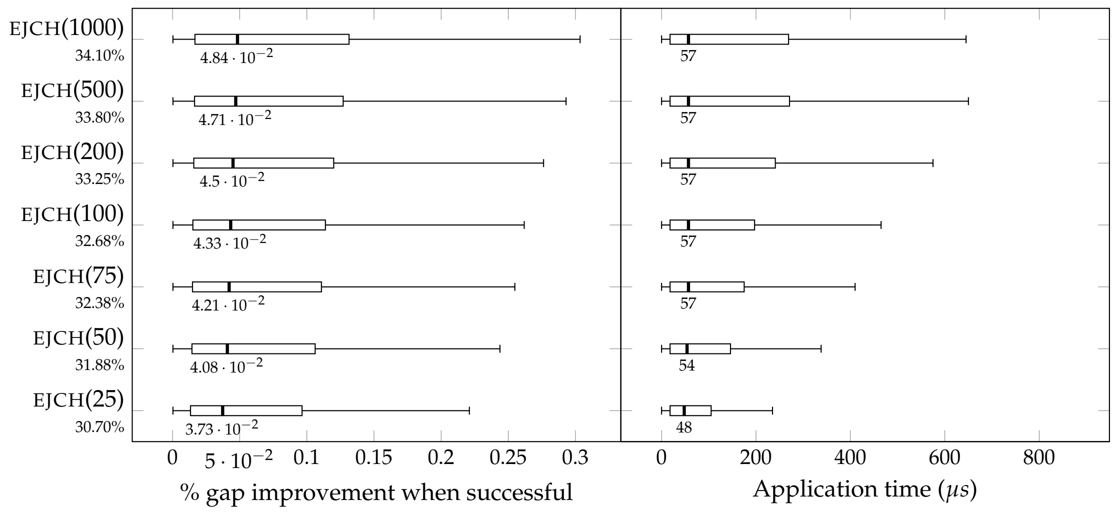


Figure 2.15: Statistics for local search operators when applied to HRVND first tier local optima. For each operator, we report the expected gap improvement when successfully applied (left), the total exploration time (right), and the success ratio (below each operator's name). The median value is shown below each boxplot.

A.3 Extreme Runs

Table 2.7 shows the results we obtained by setting $\Delta_{CO} = 10^7$.

A.4 Computations with Limited Memory Footprint

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

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

Table 2.7: Long computations on large-sized $\mathbb{X}$ instances.
New best solutions: (X-n801-k40, 73311);(X-n895-k37, 53906);(X-n957-k87, 85467).

B Move Generators and Static Move Descriptors

Efficiently managing move generators is crucial for any local search-based algorithm making use of GNs. In addition, flexibility might be required when experimenting different sparsification rules and composition of them, as we did in the analysis proposed in Section 2.4.4. In the following sections, we first discuss a flexible but still efficient implementation of move generators for the symmetric CVRP, supporting the union of different sparsification rules and a vertex-wise dynamic management. Then, we show how it can be used to efficiently implement SMD-based local search operators.

B.1 Move Generators: Storage and Management

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

In the following, we describe a possible implementation of the above defined general framework for move generators. An illustrative example, representing the implementation of a set of move generators T defined by the union of two sparsification rules r_0 and r_1, is shown in Figure 2.16.

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

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

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

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

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

Depending on how the inclusion percentage values are defined we can model the classical or the vertex-wise management of the dynamic move generators for a sparsification rule $r \in R$.

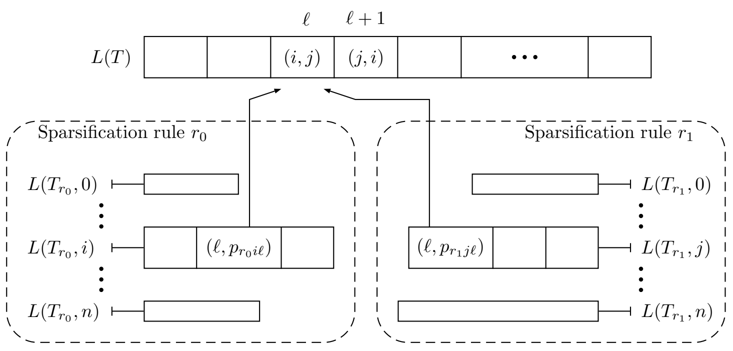


Figure 2.16: Implementation of the list $L(T)$ of move generators defined by two sparsification rules r_0 and r_1. Two move generators (i,j) and (j,i) indexed ℓ and $\ell+1$ respectively, are explicitly shown. As can be seen, those move generators are defined by both sparsification rules and the associated entry in the lists point to the same shared entry in $L(T)$. The inclusion percentages $p_{r_0 i\ell}$ and $p_{r_1 j\ell}$ might however cause those move generators to be active at different times according to the value of γ_i and γ_j.

In the classical management $\sum_{i\in V}\sum_{\ell\in L(T_r,i)} p_{ri\ell} = 1$ whereas in the vertex-wise management $\sum_{\ell\in L(T_r,i)} p_{ri\ell} = 1$ for each $i \in V$.

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

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

In our implementation, the Sparsification Rule r_0 described in Section 2.2.2 is implemented by defining $p_{r_0 v\ell} = n/|L(T_{r_0},v)|$, where n is the index of move generator ℓ in a list of move generators $(i,j) \in L(T_{r_0},i)$ sorted in increasing c_{ij} cost. The additional Sparsification Rule r_1 employed in the analysis of Section 2.4.4 defines instead $p_{r_1 v\ell} = n/|T_{r_1}|$, where n is the index of move generator ℓ in a list of move generators $(i,j) \in L(T_{r_1})$ sorted in increasing c_{ij} cost.

B.2 An Abstract SMD-based Local Search Operator

The general structure of all SMD-based local search operators we used is shown in Algorithm 7. Note that, as mentioned in Section 2.2.2, we use the terms SMD and move generator interchangeably.

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

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

For each move generator $(i, j) \in T^\gamma(S)$, the δ-tag $\delta(i, j)$, identifying the effect of the application of the move induced by (i, j) on S, is computed. Every (i, j) inducing an improving move (i.e., $\delta(i, j) < 0$) is inserted into the heap data structure $\mathcal{H}$. By only inserting improving move generators, the heap computational complexity is kept at its minimum.

The heap $\mathcal{H}$ is linearly scanned until a feasible move is found. If such a move cannot be found, then S is considered to be a local optimum with respect to the local search operator under examination. Note that by heuristically restricting the initialization stage to only consider move generators involving vertices in $\bar{V}_S$ some improvement may be overlooked, but, as shown by the experiments in Section 2.4.5, this does not significantly affect the final solution quality.

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

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

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

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

Restricted Update for Asymmetric Neighborhoods. The δ-tag update of move generators involving a vertex $i \in A$ for a local search operator defining an asymmetric neighborhood may sometimes be restricted from $\{(i, v_2), (v_1, i) : v_1, v_2 \in V\} \cap T^\gamma$ to only one between $\{(i, v_2) : v_2 \in V\} \cap T^\gamma$ and $\{(v_1, i) : v_1 \in V\} \cap T^\gamma$. As an example, consider the 3OEX application shown

Algorithm 7 Abstract SMD-Based Local Search Operator

```
 1: procedure APPLY($S, T^\gamma$)
 2:     PREPROCESS($S$)
 3:     $\mathcal{H} \leftarrow$ INITIALIZATION($S, T^\gamma$)
 4:     $n \leftarrow 0$
 5:     while $n < len(\mathcal{H})$ do
 6:         $(i, j) \leftarrow$ PEEK($\mathcal{H}, n$)
 7:         $n \leftarrow n + 1$
 8:         if $\neg$ISFEASIBLE($S, (i, j)$) then continue
 9:         $A \leftarrow$ AFFECTEDVERTICES($S, (i, j)$)
10:         $S \leftarrow$ EXECUTE($S, (i, j)$)
11:         UPDATE($\mathcal{H}, T^\gamma, A$)
12:         $n \leftarrow 0$
13:     end while
14: end procedure
```

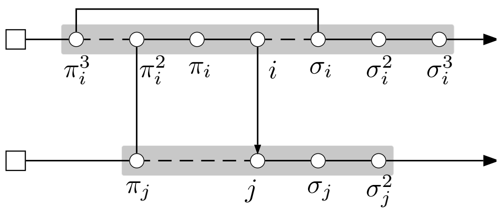


Figure 2.17: A 3OEX application induced by move generator (i, j) relocating path $(\pi_i^2 - i)$ between π_j and j. Some SMDs involving vertices in the gray area require an update to their δ-tag after the move execution.

in Figure 2.17. The set of affected vertices is $A = \{\pi_i^3, \pi_i^2, \pi_i, i, \sigma_i, \sigma_i^2, \sigma_i^3, \pi_j, j, \sigma_j, \sigma_j^2\}$. However, not all move generators $\{(i, j), (j, i) : i \in A, j \in V\} \cap T^\gamma$ have to be updated. For example, considering vertex π_i^3, move generators $(\pi_i^3, j), j \in V$ require an update since the successor of π_i^3 changes after the move execution, however, move generators $(j, \pi_i^3), j \in V$ do not, in fact, the predecessor of π_i^3 remains the same. For operator 3OEX (and ignoring the current sparsification level), the total number of move generators requiring an update after a 3OEX application can be reduced of approximately 36%.

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

Dynamic Vertex-wise Move Generators for SMD-based Local Search Operators

Vertex-wise management of move generators requires a little extra care for the SMD update stage to be correctly performed. To highlight this, consider a scenario in which a vertex j is currently marked as cached at the beginning of a neighborhood exploration for a solution S; that is, $j \in \bar{V}_S$. An illustrative example is shown in Figure 2.18. The figure represents a portion of an instance with six customers together with a number of move generators depicted as lines ending with little circles. In particular, for a move generator (v_1, v_2), a full circle near to v_1 means that the move generator is currently active in v_1, i.e., $p_{rv_1\ell} \leq \gamma_{v_1}$ where ℓ is the index of $(v_1, v_2)_\ell$ in $L(T)$, while an empty circle represents the opposite scenario, i.e., $p_{rv_1\ell} > \gamma_{v_1}$. In the example, (i, j) is active in j but not in i. Finally, the grayed area identifies the set $\bar{V}_S$ of currently cached vertices for S.

During the SMD initialization stage, move generators involving vertices in $\bar{V}_S$ are considered to be inserted into the heap data structure $\mathcal{H}$. Suppose that, because inducing an improving

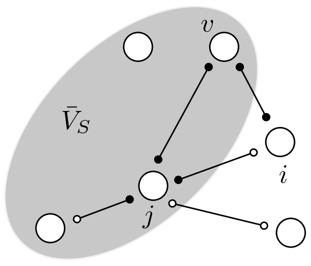


Figure 2.18: A portion of an instance containing six customers (circles) and five move generators (lines). A move generator $(v_1, v_2)_\ell$ active in v_1, i.e., $p_{rv_1\ell} \leq \gamma_{v_1}$, is represented with a full circle near to v_1, whereas an empty circle denotes the opposite, i.e., $p_{rv_1\ell} > \gamma_{v_1}$. As an example (i, j) is active in j but not in i. The direction of move generators is not shown because not relevant.

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

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

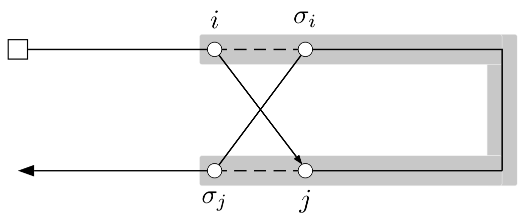
Finally, note that this approach works correctly under the assumption that the sparsification vector γ is not changed during a neighborhood exploration.

B.3 SMD Implementation Details of Local Search Operators

In the following, we provide a detailed description of the implementation for the local search operators used in FILO. In particular, we detail the operator-dependant procedures to be defined when applying a move induced by a move generator (i, j) within an SMD-based operator. To better accomplish this, we introduce few additional notation elements. In particular, we denote by π_i^ℓ and σ_i^ℓ the ℓ^{th}- predecessor and successor of vertex $i \in V$, respectively, in the solution under examination. We omit the apex when $\ell = 1$. Moreover, we identify with q_i^{up} and q_i^{from} the cumulative load up to and from any customer $i \in V_c$ included, i.e., $q_i^{up} = q_i + q_{\pi_i} + q_{\pi_i^2} + \ldots + q_0$ and $q_i^{from} = q_i + q_{\sigma_i} + q_{\sigma_i^2} + \ldots + q_0$. In the following paragraphs, a figure is shown for each local search operator highlighting the move induced by a move generator (i, j) and the set of vertices affected by its application (grayed area). Note that the move generator direction does not necessarily reflect the crossing direction in the resulting route. Finally, as defined in Section 2.2.2, path is used to refer to a contiguous sequence of vertices belonging to a route, and head and tail are used to denote a path belonging respectively to the initial and final part of a route.

TWOPT

Replace a path of vertices with its reverse.

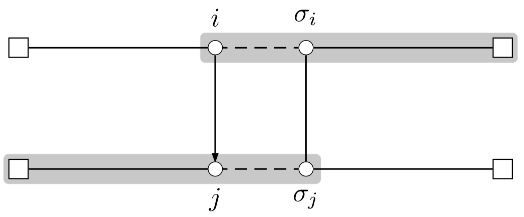


- Type: symmetric.

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

SPLIT

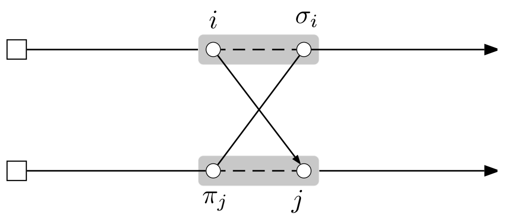
Replace the tail of route r_i with the reversed head of route r_j and replace the head of route r_j with the reversed tail of route r_i.



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

TAILS

Swap the tails of two different routes.

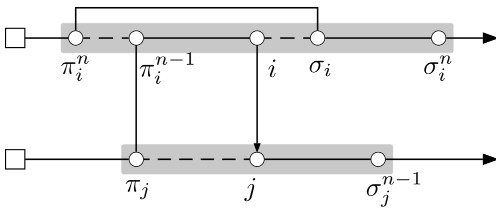


- Type: asymmetric.
- Pre-processing: compute q_i^{up} and q_i^{from} for any customer $i \in V_c$.
- Cost Computation: $\delta_{ij} = -c_{i\sigma_i} + c_{ij} - c_{j\pi_j} + c_{\pi_j\sigma_i}$.
- Feasibility Check: $(r_i \neq r_j) \wedge (q_i^{up} + q_j^{from} \leq Q) \wedge (q_{\pi_j}^{up} + q_{\sigma_i}^{from}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

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

*n*0EX with $n \geq 1$

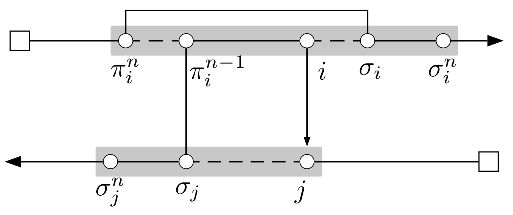
Relocate a path of n vertices within the same or into a different route.



- Type: asymmetric.
- Pre-processing: none.
- Cost Computation: $\delta_{ij} = -c_{\pi_i^n \pi_i^{n-1}} - c_{i\sigma_i} - c_{j\pi_j} + c_{\pi_i^n \sigma_i} + c_{\pi_j \pi_i^{n-1}} + c_{ij}$.
- Feasibility Check: logical disjunction of
  - $(r_i = r_j) \wedge \bigwedge_{\ell=1}^{n-1} (j \neq \pi_i^\ell) \wedge (j \neq \sigma_i)$.
    When $r_i = r_j$ and $\bigvee_{\ell=1}^{n-1} j = \pi_i^\ell$, the path to relocate overlaps with the destination position.
    When $r_i = r_j$ and $j = \sigma_i$, the move does nothing but the δ_{ij} computation is not correct.
  - $(r_i \neq r_j) \wedge (i \neq 0) \wedge \bigwedge_{\ell=1}^{n-1} (\pi_i^\ell \neq 0) \wedge (q_{r_j} + q_i + \sum_{\ell=1}^{n-1} q_{\pi_i^\ell} \leq Q)$.
    The condition makes sure the depot is not relocated and the capacity constraint of the target route is respected.
- Update List: $\bigcup_{\ell=1}^{n} \pi_i^\ell \cup i \cup \bigcup_{\ell=1}^{n} \sigma_i^\ell \cup \pi_j \cup j \cup \bigcup_{\ell=1}^{n-1} \sigma_j^\ell$.
  The update can be restricted to move generators
  - $\{(\pi_i^n, v), (\pi_i^{n-1}, v), (v, \pi_i^{n-1}) : v \in V\} \cap T^\gamma$;
  - $\{(\pi_i^\ell, v) : \ell = n-2, \dots, 1 \text{ and } v \in V\} \cap T^\gamma$;
  - $\{(i,v) : v \in V\} \cap T^\gamma$;
  - if $n = 1$ include $\{(v,i) : v \in V\} \cap T^\gamma$;
  - $\{(\sigma_i, v), (v, \sigma_i) : v \in V\} \cap T^\gamma$;
  - $\{(\sigma_i^\ell, v) : \ell = 2, \dots, n \text{ and } v \in V\} \cap T^\gamma$;
  - $\{(\pi_j, v) : v \in V\} \cap T^\gamma$;
  - $\{(j,v), (v,j) : v \in V\} \cap T^\gamma$;
  - $\{(\sigma_j^\ell, v) : \ell = 1, \dots, n-1 \text{ and } v \in V\} \cap T^\gamma$.
- Execution: replace $(\pi_i^n, \pi_i^{n-1}), (i, \sigma_i)$ and (j, π_j) with $(\pi_i^n, \sigma_i), (\pi_i^{n-1}, \pi_j)$ and (i,j).

*n*0REX with $n \geq 2$

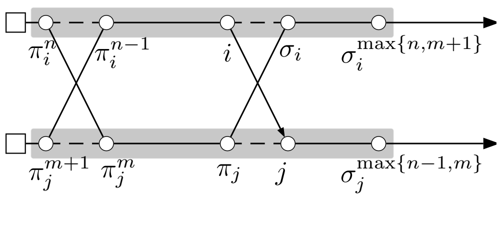
Relocate a reversed path of n vertices within the same or into a different route.



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

nmEX with $1 \leq m \leq n$

Swap a path of n vertices with a path of m vertices within the same or between different routes.



- Type: asymmetric.
- Pre-processing: none.
- Cost Computation: $\delta_{ij} = -c_{\pi_i^n \pi_i^{n-1}} - c_{i\sigma_i} - c_{\pi_j^{m+1} \pi_j^m} - c_{j\pi_j} + c_{\pi_i^n \pi_j^m} + c_{\pi_j \sigma_i} + c_{\pi_j^{m+1} \pi_i^{n-1}} + c_{ij}$.
- Feasibility Check: logical disjunction of
  - $(r_i = r_j) \wedge \bigwedge_{\ell=1}^{n-1} (j \neq \pi_i^\ell) \wedge \bigwedge_{\ell=1}^{m+1} (j \neq \sigma_i^\ell)$.
    When $r_i = r_j$ and $\bigvee_{\ell=1}^{m} j = \sigma_i^\ell \vee \bigvee_{\ell=1}^{n-2} j = \pi_i^\ell$, the paths overlap.
    When $r_i = r_j$ and $j = \sigma_i^{m+1}$, the move could be reduced to a $m0$EX induced by move generator (σ_i, π_i^{n-1}) or to a $n0$EX induced by move generator (i, j) but would require a special handling in this context.

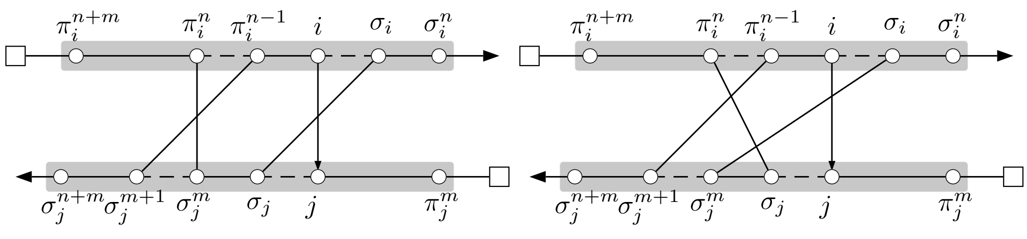
When $r_i = r_j$ and $j = \pi_i^{n-1}$, vertex j is at the same time part of the path to move and destination of the movement.

– $(r_i \neq r_j) \wedge (i \neq 0) \wedge \bigwedge_{\ell=1}^{n-1}(\pi_i^\ell \neq 0) \wedge \bigwedge_{\ell=1}^{m}(\pi_j^\ell \neq 0) \wedge (q_{r_j} + q_i + \sum_{\ell=1}^{n-1} q_{\pi_i^\ell} - \sum_{\ell=1}^{m} q_{\pi_j^\ell} \leq Q) \wedge (q_{r_i} - q_i - \sum_{\ell=1}^{n-1} q_{\pi_i^\ell} + \sum_{\ell=1}^{m} q_{\pi_j^\ell} \leq Q)$.
The condition makes sure the depot is not relocated and the capacity constraints are not violated.

- Update List: $\bigcup_{\ell=1}^{n} \pi_i^\ell \cup i \cup \bigcup_{\ell=1}^{\max\{n,m+1\}} \sigma_i^\ell \cup \bigcup_{\ell=1}^{m+1} \pi_j^\ell \cup j \cup \bigcup_{\ell=1}^{\max\{n-1,m\}} \sigma_j^\ell$.
The update can be restricted to move generators
  – $\{(\pi_i^n, v) : v \in V\} \cap T^\gamma$;
  – $\{(\pi_i^\ell, v) : \ell = n-1, \ldots, 1 \text{ and } v \in V\} \cap T^\gamma$;
  – $\{(v, \pi_i^\ell) : \ell = n-1, \ldots, n-1-m \text{ and } v \in V\} \cap T^\gamma$;
  – $\{(i, v) : v \in V\} \cap T^\gamma$;
  – if $n - m < 2$ include $\{(v, i) : v \in V\} \cap T^\gamma$;
  – $\{(\sigma_i, v), (v, \sigma_i) : v \in V\} \cap T^\gamma$;
  – $\{(\sigma_i^\ell, v) : \ell = 2, \ldots, n \text{ and } v \in V\} \cap T^\gamma$;
  – $\{(v, \sigma_i^\ell) : \ell = 2, \ldots, m+1 \text{ and } v \in V\} \cap T^\gamma$;
  – $\{(\pi_j^{m+1}, v) : v \in V\} \cap T^\gamma$;
  – $\{(\pi_j^\ell, v), (v, \pi_j^\ell) : \ell = m, \ldots, 1 \text{ and } v \in V\} \cap T^\gamma$;
  – $\{(j, v), (v, j) : v \in V\} \cap T^\gamma$
  – $\{(\sigma_j^\ell, v) : \ell = 1, \ldots, n-1 \text{ and } v \in V\} \cap T^\gamma$;
  – $\{(v, \sigma_j^\ell) : \ell = 1, \ldots, m \text{ and } v \in V\} \cap T^\gamma$.
- Execution: replace (π_i^n, π_i^{n-1}), (i, σ_i), (π_j^{m+1}, π_j^m) and (j, π_j) with (π_i^n, π_j^m), $(\pi_i^{n-1}, \pi_j^{m+1})$, (π_j, σ_i) and (i, j).

***nm*REX (and *nm*REX*) with $1 \leq m \leq n$**

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



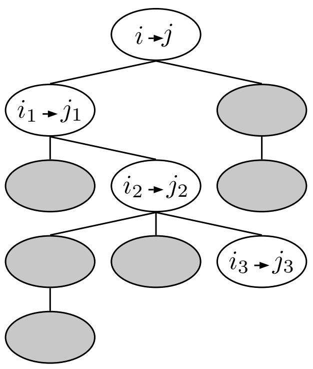
- Type: asymmetric.
- Pre-processing: none.
- Cost Computation:
  – *nm*REX*: $\delta_{ij} = -c_{\pi_i^n \pi_i^{n-1}} - c_{i\sigma_i} - c_{j\sigma_j} - c_{\sigma_j^m \sigma_j^{m+1}} + c_{\pi_i^n \sigma_j^m} + c_{\pi_i^{n-1} \sigma_j^{m+1}} + c_{\sigma_j \sigma_i} + c_{ij}$.
  – *nm*REX: $\delta_{ij} = -c_{\pi_i^n \pi_i^{n-1}} - c_{i\sigma_i} - c_{j\sigma_j} - c_{\sigma_j^m \sigma_j^{m+1}} + c_{\pi_i^n \sigma_j} + c_{\pi_i^{n-1} \sigma_j^{m+1}} + c_{\sigma_j^m \sigma_i} + c_{ij}$.

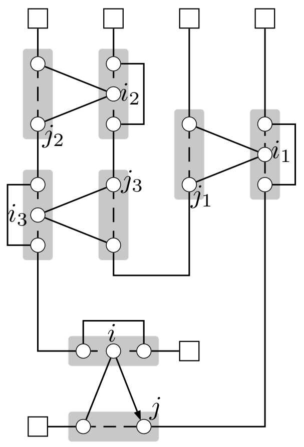
- Feasibility Check: logical disjunction of
  - $(r_i = r_j) \wedge \bigwedge_{\ell=1}^{n+m}(j \neq \pi_i^\ell)$.
    When $r_i = r_j$ and $\bigvee_{\ell=1}^{n+m-1} j = \sigma_i^\ell$, the paths overlap.
    When $r_i = r_j$ and $j = \sigma_i^{m+n}$
    - *nm*REX*: the move could be reduced to a TWOPT induced by move generator (j, i);
    - *nm*REX: the move could be reduced to a *n*0REX induced by move generator (i, j);

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

EJCH

Perform a sequence of 10EX applications.





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

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

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

- $q^s_{r_{i_{n+1}}} - q_{i_{n+1}} \leq Q$.
  The removal of i_{n+1} restores the feasibility of route $r_{i_{n+1}}$ that was violated by the previous insertion of j_n.
- $i_{n+1} \notin \mathcal{F}^s_i$.
  Customer i_{n+1} can be relocated.

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

- $j_{n+1} \neq 0$.
  The depot alone does not allow to identify a specific route.
- $r_{i_{n+1}} \neq r_{j_{n+1}}$.
  Customer i_{n+1} is relocated into a route different from the origin one.
- $\delta_s + \delta_{i_{n+1}j_{n+1}} < 0$.
  Sequence s still provides an improvement to the objective function.

  - $j_{n+1} \notin \mathcal{F}_j^s$.
    Customer j_{n+1} can be the target of a relocation.

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

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

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

- Update List: $\mathcal{F}_i^{s^*}$ with s^* a feasible improving sequence.
  For each 10EX (i, j) move generator composing the sequence s^*, the update can be restricted to move generators $\{(\pi_i, v), (i, v), (v, i), (\sigma_i, v), (v, \sigma_i), (j, v), (v, j), (\pi_j, v) : v \in V\} \cap T^\gamma$

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

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

Detailed results about computations on the $\mathbb{X}$ dataset can be found in Tables 2.9 – 2.11 and Figure 2.19. Moreover, to better assess whether the results obtained by FILO are statistically different with respect to competing algorithms, we followed the procedure used in Christiaens and Vanden Berghe (2020). In particular, we conducted a one-tailed Wilcoxon signed-rand test (Wilcoxon (1945)) in which we consider a null hypothesis H_0

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

and an alternative hypothesis H_1

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

where X can be ILS-SP, HGSADC, KGLS, and SISR. We tested the above hypotheses on small, medium, large and over all the instances. The p-values associated with the tested hypothesis are shown in Table 2.8 (left). The p-values for a similar analysis in which we compared FILO (long) with competing methods are shown in Table 2.8 (right).

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 the two compared methods are not statistically different. On the other hand, when H_0 is rejected, the average results obtained by the methods are statistically different and the alternative hypothesis H_1 can be tested to find whether the average results obtained by FILO are statistically greater than those of the competing method. Rejecting H_1 implies that FILO performs better than the competing method.

When performing multiple comparisons involving the same data, the probability of erroneously rejecting a null hypothesis increases. To control these errors, the significance level α is typically adjusted to lower values. Bonferroni correction (Dunn (1961)) is a simple method used to adjust α when performing multiple comparisons. In particular, given n comparisons, the significance level is set to α/n. In our case, for each FILO configuration, we tested a total number of $n = 8$ hypotheses corresponding to the partitioning of instances (Small, Medium, Large, and All) and

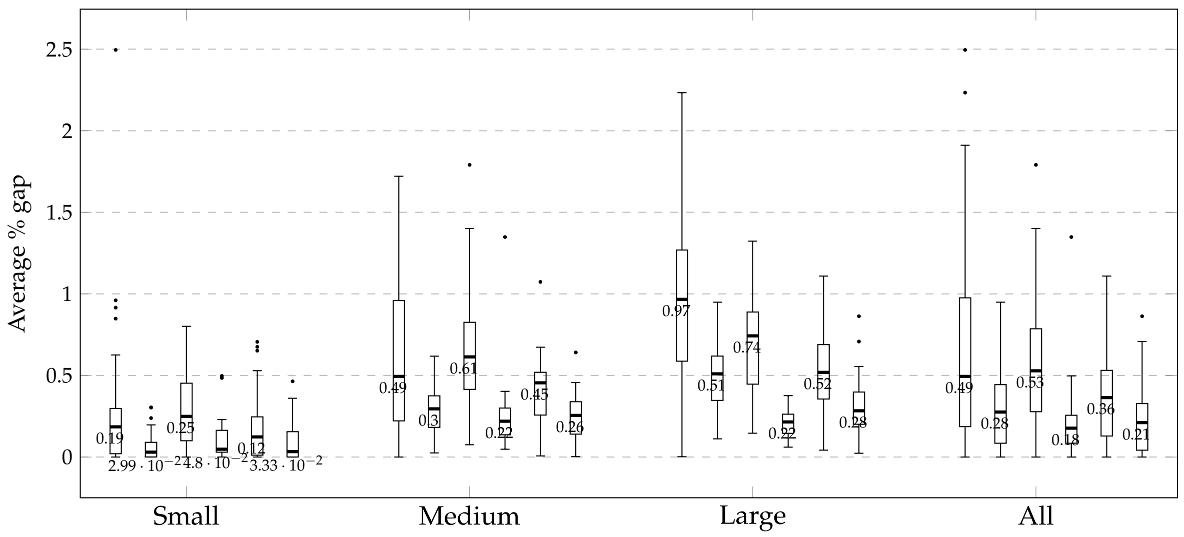


Figure 2.19: Comparison of average gaps obtained by algorithms on the $\mathbb{X}$ dataset. For each group, boxplots, from left to right, are associated with: ILS-SP, HGSADC, KGLS, SISR, FILO, and FILO (long).

	Small			
	ILS-SP	HGSADC	KGLS	SISR
H_0	0.309491	**0.000273**	0.020427	0.064141
H_1	0.154746	0.999876	0.010214	0.969381
	Similar	Worse	Similar	Similar
	Medium			
	ILS-SP	HGSADC	KGLS	SISR
H_0	0.071754	**0.001737**	**0.000000**	**0.000066**
H_1	0.035877	0.999183	**0.000000**	0.999970
	Similar	Worse	Better	Worse
	Large			
	ILS-SP	HGSADC	KGLS	SISR
H_0	**0.000335**	1.000000	**0.000234**	**0.000000**
H_1	**0.000167**	0.503678	**0.000117**	1.000000
	Better	Similar	Better	Worse
	All			
	ILS-SP	HGSADC	KGLS	SISR
H_0	**0.000036**	0.007739	**0.000000**	**0.000000**
H_1	**0.000018**	0.996173	**0.000000**	1.000000
	Better	Similar	Better	Worse

	Small			
	ILS-SP	HGSADC	KGLS	SISR
H_0	**0.000273**	0.808654	**0.000140**	0.130121
H_1	**0.000136**	0.404327	**0.000070**	0.065061
	Better	Similar	Better	Similar
	Medium			
	ILS-SP	HGSADC	KGLS	SISR
H_0	**0.001057**	0.046584	**0.000000**	0.346122
H_1	**0.000528**	0.023292	**0.000000**	0.830925
	Better	Similar	Better	Similar
	Large			
	ILS-SP	HGSADC	KGLS	SISR
H_0	**0.000000**	**0.000837**	**0.000000**	**0.002227**
H_1	**0.000000**	**0.000419**	**0.000000**	0.998963
	Better	Better	Better	Worse
	All			
	ILS-SP	HGSADC	KGLS	SISR
H_0	**0.000000**	**0.000111**	**0.000000**	0.065432
H_1	**0.000000**	**0.000056**	**0.000000**	0.967546
	Better	Better	Better	Similar

Table 2.8: Computations on the $\mathbb{X}$ dataset: p-values for FILO on the left and 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, FILO is not statistically different from the competing method when H_0 cannot be rejected (Similar), FILO is statistically better when both H_0 and H_1 are rejected (Better), and, finally, FILO is statistically worse when H_0 is rejected and H_1 is not rejected (Worse).

to the two hypotheses (H_0 and H_1). Thus, by assuming an initial significance level $\alpha_0 = 0.025$, the adjusted value becomes $\alpha = \alpha_0/8 = 0.003125$.

As can ben seen from Table 2.8 (left)

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

Table 2.8 (right) shows a similar analysis comparing FILO (long) with the other methods. In particular,

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

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

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

Table 2.9: Computations on small-sized $\mathbb{X}$ instances.
[1]computed by considering the range of single-thread rating of compatible CPUs.

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

Table 2.10: Computations on medium-sized $\mathbb{X}$ instances.
[1]computed by considering the range of single-thread rating of compatible CPUs.

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

Table 2.11: Computations on large-sized 𝕏 instances.
[1]computed by considering the range of single-thread rating of compatible CPUs. New best solutions: (X-n801-k40, 73313);(X-n856-k95, 88989)

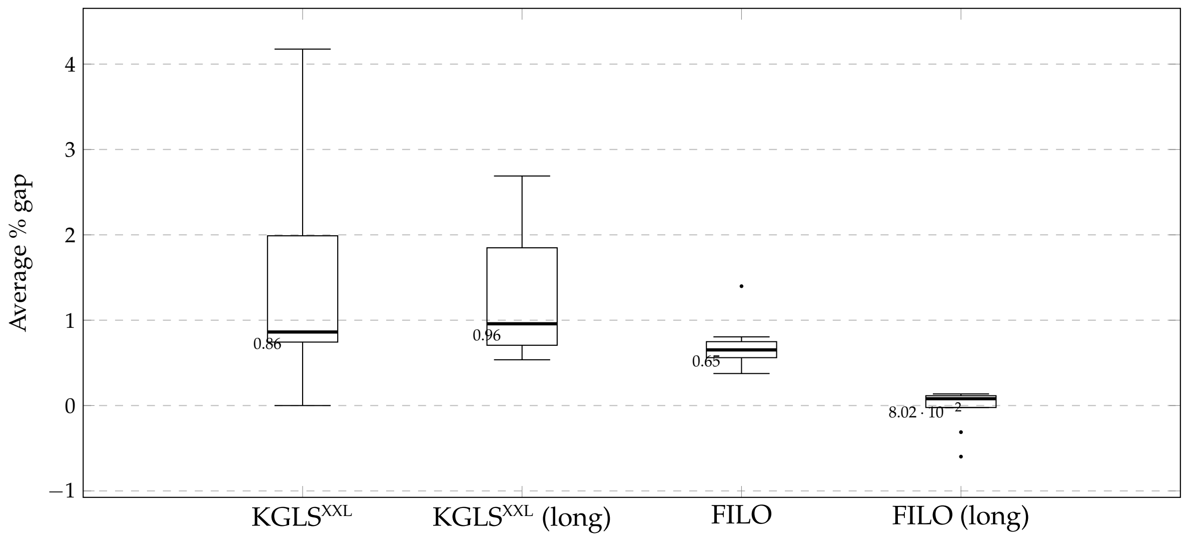


Figure 2.20: Comparison of average gaps obtained by algorithms on the $\mathbb{B}$ dataset.

D Computational details for very large-scale instances

This section contains computational details associated with large-scale datasets. In particular, Figures 2.20 – 2.22 show by means of boxplots the average gaps obtained by algorithms on the $\mathbb{B}$, $\mathbb{K}$, and $\mathbb{Z}$ dataset, respectively. Average solution values are analyzed by conducting analyses similar to those for the $\mathbb{X}$ dataset described in Section C. In particular, the null hypothesis H_0 and the alternative hypothesis H_1 are the same. However, contrarily to the previous analysis, we did not partition dataset instances in smaller groups. Thus the total number of analysis performed for each dataset is $n = 2$, one for each hypothesis. The initial confidence level $\alpha_0 = 0.025$ is thus adjusted through the Bonferroni correction to $\alpha = 0.025/2 = 0.0125$. Tables 2.12 and 2.13 show the p-values associated with the $\mathbb{B}$ and $\mathbb{K}$ datasets, respectively. Finally, due to the very limited number of instances of the $\mathbb{Z}$ dataset, the Wilcoxon signed-rank test cannot be used because it cannot give a significant result.

As can be seen from Table 2.12

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

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

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

Table 2.12: Computations on the $\mathbb{B}$ dataset: p-values for FILO on the left and for FILO (long) on the right.
p-values in bold are associated with rejected hypothesis when $\alpha = 0.0125$.
The last row contains a p-value interpretation when $\alpha = 0.0125$. In particular, FILO is not statistically different from the competing method when H_0 cannot be rejected (Similar), FILO is statistically better when both H_0 and H_1 are rejected (Better), and, finally, FILO is statistically worse when H_0 is rejected and H_1 is not rejected (Worse).

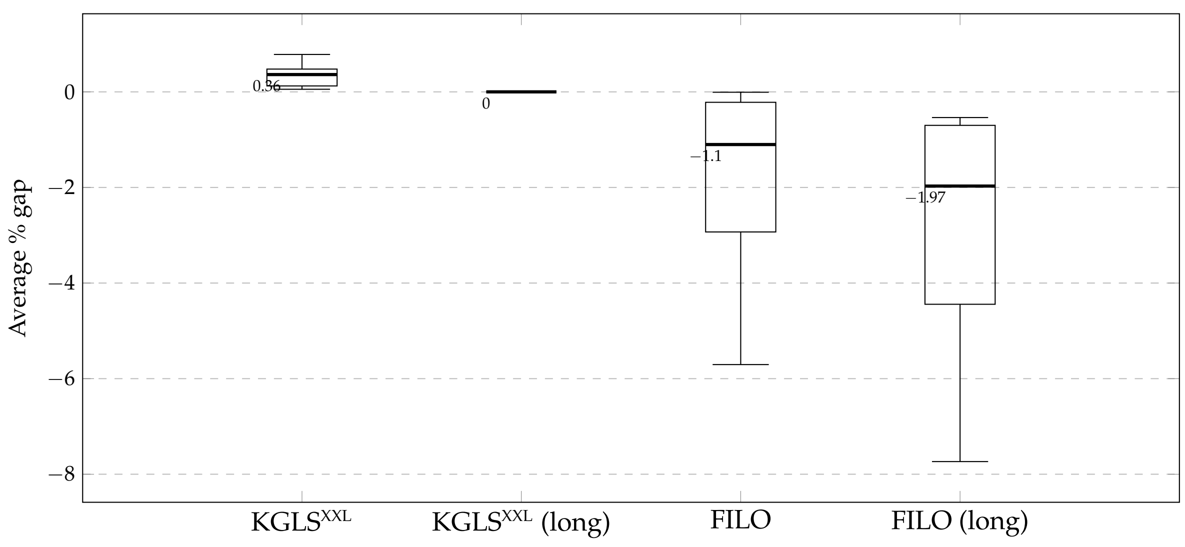


Figure 2.21: Comparison of average gaps obtained by algorithms on the $\mathbb{K}$ dataset.

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

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

Table 2.13: Computations on the $\mathbb{K}$ dataset: p-values for FILO on the left and for FILO (long) on the right.
p-values in bold are associated with rejected hypothesis when $\alpha = 0.0125$.
The last row contains a p-value interpretation when $\alpha = 0.0125$. In particular, FILO is not statistically different from the competing method when H_0 cannot be rejected (Similar), FILO is statistically better when both H_0 and H_1 are rejected (Better), and, finally, FILO is statistically worse when H_0 is rejected and H_1 is not rejected (Worse).

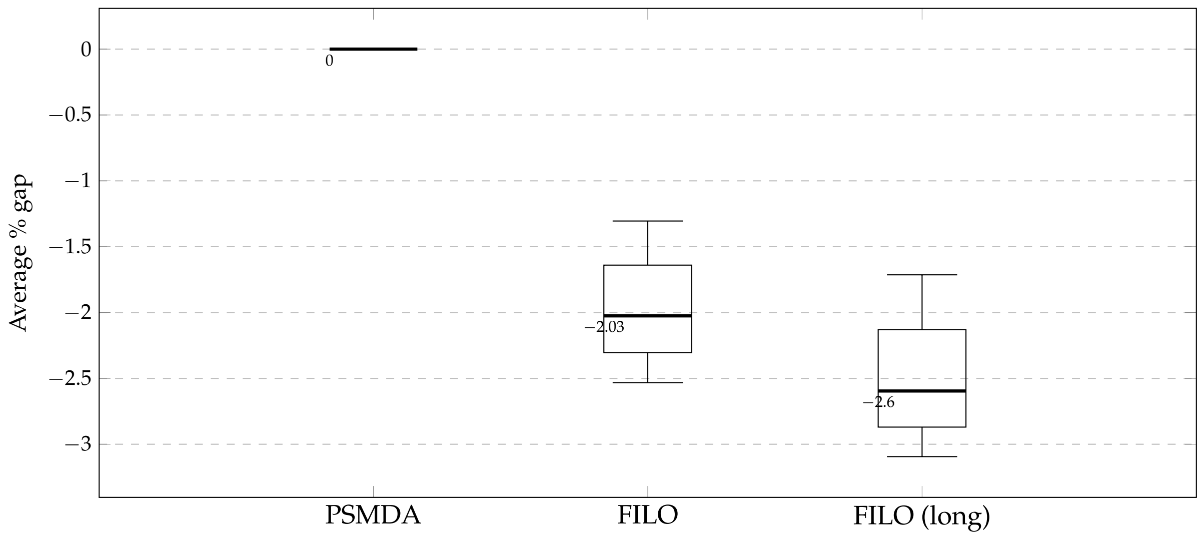


Figure 2.22: Comparison of average gaps obtained by algorithms on the $\mathbb{Z}$ dataset.

As can be seen from Table 2.13, both FILO and FILO (long) performs better than KGLS$^{\text{XXL}}$ and KGLS$^{\text{XXL}}$ (long).

Chapter 3

A Hybrid Metaheuristic for Single Truck and Trailer Routing Problems[1]

In this chapter, we propose a general solution approach for a broad class of vehicle routing problems that all use a single vehicle, composed of a truck and a detachable trailer, to serve a set of customers with known demand and accessibility constraints. A more general problem, called the Extended Single Truck and Trailer Routing Problem (XSTTRP), is used as a common baseline to describe and model this class of problems. In particular, the XSTTRP contains, all together, a variety of vertex types previously only considered separately: truck customers, vehicle customers with and without parking facilities, and parking-only locations. To solve XSTTRP we developed a fast and effective hybrid metaheuristic, consisting of an iterative core part, in which routes that define high-quality solutions are stored in a pool. Eventually, a set-partitioning based post-optimization selects the best combination of routes that forms a feasible solution from the pool. The algorithm is tested on extensively studied literature problems such as the Multiple Depot Vehicle Routing Problem, the Location Routing Problem, the Single Truck and Trailer Routing Problem with Satellite Depots, and the Single Truck and Trailer Routing Problem. Finally, computational results and a thorough analysis of the main algorithm's components on newly designed XSTTRP instances are provided. The obtained results show that the proposed hybrid metaheuristic is highly competitive with previous approaches designed to solve specific specialized problems, both in terms of computing time and solution quality.

1 Introduction

Truck and trailer routing problems constitute a very well-studied class of vehicle routing problems (VRPs) in which vehicle capacities may be augmented with trailers. This class of problems was introduced by Chao (2002) as an extension of the basic VRPs to better model real-world applications arising mainly in freight distribution and city logistics. The literature contains several variations on the basic settings for these problems that add specific constraints modeling specific scenarios. In Section 2 we give a brief overview of the literature, and we refer the interested reader to Cuda, Guastaroba, and Speranza (2015) for a general survey of related VRPs, including truck and trailer problems.

In this chapter, we study the Extended Single Truck and Trailer Routing Problem (XSTTRP) as a more general variant which allows us to derive a unified solution approach for a class of single truck and trailer routing problems. In particular, the XSTTRP contains, all together, a variety of vertex types previously only considered separately, namely: truck customers, vehicle customers with and without parking facilities, and parking-only locations. The XSTTRP calls for servicing

[1]The results of this chapter appears in: L. Accorsi and D. Vigo, "A Hybrid Metaheuristic for Single Truck and Trailer Routing Problems", *Transportation Science*, 2020, 54:5, 1351-1371

a set of customers with known demand using a single vehicle, composed of a capacitated truck and a non-autonomous, detachable trailer, which is initially located at a (main) depot. In the following, the term “truck" refers to the vehicle when the trailer is detached and parked at some parking location, while “complete vehicle" denotes the vehicle when the trailer is attached to the truck. The customers are partitioned into two different sets: *truck customers* and *vehicle customers*. The *accessibility constraints* impose that truck customers must be visited by the truck only, while vehicle customers can be visited either by the complete vehicle or by the truck. Vehicle customers are, in turn, split into vehicle customers *with parking facilities* and vehicle customers *without parking facilities*. The problem also contains a set of *satellite depots* (or just satellites), i.e., locations (which are not customers) where the trailer may be parked whenever necessary.

An XSTTRP solution is made up of a main route, in the following referred to as *main-route*, traveled by the complete vehicle, which starts from the main depot, visits a subset of vehicle customers and satellites, and returns to the depot. When the vehicle visits a *parking location* (i.e., either a satellite depot or a vehicle customer with parking facilities) it can detach its trailer, serve a subset of customers with the truck, and then return to pick up the trailer. We call this a *sub-route*, and the place where the trailer has been decoupled is the *root* of the sub-route. The objective is to find a solution which serves all customers while minimizing the total traveling cost and respecting both the truck capacity along the sub-routes and the accessibility constraints.

We developed a comprehensive, yet effective, heuristic solution approach to the XSTTRP. The resulting metaheuristic has been extensively tested on many instances, including special cases involving well-known problems such as the Multiple Depot Vehicle Routing Problem (MDVRP; see e.g., Cordeau, Gendreau, and Laporte (1997)), the Location Routing Problem (LRP; see Schneider and Drexl (2017)), the Single Truck and Trailer Routing Problem with Satellite Depots (STTRPSD; see Villegas et al. (2010)) and the Single Truck and Trailer Routing Problem (STTRP; see Bartolini and Schneider (2018)). The results show that the proposed method is highly successful in tackling the studied problems, producing good quality solutions in short computing time. The XSTTRP was originally introduced in a preliminary work by Accorsi (2017).

The chapter is structured as follows. In Section 2 we review the literature relating to the XSTTRP in more depth and discuss related problems. In Section 3 we describe the XSTTRP more comprehensively and compare it with existing related problems. Section 4 describes the details of our solution approach, and experimental results are provided in Section 5. Section 6 offers an experimental analysis of the algorithm components. Finally, the chapter ends with possible future research directions and concluding remarks in Section 7.

2 Literature review

The first comprehensive reference for the class of truck and trailer problems is the work by Chao (2002) who introduced the original Truck and Trailer Routing Problem (TTRP). Some earlier papers, such as that written by Semet and Taillard (1993), introduced similar variants which will be examined later in this section. The TTRP identifies a class of vehicle routing problems in which a fleet of capacitated vehicles, each composed of a truck and possibly a trailer, is used to serve a set of customers with a known demand. The objective is to minimize the total traveling cost without violating the capacity and accessibility constraints. The customers are split into *truck customers* and *vehicle customers*. Truck customers can only be visited by the truck without the trailer, so prior to visiting them, the vehicle has to park its trailer at an appropriate vehicle customer. The truck must then return to the same customer to pick up the trailer before continuing the journey. Vehicle customers can instead be served either by the truck or by the complete vehicle. Chao (2002) associated three possible types of route with each vehicle: a *pure truck route* which starts from the depot with the truck only, visits a subset of customers and returns to the depot; a *pure vehicle route* which starts from the depot with the complete vehicle, visits a subset of vehicle customers and returns to the depot; and a *complete vehicle route* consisting of a pure vehicle route and a set of pure truck routes starting from customer locations where the trailer could be parked. A feasible solution for the TTRP is thus composed of a set of routes, at most one for each vehicle, satisfying capacity constraints and belonging to one of the three types.

The current state-of-the-art algorithm for the TTRP is a matheuristic developed by Villegas et al. (2013). They populated a pool of high-quality solutions using a GRASP $\times$ ILS method (a greedy randomized adaptive search procedure combined with an iterated local search), and then they selected the best combination of routes from the pool by solving a set-partitioning formulation. The core of their approach was the construction of a giant tour that used a randomized nearest-neighbor heuristic to visit all the customers; the tour was further optimized by the interleaved application of a variable neighborhood descent (VND) and a perturbation phase. Each local optimum of the VND phase is stored in the set-partitioning solutions pool that is used to provide the final solution. The authors dealt with the standard TTRP as well as the Relaxed TTRP (see Lin, Yu, and Chou (2010)) which has an unlimited fleet. A previous method based on a GRASP with evolutionary path relinking was presented by Villegas et al. (2011).

Villegas et al. (2010) introduced the Single Truck and Trailer Routing Problem with Satellite Depots (STTRPSD). In this problem, a set of truck customers is served by a single capacitated vehicle composed of a truck and a trailer. The problem does not contain any vehicle customer; instead, it contains a set of parking locations called *satellite depots* where the trailer can be parked. Satellites do not have an associated demand and thus can be left unvisited if not needed. The authors did not consider the depot to be a parking location.

The presence of satellites introduces an interesting variation to the problem, relating it to the well-known class of Location Routing Problems (LRPs). For a comprehensive survey of LRPs refer to Schneider and Drexl (2017). In fact, the STTRPSD generalizes the 2-echelon LRP with zero opening costs, uncapacitated depots and capacitated vehicles (see e.g., Tuzun and Burke (1999)). The authors proposed four heuristic methods for solving the STTRPSD, which they also tested on the MDVRP (a special case of the LRP). The best-performing of the four was a multi-start evolutionary local search. Recently, Belenguer et al. (2016) proposed an exact branch-and-cut algorithm for the STTRPSD, based on an arc-flow formulation, which was able to consistently handle instances with up to sixty vertices. A paper that is important for its work towards unifying the TTRP variants is the Generalized Truck and Trailer Routing Problem (GTTRP) and its extension, the Vehicle Routing Problem with Trailers and Transshipment (VRPTT) proposed by Drexl (2011) and Drexl (2014). In the GTTRP, vehicles can leave the trailer either at vehicle customers or at transshipment locations (similar to satellites in STTRPSD); both may have associated time windows. The VRPTT extends the GTTRP by dropping the fixed assignment of a truck to a trailer; that is, any trailer may be pulled by any compatible truck for all or part of its itinerary. Moreover, the VRPTT adds the possibility that trucks may transfer, all or part of their load to any trailer at any transshipment location (with load-dependent transfer times). Note that, different vehicles may have different utilization (fixed and distance-dependent) costs. Drexl (2011) addressed the GTTRP with a branch-and-price algorithm and several heuristic variants, whereas in a different paper Drexl (2014) solved the VRPTT using five different branch-and-cut algorithms. As previously mentioned, real-world TTRP-like applications were documented long before the actual problem was defined by Chao (2002). For example, in the Site-Dependent Vehicle Routing Problem considered by Semet and Taillard (1993), a fleet of heterogeneous vehicles composed of trucks and trailers serves a number of

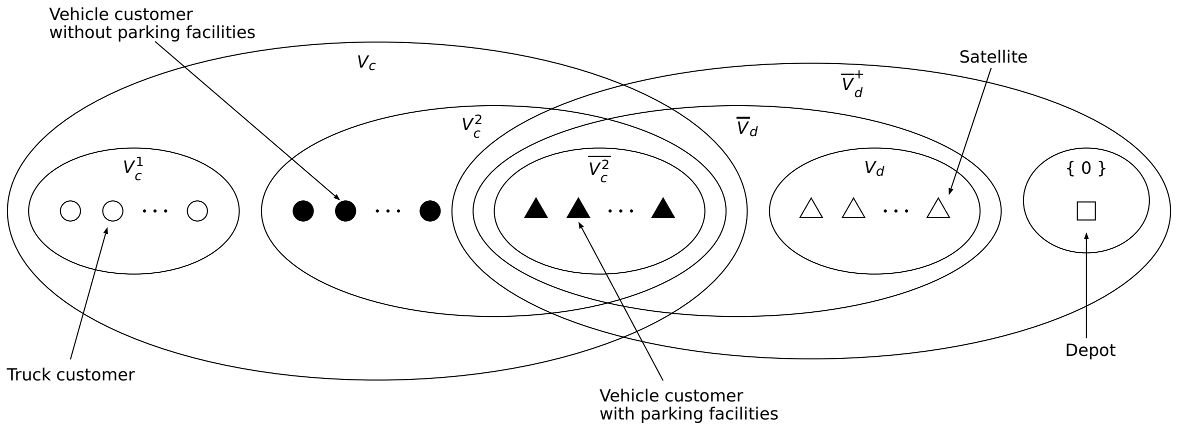


Figure 3.1: Relations between vertex types.

grocery stores in Switzerland cantons, taking into account vehicle-location incompatibilities, time windows for deliveries and vehicle-dependent utilization costs. The Partial Accessibility Constraint Vehicle Routing Problem analyzed in Semet (1995) consists of defining an appropriate number of trailers to increase the capacity of some trucks. Gerdessen (1996) considered the Vehicle Routing Problem with Trailers, which has been used to model two real-world scenarios: a Dutch dairy industry that has to serve customers located in crowded cities, where maneuvering a complete vehicle is difficult or forbidden; and the distribution of animal feed among farmers, some of whom are only reachable traversing narrow roads or small bridges. In both cases the trailer is often parked in appropriate locations before the truck performs the service. Further real-world TTRP applications include milk collection from farms (see Caramia and Guerriero (2010)), fuel oil delivery to private households (see Drexl (2011)), postal delivery (see Bodin and Levy (2000)) and container movement (see Tan, Chew, and Lee (2006)).

3 Problem description

The XSTTRP can be formulated by using an undirected graph $G = (V, E)$, where V is the vertex set and E is the edge set. The vertex set V is partitioned into $V = \{0\} \cup V_c \cup V_d$, where 0 is the depot, V_c is the set of customers and V_d is the set of satellite depots. V_c is, in turn, decomposed into $V_c = V_c^1 \cup V_c^2$, where V_c^1 represents the truck customers and V_c^2 the vehicle customers. Finally, we assume that a subset $\overline{V_c^2} \subseteq V_c^2$ of vehicle customers has parking facilities. From now on, $\overline{V}_d = V_d \cup \overline{V_c^2}$ denotes the parking locations where the vehicle is allowed to detach its trailer, namely the satellite depots and the vehicle customers with parking facilities, and $\overline{V}_d^+ = \overline{V}_d \cup \{0\}$. Figure 3.1 provides a graphic summary of the vertex sets and their relations. 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. Each customer $i \in V_c$ requires an integer quantity $q_i > 0$ from the depot and we assume that $q_i = 0, i \in \{0\} \cup V_d$. A vehicle with capacity $Q = Q_1 + Q_2$ is located at the depot, where Q_1 is the truck capacity and Q_2 is the trailer capacity. We assume that the vehicle is able to serve all the customers; that is, $\sum_{i \in V_c} q_i \leq Q$. We further suppose that, at parking locations, the goods required to satisfy the demand of the subsequent sub-routes are instantaneously transferred from the trailer to the truck. Therefore, all customers can be served by just one vehicle. Truck customers $i \in V_c^1$ can be visited by the truck only, i.e., without the trailer, while vehicle customers $i \in V_c^2$ can be visited either by the truck or by the complete vehicle. A *feasible solution* requires that the trailer is detached (either at an appropriate satellite depot $k \in V_d$ or at a vehicle customer with parking facilities $j \in \overline{V_c^2}$) before visiting any truck customer $i \in V_c^1$ and, possibly, some vehicle customer $i \in V_c^2$. Then, the truck returns to the

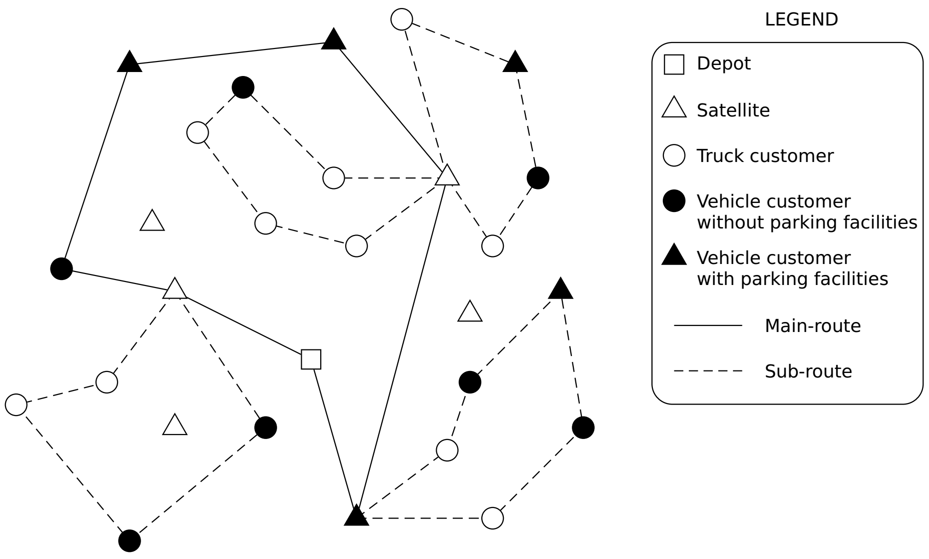

LEGEND
Depot
Satellite
Truck customer
Vehicle customer without parking facilities
Vehicle customer with parking facilities
Main-route
Sub-route

Figure 3.2: An example of a feasible solution to an XSTTRP instance.

parking location where the trailer has been detached, to pick the trailer up before moving to the next vehicle customer, satellite, or even to the main depot. In other words, reminding that a Hamiltonian circuit is a closed cycle that visits a set of vertices exactly once, a feasible XSTTRP solution is composed of:

- a Hamiltonian circuit, called *main-route*, that starts from the depot, visits a subset of vehicle customers $i \in V_c^2$ and satellites $k \in V_d$ and eventually returns to the depot;
- a number of Hamiltonian circuits, called *sub-routes*, each of which starts from a parking location $k \in \overline{V}_d$ visited by the main-route, visits one or more customers $i \in V_c$ and ends at the starting parking location k. It is allowed to have more than one sub-route rooted at the same parking location.

Each customer must be visited exactly once, while satellite depots may remain unvisited if not necessary. Moreover, in order to be compatible with the STTRPSD definition, sub-routes directly starting from the depot are not allowed. This limit can be easily overcome by creating a satellite coincident with the main depot location. Figure 3.2 shows a possible solution for an XSTTRP instance where the empty square denotes the main depot, the empty triangles represent satellite depots, the empty circles denote truck customers, the full circles represent vehicle customers without parking facilities and the full triangles are vehicle customers with parking facilities. The main-route is defined by a continuous line and the sub-routes by dashed lines.

3.1 Comparative analysis of related problems

The XSTTRP is inspired by the combination of STTRPSD and STTRP. Both problems are direct specializations of the XSTTRP, and as such include just a subset of its vertex types. However, the XSTTRP generalizes many other vehicle routing problem variants. To illustrate this, we focus on the MDVRP (with capacitated vehicles) and the LRP (with uncapacitated depots and capacitated vehicles), because both of these problems require two levels of decisions (assignment of customers to depots and routing) so they naturally fit into the XSTTRP model. We note that the Capacitated VRP (CVRP), being a special case of the MDVRP, is also generalized by the XSTTRP, which is therefore NP-Hard in the strong sense. The MDVRP, LRP and CVRP could be directly encoded as XSTTRP by: *(i)* mapping depots and customers to satellites and truck customers respectively, *(ii)* adding an additional dummy main depot, *(iii)* setting to 0 all costs

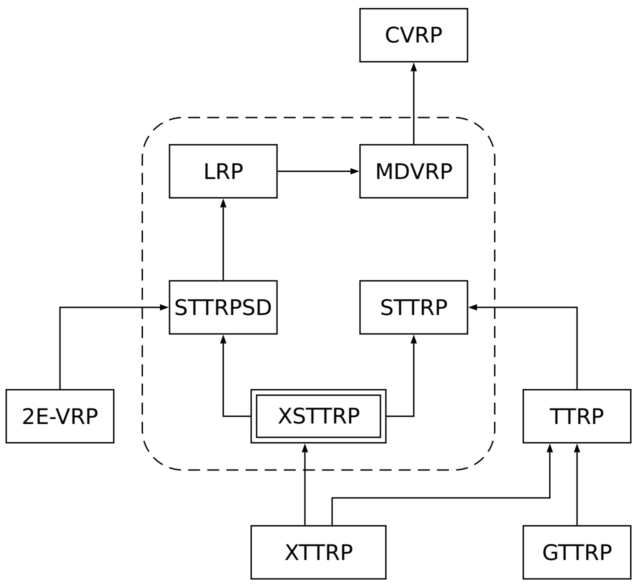


Figure 3.3: A possible topology (for a subset) of VRPs.
The relation $X \to Y$ means X is a generalization of Y. The problem variants studied in this chapter are those enclosed by the dashed line.

associated with the edges between satellites and those between satellites and depot, and *(iv)* setting the truck capacity equal to the original problem vehicle capacity and the trailer capacity equal to, at least, the sum of customer demands. The MDVRP usually imposes a maximum on the number of vehicles available in each depot, while the LRP defines a fixed cost associated with each vehicle and depot used. Both features are not directly modeled by the XSTTRP; thus they must either be handled by the solution procedure or indirectly encoded in the instance definition, e.g., by changing the cost of some arcs.

Figure 3.3 shows a possible hierarchical structure where several related problem variants are linked together by generalization/specialization relationships. From the figure we deduce that the natural generalization of the XSTTRP is the Extended TTRP (XTTRP); several, possibly heterogeneous, capacitated vehicles (possibly equipped with a trailer) are available to serve the customers. We also note that the STTRPSD is in fact a special case of the 2 Echelon-VRP (2E-VRP). In the latter, satellites are capacitated and might be visited multiple times by different vehicles. Finally, the GTTRP defined by Drexl (2011) is a complex and interesting unified model for vehicle routing problems with trailers. It generalizes the TTRP by including transshipment locations and several complex side-constraints. The XSTTRP and the XTTRP are much more basic problems, including just capacity and accessibility constraints. Moreover, the XSTTRP differs structurally from the GTTRP because it includes vehicle customers without parking facilities. Table 3.1 classifies the previously introduced problems according to the vertex types and the available number of vehicles they contain. In our computational testing we focused on the variants, enclosed in the dashed line of Figure 3.3, that preserve the multiple-depot structure of XSTTRP in which satellites are present.

4 Solution approach

In this section we describe our metaheuristic for the XSTTRP, resulting in the algorithm AVXS. The algorithm consists of a core part followed by a post-optimization phase. The core part iterates through the following three *phases*:

	Depot	Satellite	Vehicle customer		Truck customer	
			with p. f.	without p. f.		
Problem	□	△	▲	●	○	**#veh**
CVRP	✓	✗	✗	✓	✗	≥ 1
MDVRP	✓	✗	✗	✓	✗	≥ 1
LRP	✓	✗	✗	✓	✗	≥ 1
2E-VRP	✓	✗	✓	✗	✓	≥ 1
STTRPSD	✓	✓	✗	✗	✓	1
STTRP	✓	✗	✓	✗	✓	1
TTRP	✓	✗	✓	✗	✓	≥ 1
XSTTRP	✓	✓	✓	✓	✓	1
XTTRP	✓	✓	✓	✓	✓	≥ 1
GTTRP	✓	✓	✓	✗	✓	≥ 1

Table 3.1: A problem classification based on the vertex types and number of vehicles **#veh**.
The symbol ✓ means that the problem contains the vertex type and ✗ that it does not. The abbreviation p. f. stands for parking facilities.

1. The *assignment* phase associates each vertex $i \in V$ with an appropriate vertex $j \in V$, e.g., customers are assigned to parking locations or to the depot;
2. The *construction* phase builds an initial routing solution which is complete and feasible, starting from the given assignment;
3. The *improvement* phase may further optimize the given solution.

These phases are executed for a fixed number of iterations, called *restarts*, during which a limited set of routes composing high-quality solutions are stored in a pool $\mathcal{P}$. The set-partitioning model F, described in Section 4.4, is then populated with the routes from $\mathcal{P}$. A concluding post-optimization *polishing* phase then selects the best combination of routes as the final solution. Pseudo-code for AVXS is given in Algorithm 8, while the following paragraphs provide a detailed description of the four phases. Finally, the section ends with a discussion on how to handle some special requirements arising in problem variants within AVXS, such as the MDVRP and the LRP.

Algorithm 8 AVXS algorithm

```
1:  procedure AVXS($instance, seed$)
2:      $\mathcal{R} \leftarrow$ RANDOMENGINE($seed$)        ▷ Initialize the pseudo-random number generator
3:      $\mathcal{P} \leftarrow \emptyset$        ▷ Initialize the pool that will eventually contain high-quality routes
4:      $S^* \leftarrow$ EMPTYSOLUTION($instance$)        ▷ Initialize the best known solution
5:      for $r \leftarrow 1$ to $\Delta$ do
6:          $S \leftarrow$ EMPTYSOLUTION($instance$)        ▷ Initialize the current restart best solution
7:          $\mathcal{A} \leftarrow$ ASSIGNMENT($instance, \mathcal{R}$)
8:          $S \leftarrow$ CONSTRUCTION($S, \mathcal{A}$)
9:          $S \leftarrow$ IMPROVEMENT($S, \mathcal{P}, \mathcal{R}$)
10:         if COST($S$) $<$ COST($S^*$) then $S^* \leftarrow S$        ▷ Update the best known solution
11:     end for
12:     $F \leftarrow$ BUILDMODEL($\mathcal{P}$)        ▷ Populate the set-partitioning model $F$ using the routes in $\mathcal{P}$
13:     $S^* \leftarrow$ POLISHING($S^*, F$)
14:     return $S^*$
15: end procedure
```

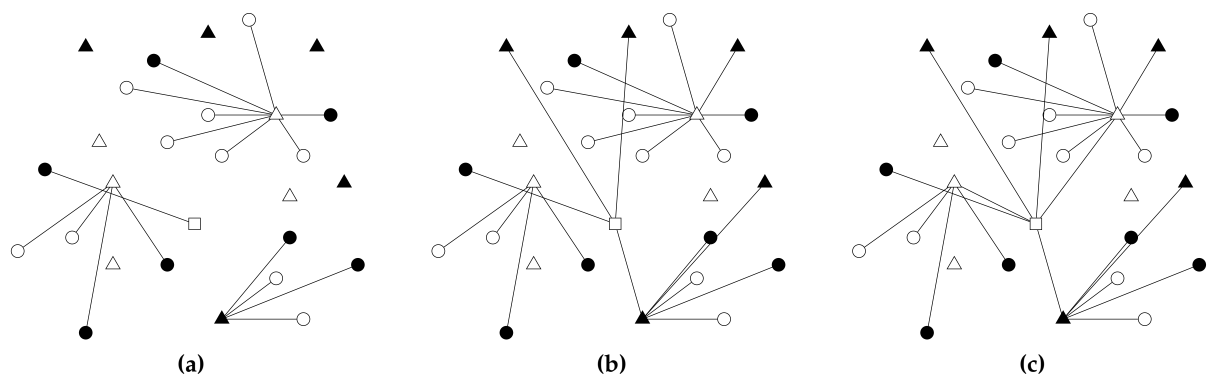


Figure 3.4: Construction of an assignment tree starting from the instance depicted in Figure 3.2. Each separate picture represents the result of each step of the assignment phase.

4.1 Assignment

In this phase we gradually build a feasible *assignment tree* by iteratively assigning customers to parking locations and the depot. To this end we define the *assignment cost* $\hat{c}_{ij}$ as the estimated cost of serving vertex $i \in V$ from vertex $j \in V$. In our implementation, we define the assignment costs as

$$\hat{c}_{ij} = \begin{cases} c_{ij} & (i \in V_c^1 \wedge j \in \overline{V}_d) \vee (i \in V_c^2 \wedge j \in \overline{V}_d^+ \setminus \{i\}) \vee (i \in V_d \wedge j = 0) \\ +\infty & \text{otherwise} \end{cases}$$

As an alternative, one could use a more sophisticated estimate, such as the reduced costs obtained by a combinatorial optimization bound (see e.g., Toth and Vigo (2002)). However, our preliminary computational experiments showed that the additional effort of using reduced costs for the assignments did not provide substantial improvement compared to the original costs.

We define a parking location $k \in \overline{V}_d$ as *open* if it has at least one customer assigned to it, and *closed* otherwise. The assignment phase performs the following steps in sequence:

(a) Truck customers and vehicle customers without parking facilities $i \in V_c^1 \cup (V_c^2 \setminus \overline{V_c^2})$ are probabilistically assigned to vertices $k_i \in \overline{V}_d^+$ through a roulette-wheel selection using an assignment fitness function f^i. Once the step is completed some vehicle customers with parking facilities will be open and others will be closed. See Figure 3.4(a) for an illustration.

(b) Closed vehicle customers with parking facilities $i \in \overline{V_c^2}$ are assigned to open parking locations and the depot $j \in \overline{V}_d \cup \{0\}$, using the same strategy as in step (a). Open vehicle customers with parking facilities $k \in \overline{V_c^2}$ are assigned to the depot, as depicted in Figure 3.4(b).

(c) Open satellites $k \in V_d$ are assigned to the depot and the closed ones are ignored, as in Figure 3.4(c).

Observe that the assignment resulting from this phase will always respect the accessibility constraints defined by the XSTTRP whenever the instance is feasible. The definition of the assignment fitness function f^i for each vertex $i \in V$ obviously influences the outcome of this phase. In our implementation, we used the d-NEAR assignment fitness function with d equal to 25. The

d-NEAR is a restricted GRASP-based function defined as

$$f^i(j) = \begin{cases} 1 & j \in \mathcal{H}_i^d(\hat{\boldsymbol{c}}) \\ 0 & \text{otherwise} \end{cases}$$

where $\mathcal{H}_i^d(\hat{\boldsymbol{c}})$ denotes the d nearest vertices to i according to the assignment costs $\hat{\boldsymbol{c}}$. For a comprehensive analysis of the behavior of the d-NEAR function we refer to Section 6.1. Finally, the probability p_{ij} of assigning vertex $i \in V$ to vertex $j \in V$ is defined as $p_{ij} = f^i(j) / \sum_{k \in V} f^i(k)$. It is worth noting that, due to the roulette-wheel selection procedure, subsequent calls of the assignment phase might produce different assignment trees.

4.2 Construction

Given an assignment tree, an initial feasible solution to XSTTRP is defined as follows. First, the main-route is determined by solving a Traveling Salesman Problem (TSP) visiting the depot, the open parking locations, and any vehicle customer without parking facilities assigned to the depot. Then, for each open parking location $k \in \overline{V}_d$, the sub-routes are obtained by solving a specific CVRP with the customers belonging to the sub-tree rooted in k. Note that there is no limit on the number of sub-routes rooted in a given parking location. As extensively discussed in Section 6.1, the quality of the initial solution is not crucial for the algorithm's overall performance. Therefore, we solved both the CVRP and the TSP using an adaptation of the well-known savings algorithm by Clarke and Wright (1964b).

4.3 Improvement

The initial solution defined in the construction phase is improved using a specific procedure based on the Iterated Local Search paradigm (ILS, see Lourenço, Martin, and Stützle (2003)) in which the local search is performed using a Randomized Variable Neighborhood Descent (RVND, see e.g., Subramanian, Uchoa, and Ochi (2013)). In addition, to speed up the local search execution, we adopted a granular approach (see, Toth and Vigo (2003) and Schneider, Schwahn, and Vigo (2017)). Our ILS intensifies the search around the current best-known solution. The diversification is naturally provided throughout the procedure itself, strengthened by a global restart mechanism which provides a different starting assignment tree. The randomization in the RVND is used for selecting the order of the neighborhoods and that of the moves within each neighborhood, the latter are searched according to a first-improvement strategy. This improves the likelihood that different search pathways are explored, thus diversifying while improving the solution. We considered the following neighborhoods:

- 1-0 exchange (RELOCATE) in which each customer is removed from its current position and re-inserted in the position that minimizes the insertion cost in the current solution;
- 1-1 exchange (SWAP) in which a pair of customers is exchanged;
- intra-route & intra-park 2-opt (TWOPT) in which an intra-route 2-opt procedure is applied to each route and an inter-route 2-opt procedure is applied to each VRP, defined by an open parking location and the vertices assigned to it;
- sub-routes segments-swap (SEGSWAP) in which contiguous, possible empty, paths within sub-routes including from two up to five vertices are swapped. SEGSWAP is a simple adaptation of the CROSS-exchange operator introduced by Taillard et al. (1997b). Note that, for efficiency purposes, our definition of SEGSWAP works on already open satellites and sub-routes only. Therefore, it does not include RELOCATE or SWAP moves that can

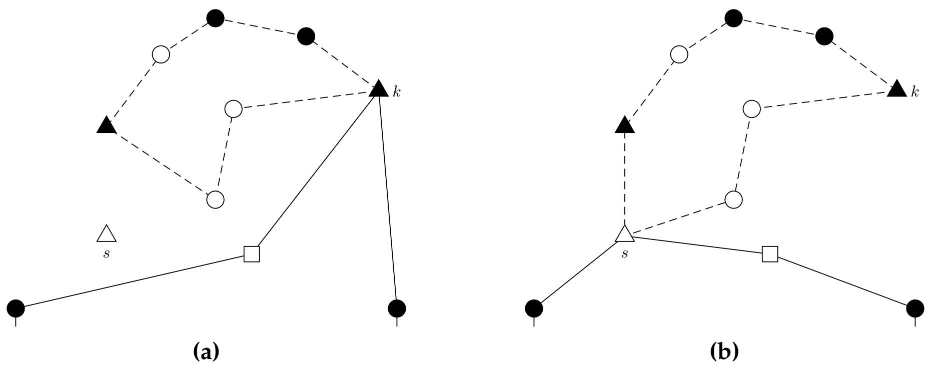


Figure 3.5: A special case of the ROOTREF move.
In (a) we show the initial solution with the sub-route rooted in vertex k. In (b) the root is changed to s, and k is removed from the main-route and inserted into the sub-route.

work on main- and sub-routes. In fact, from our experience, swapping a contiguous path of vertices between the main-route and a sub-route seldom results in a feasible exchange;

- sub-route root-refine (ROOTREF) (see Chao (2002)), consists of replacing the root of a sub-route with another one. We extend the operator by also considering the possibility of leaving the original root, corresponding to a vehicle customer with parking facilities, in the sub-route (as shown in Figure 3.5).

At this point, we have five local search operators, all but one of which, the TWOPT, can change the initial assignment tree. As will be analyzed in Section 6.1, the local search step is capable of greatly improving the quality of the solutions. In the RVND we examine each granular neighborhood exhaustively before moving to the next one. If an improving move is found, we apply the move and restart the search using the current neighborhood. The local search terminates once a complete RVND is executed without finding an improving solution.

Once the local search is stopped at a local optimum, we apply a *shake* procedure to the solution in a *ruin-and-recreate* style (see Schrimpf et al. (2000b)). The shake is always applied to the best known solution found during the current restart. More precisely, we designed three different removal procedures to be used in the *ruin* step. The first one, called *main-vertex-removal* (MREM), operates by removing, with probability η, each vehicle customer served by the main-route that is not an open parking location (as discussed in Section 5.2, we used $\eta = 0.5$). The other removal procedures work on sub-routes. The *sub-routes-removal* procedure (SREM) aims to improve the assignment by destroying a number of long routes from a set of randomly chosen, open parking locations. For each selected parking location k, we remove the r longest sub-routes, where r is randomly chosen between 0 and the total number of sub-routes rooted in k. Note that if the parking location is a satellite, then all the sub-routes rooted in that satellite are removed and the satellite is closed. We define this special case as SREM.1. On the other hand, the *sub-routes unloading* procedure (SUNLOAD), which is intended to improve the routing solution, consists of removing random customers from each sub-route until its load is less than a threshold $\zeta \times load_{avg}$, where ζ is a coefficient that is uniformly randomly generated on the interval $[0.3, 0.9]$ (at each shake application) and $load_{avg}$ is the average load of all sub-routes. At each shake application we always execute MREM, and we alternate the executions of SREM and SUNLOAD. The *recreate* step inserts each removed customer in the position of the current solution that minimizes the insertion cost. The order in which customers are considered affects the final result. Therefore, at every recreate step we examine the customers according to an ordering randomly chosen among five possibilities: ascending or descending values of their x or

y coordinates, respectively, or a random permutation. Pseudo-code for the improvement phase is provided in Algorithm 9.

Algorithm 9 Improvement phase

```
 1: procedure IMPROVEMENT($S, \mathcal{P}, \mathcal{R}$)
 2:     $\pi \leftarrow \pi_{base}$                         ▷ Initialize the sparsification factor
 3:     $i \leftarrow 0, g \leftarrow 0$                     ▷ Initialize ILS and granular counters
 4:     $S' \leftarrow S$                                  ▷ Initialize the current restart best solution
 5:     while true do
 6:         $S \leftarrow$ RVND($S, \mathcal{R}$)          ▷ Improve the current solution using the randomized VND
 7:         if COST($S$) < COST($S'$) then               ▷ Update the current restart best solution, if necessary
 8:             $i \leftarrow 0, g \leftarrow 0, S' \leftarrow S, \pi \leftarrow \pi_{base}$   ▷ Reset some variables
 9:             $\mathcal{P} \leftarrow \mathcal{P} \cup$ ROUTESOF($S$)   ▷ Add routes to the route pool
10:         end if
11:         $i \leftarrow i + 1, g \leftarrow g + 1$         ▷ Increment ILS and granular counters
12:         if $i \geq \delta$ then break                ▷ Terminate the improvement phase
13:         if $g \geq \phi \cdot \delta$ then $\pi \leftarrow \lambda \cdot \pi, g \leftarrow 0$   ▷ Update the move generators set
14:         $S \leftarrow S'$
15:         $S \leftarrow$ SHAKE($S$)
16:     end while
17:     return $S'$
18: end procedure
```

Finally, as to the algorithm's efficiency, we observe that both RELOCATE and ROOTREF might evaluate moves which require the opening of closed satellites. To identify the insertion position for the satellite which minimizes the insertion cost in the main-route, we adopted a lazy caching strategy. In particular, we use an additional cache data structure that for each closed satellite stores its best insertion position. At the beginning, the cache is empty (or invalid). When a move needs to evaluate the opening of a closed satellite, we check the cache for a valid result. If the result is valid, we return the stored position, otherwise we scan the current main-route, find the current best insertion position, and update the corresponding cached value. The whole cache is invalidated each time the main-route is modified.

Granular neighborhoods.

Granular neighborhoods have been successfully applied to several Vehicle Routing Problems to speed up local search procedures while preserving solution quality compared to complete neighborhood exploration (see Toth and Vigo (2003)). Following the guidelines presented by Toth and Vigo (2003) and Schneider, Schwahn, and Vigo (2017) a granular neighborhood is completely defined by a set $T \in E$ of appropriately chosen arcs, called *move generators*. Each arc in T is used in the neighborhood search to generate a uniquely defined single move, thus reducing the search time to $O(|T|)$. Typically, the set T of move generators for a specific neighborhood is defined according to problem-related criteria, often called *sparsification rules*. For example, according to Toth and Vigo (2003), arcs are chosen if their cost is below a given threshold, whereas Schneider, Schwahn, and Vigo (2017) used the reduced cost coming from a simple relaxation for the same purpose. In our algorithm all the neighborhoods used in the RVND are defined as granular and we adopted a simple cost-based sparsification rule.

The multi-level nature of a problem with a main-route and several sub-routes calls for additional care in the definition of the set of move generators. More precisely, by examining Figure 3.6, which reports a sub-optimal solution for instance 25 of Villegas et al. (2010), one may easily observe that the arcs in the main-route are considerably longer than those of the sub-routes. Therefore, if we use a relatively small cost threshold to define the sparsification rule, very few arcs between satellites are likely to be inserted in T, possibly causing severe harm to the solution quality. As a consequence, we used two different sparsification rules for the

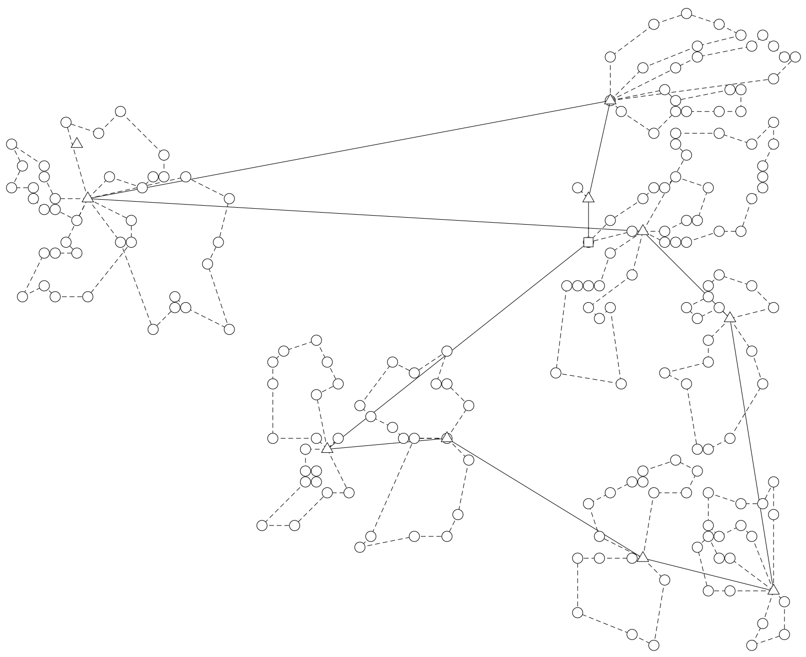


Figure 3.6: A solution for instance 25 by Villegas et al. (2010), a clustered instance with 200 truck customers and ten satellite depots. The main-route is clearly sub-optimal.

arcs of the main-route $\{(i,j) : i,j \in V_c^2 \cup V_d \cup \{0\}\}$, called *main-route arcs*, and for those of the sub-routes $\{(i,j) : i,j \in V_c \cup V_d\}$, called *sub-route arcs*. Note that some arcs, e.g., those between vehicle customers, may belong to both sets because such customers may be served either in the main-route or in the sub-routes. Given a sparsification factor π, we define T as the union of two sets T_M and T_S corresponding to main- and sub-route arcs. The set T_M includes the $\pi \cdot (|V_c^2| + |V_d| + 1)$ shortest main-route arcs, while the set T_S includes the $\pi \cdot (|V_c| + |V_d|)$ shortest sub-route arcs.

Finally, following Schneider, Schwahn, and Vigo (2017), we adopted a dynamic updating of the sparsification level. We initially set $\pi = \pi_{base}$, and whenever $\phi \cdot \delta$ non-improving iterations are performed, $\pi = \pi \cdot \lambda$. The value of π is reset to π_{base}, however, whenever an improving solution is found.

4.4 Polishing

We store every route that constitutes an improving solution produced by the RVND procedure in a pool $\mathcal{P}$ of high-quality routes (see line 9 of Algorithm 9). Note that we only store local minima which are improving with respect to the current restart best solution and not all the generated ones. We found that this allows us to limit the size of the pool without a significant worsening of the solution quality. In storing the routes we discard duplicates and, possibly, update the existing ones with improved versions. We effectively handle routes retrieval by using an additional hash-table data structure that, given a set of nodes, is able to efficiently check whether a route passing through the given set of nodes exists or not. Moreover, routes in $\mathcal{P}$ are stored as set of vertices with an associated cost label which depends on the sequence of visits of the vertices. Given two routes passing through the same set of vertices we only maintain the one with the lowest cost label. The final result of our algorithm is the best solution found by a set-partitioning post-optimization performed on the routes from $\mathcal{P}$.

More precisely, let $\mathcal{M}$ be the subset of all the main-routes in $\mathcal{P}$ starting from the depot 0 and passing through a subset of vehicle customers V_c^2 and, possibly, through one or more satellites in V_d. Let $\mathcal{M}_k \subset \mathcal{M}$ be the subset of main-routes passing through $k \in V_c^2 \cup V_d$. Note that $\mathcal{M}_k = \emptyset$ for each truck customer $k \in V_c^1$. Given a main-route $r \in \mathcal{M}$, we denote by g_r its cost and by R_r the subset of vehicle customers and satellite depots visited by r. Let $\mathcal{R}_k$ be the subset of all the sub-routes in $\mathcal{P}$ rooted in parking location $k \in \overline{V}_d$. We denote by $\mathcal{R}_{ki} \subset \mathcal{R}_k$ the subset of sub-routes rooted in $k \in \overline{V}_d$ that pass through customer $i \in V_c$. Let $\ell \in \mathcal{R}_k$ be a sub-route rooted in k; we denote by $d_{k\ell}$ its cost and by R_ℓ^k the subset of customers visited by ℓ and we assume that $k \notin R_\ell^k$. Let y_r be a binary variable that is equal to 1 if and only if the main-route $r \in \mathcal{M}$ is in the solution, and 0 otherwise. Furthermore, let $x_{k\ell}$ be a binary variable that is equal to 1 if and only if the sub-route $\ell \in \mathcal{R}_k, k \in \overline{V}_d$ is in the solution, and 0 otherwise. The XSTTRP route-based formulation F is defined as follows.

$$
\begin{aligned}
(F) \quad \min \quad & \sum_{k \in \overline{V}_d} \sum_{\ell \in \mathcal{R}_k} d_{k\ell} x_{k\ell} + \sum_{r \in \mathcal{M}} g_r y_r && && (1)\\
\text{s.t.} \quad & \sum_{k \in \overline{V}_d} \sum_{\ell \in \mathcal{R}_{ki}} x_{k\ell} = 1, && i \in V_c^1 && (2)\\
& \sum_{k \in \overline{V}_d} \sum_{\ell \in \mathcal{R}_{ki}} x_{k\ell} + \sum_{r \in \mathcal{M}_i} y_r = 1, && i \in V_c^2 && (3)\\
& \sum_{\ell \in \mathcal{R}_{ki}} x_{k\ell} - \sum_{r \in \mathcal{M}_k} y_r \le 0, && k \in \overline{V}_d, i \in V_c \setminus \{k\} && (4)\\
& \sum_{r \in \mathcal{M}} y_r = 1 && && (5)\\
& x_{k\ell} \in \{0,1\}, && \ell \in \mathcal{R}_k, k \in \overline{V}_d && (6)\\
& y_r \in \{0,1\}, && r \in \mathcal{M} && (7)
\end{aligned}
$$

where the objective function (1) imposes the selection of the best combination of main-route and sub-routes. Equations (2) and (3) require that each customer $i \in V_c$ be visited exactly once by a compatible route. Inequalities (4) specify that a customer $i \in V_c$ can be visited by a sub-route rooted in a parking location $k \in \overline{V}_d$ only if that location is visited by the main-route. Constraint (5) dictates that just one main-route must be in the solution. Finally, constraints (6) and (7) define the variables as binary. The mathematical formulation was introduced in a preliminary work by Accorsi (2017). The polishing phase brings together the otherwise unrelated restart results. Similar methods have proved effective in other vehicle routing algorithms (see e.g., the works by Subramanian, Uchoa, and Ochi (2013) and Villegas et al. (2013)). In doing the post-optimization, we supply the best found solution as a warm start; we stop the model resolution after a predetermined computer-independent number of ticks, see Section 5.1.

4.5 Extensions to handle specific sub-problems

The conversion from MDVRP and LRP instances to XSTTRP ones is straightforward, although both problems include special requirements not directly definable in XSTTRP terms. In particular, the MDVRP imposes a maximum on the number of vehicles available in each depot and the LRP defines a fixed cost on the use of vehicles and on the opening of depots. As discussed in Section 3.1, these problems are encoded as XSTTRP by: *(i)* mapping depots and customers to satellites and truck customers respectively, *(ii)* adding an additional dummy main depot, *(iii)* setting to 0 all costs associated with the edges between satellites and those between satellites and depot, and *(iv)* setting the truck capacity equal to the original problem vehicle capacity and the trailer capacity equal to, at least, the sum of customer demands.

In the MDVRP, we build a zero-cost main-route through the main depot and all the satellites (that is, we assume all satellites have been visited). Next, in order to handle a maximum number of available vehicles (say n_v) in each satellite, we simply introduce a penalty; the solution cost is incremented by a very large value for each additional vehicle used in the solution. Although this alone does not guarantee the discovery of feasible solutions, it has been proved effective experimentally in solving the instances we consider. Furthermore, this same constraint must be considered in model F by adding

$$\sum_{\ell \in \mathcal{R}_k} x_{k\ell} \leq n_v, \quad k \in V_d$$

Finally, we relax the shaking procedure SREM by discarding the special case SREM.1 defined in Section 4.3, which removes all routes rooted in a satellite chosen by the SREM shaking procedure. Since the main-route does not contribute to the objective function in the MDVRP, any empty visited satellite can remain open. Thus, instead of removing all sub-routes from a satellite, we remove a random number of them.

To handle the LRP, we simply incorporate the fixed costs of using a vehicle $\mathcal{F}_v$ and opening an LRP depot $\mathcal{F}_d$ into the cost matrix as follows.

$$c'_{ij} = \begin{cases} \frac{\mathcal{F}_d}{2} & (i \in V_d \wedge j = 0) \vee (i = 0 \wedge j \in V_d) \\ \mathcal{F}_d & i \in V_d \wedge j \in V_d \wedge i \neq j \\ \frac{\mathcal{F}_v}{2} & (i \in V_c \wedge j \in V_d) \vee (i \in V_d \wedge j \in V_c) \\ c_{ij} & \text{otherwise} \end{cases}$$

We use this new cost matrix $\boldsymbol{c}'$ to execute the optimization. It is important to note that, when opening a satellite has high fixed cost, we cannot rely on a probabilistic assignment phase, because a wrong initial assignment is very difficult to modify when the fixed cost is greater than the routing contribution. In other words, the attractiveness of already open satellites might be very high with respect to the opening of closed ones. In particular, our local search procedures do not contain any specialized technique which is able to completely move several routes from a satellite to another one in a single evaluation. The SREM ruin procedure might cause a satellite to be completely closed but, again, if opening a satellite has high fixed cost compared to the routing cost, the recreate step will prefer to insert the removed customers in routes of already open satellites instead of opening new ones. Hence, in this situation, a meaningful selection of initial satellites during the assignment phase can be very important. To handle this, we use the d-NEAR assignment fitness function with d equal to 1, which deterministically assigns a vertex $i \in V$ to its nearest neighbor $j \in V$ according to the assignment cost $\hat{\boldsymbol{c}}$.

5 Computational results

The computational testing had two objectives. First of all we wanted to assess the success of AVXS. To this end, we used the algorithm to solve instances of four special cases of the XSTTRP which are extensively studied in the literature, namely the MDVRP, the LRP, the STTRPSD and the STTRP. We also performed a preliminary testing on the CVRP instances of the X benchmark by Uchoa et al. (2017). Unfortunately, our results with the algorithm's parameters tuned as described in Section 5.2 were not state-of-the-art on those large scale instances, therefore we do not describe them in this section. The second objective was to analyze the impact of the algorithm's main components on its performance in detail, by testing AVXS on new instances of the XSTTRP which have not been previously studied in the literature.

5.1 Implementation and experimental environment

The proposed algorithm was implemented in C++ and compiled using g++ 6.3.0. To solve the set-partitioning model F we used CPLEX 12.8 (CPLEX (2017), C callable library) with default settings and a prefixed number of ticks as a computer-independent measure to prematurely terminate the model resolution. In particular, assuming our machine performs an average number of 1900 ticks per second, we assigned CPLEX a maximum of $60 \cdot 1900 = 114000$ ticks. The experiments were performed on a 64-bit desktop computer with an Intel Core i7-8700K CPU, running at 3.7 GHz with 32 GB of RAM on a GNU/Linux Debian operating system. Both the algorithm and CPLEX were executed using a single thread. All the instances used in our testing with the corresponding graphic solutions, as well as the source code, could be downloaded from `https://acco93.github.io/avxs/`. Detailed instructions are given as an aid to others wishing to accurately reproduce our results. Because our algorithm contains randomized elements, for every experiment we performed a set of ten runs for each instance and we defined the seed of the pseudo-random engine, the Mersenne Twister (Matsumoto and Nishimura (1998)), equal to the run counter minus one. To facilitate comparisons with other algorithms run on computers with different CPUs, we used the single-thread rating defined by PassMark ®Software (2020), which assigns a score of 2704 to our CPU. Moreover, to mitigate the impact of small time-variations due to overhead of the operating system we intentionally used a clock function that reports running times with a precision set to one second as the minimum recordable time. Therefore, when the algorithm took less than one second, we considered the elapsed time to be one. Finally, if not stated otherwise, we assume that the arc costs c_{ij} were computed as $c_{ij} = Euc(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$, where x_i, x_j, y_i and y_j represent the x and y coordinates of vertices $i, j \in V$, respectively.

5.2 Parameter definitions

Algorithm AVXS requires that initial values be defined for a number of parameters. We set these values during preliminary testing, performed on a training which included the XSTTRP instances with 200 total vertices. The resulting parameter values, which were used on all problem classes, are the following:

- number of restarts, $\Delta = 100$;
- maximum number of non-improving ILS iterations, $\delta = 100$;
- probability of removing a customer from the main-route in the MREM procedure, $\eta = 0.5$;
- unloading range used in the SUNLOAD procedure, $\zeta = [0.3, 0.9]$;
- initial sparsification value used to define the granular neighborhood, $\pi_{base} = 1.25$;
- factor, $\phi = 0.2$, used to define the fraction of non-improving iterations to be performed before incrementing the sparsification parameter π;
- factor, $\lambda = 2$, used to increment the value of π when $\phi \cdot \delta$ non-improving iterations are performed.

The parameters tuning followed a simple sequential strategy. At the beginning, we identified initial reasonable values for all the parameters by performing a preliminary limited trial-and-error testing. Then, we assessed the performance of the algorithm when changing the value of one parameter at a time. We kept a new value when it allowed good quality solutions while keeping the computational time low.

						HGSADC		AVXS				
Id	$\lvert V_c^1 \rvert$	$\lvert V_d \rvert$	Q_1	*Type*	*BKS*	*Avg*	t_{10}	*Best*	*Avg*	*Worst*	t_{10}	$\hat{t}_{10}$
p01	50	4	80	rd	**576.87**	0.00	138	0.00	0.00	0.00	10	39.47
p02	50	4	160	rd	**473.53**	0.00	126	0.00	0.00	0.00	10	39.47
p03	75	2	140	rd	641.19	0.00	258	0.00	0.00	0.00	20	78.95
p04	100	2	100	rd	1001.04	0.02	1164	0.00	0.15	0.31	178	702.65
p05	100	2	200	rd	750.03	0.00	636	0.00	0.00	0.00	56	221.06
p06	100	3	100	rd	**876.50**	0.00	684	0.00	0.00	0.00	75	296.06
p07	100	4	100	rd	**881.97**	0.28	930	0.00	0.06	0.30	86	339.48
p12	80	2	60	g	**1318.95**	0.00	312	0.00	0.00	0.00	10	39.47
p15	160	4	60	g	2505.42	0.00	1152	0.00	0.00	0.00	89	351.32
p18	240	6	60	g	3702.85	0.00	2712	0.00	0.00	0.00	273	1077.65
p21	360	9	60	g	5474.84	0.03	6000	0.00	0.00	0.00	774	3055.32
Mean						0.03	1282.91	0.00	0.02	0.06	143.73	567.35

Table 3.2: Computations on MDVRP instances.
Optimality has been proved for the solutions in boldface by Baldacci and Mingozzi (2008).

In particular, we noted that few iterations in the improvement phase are typically sufficient to converge to good local optima and few restarts are generally able to produce a moderate-sized and diversified pool of high-quality routes. In fact, the results obtained by setting $\delta = 1000$ or $\Delta = 1000$ were, on average, at most 0.2% better while requiring computing times up to 10 times larger. Moreover, we noticed that the algorithm is not very sensitive to reasonable values of η and ζ. Indeed, whenever values of those parameters that cause extremely disruptive effects or no changes at all are discarded, the algorithm is able to produce comparable results in similar computing time. On the contrary, π_{base}, ϕ and λ heavily affects the algorithm performance. The value of π_{base} was set following the guidelines defined in Toth and Vigo (2003) who reported that the best results are typically obtained with a sparsification factor between 1.0 and 2.5 and used a sparsification value equal to 1.25. From our experience, starting with a very aggressive sparsification level and reducing it relatively often resulted to be the most effective and efficient strategy.

5.3 Testing on MDVRP instances

We tested our algorithm on the subset of MDVRP instances with capacitated vehicles proposed by Cordeau, Gendreau, and Laporte (1997). The set contains 11 instances with 50 to 360 customers and up to nine depots. The first seven instances have randomly distributed customers, while in the last four they are arranged in geometrical patterns. We compared AVXS with the HGSADC (Hybrid Genetic Search with Advanced Diversity Control) algorithm proposed by Vidal et al. (2012), which is currently one of the most effective MDVRP solution approaches capable of handling the complete MDVRP instance set and several other VRP variants. In our computations, we omitted MDVRP instances with constraints on route length. Table 3.2 summarizes the computational results, organized in the following columns: the instance identifier (*Id*), the number of truck customers ($|V_c^1|$), the number of satellites ($|V_d|$), the truck capacity (Q_1), the instance type (*Type*) (equal to "rd" for randomly distributed and "g" for geometrical instances), and the best known solution value (*BKS*) reported in Vidal et al. (2012). For each algorithm we report (when available) the best (*Best*), the worst (*Worst*) and the average (*Avg*) gap of the solution found by the algorithm with respect to *BKS*, the computational time in seconds for 10 runs (t_{10}) and the corresponding scaled time ($\hat{t}_{10}$) computed as $\hat{t}_{10} = t_{10}(P_A/P_B)$ where P_A is our CPU single-thread rating and P_B is the CPU rating of the competing method. The gap is computed as $100 \cdot (z - BKS)/BKS$, where z is the solution value.

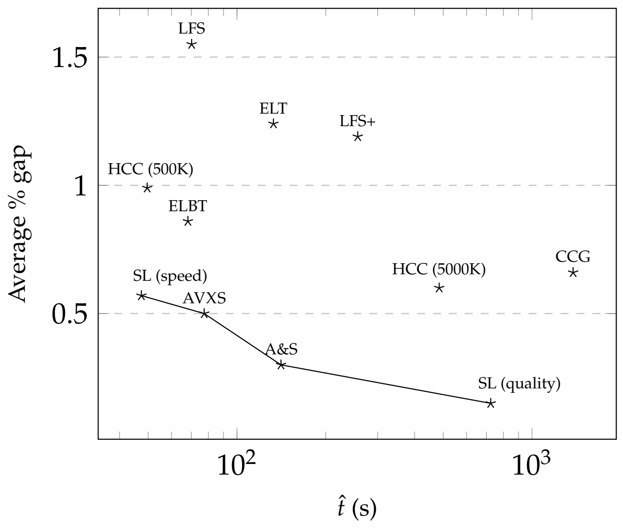


Figure 3.7: LRP algorithm performance comparison over the instances of Tuzun and Burke (1999). We excluded algorithms DLPP, YLLT and TC for which the average gaps are not available.

Because Vidal et al. (2012) reported only the average computing time of a single run, the values of t_{10} for HGSADC from the original paper have been multiplied by ten. Vidal et al. (2012) tested their algorithm on an AMD Opteron 250 CPU at 2.4 GHz. We were not able to find the exact CPU values in PassMark ®Software (2020) so we found a similar model, the AMD Opteron 275 at 2.2 GHz, which has a single-thread rating of 685; thus our CPU is roughly 3.95 times faster. As can be seen from the table, our algorithm was able to find a solution with the same value of the best known solution for all instances, and the average solution quality was comparable with that of HGSADC. Moreover, as our computing times, scaled to be comparable with those of the competing method, were much shorter, we can conclude that AVXS was able to reach state-of-the-art results on MDVRP instances within a very limited amount of time.

5.4 Testing on LRP instances

We tested AVXS on the set of 36 LRP instances proposed by Tuzun and Burke (1999), which are characterized by capacitated vehicles, uncapacitated depots, and fixed costs for using vehicles and depots. Each instance has between 100 and 200 customers, and between 10 and 20 candidate depots. Customers are uniformly randomly distributed or arranged in either three or five clusters. The vehicle capacity is 150. Table 3.3 shows the details of our computational results; the columns are the same as for Table 3.2. Type "c (h)" denotes clustered instances with h clusters. We observe that AVXS is able to find solutions close to the best known ones in relatively short computing times and even detected a new best solution for instance P123222. A general comparison of our method with other state-of-the-art methods is reported in Table 3.4, where the details of previous methods are obtained from the recent literature survey by Schneider and Drexl (2017). The table contains the following columns: the algorithm acronym (*Algorithm*), the best (*Best*) and, when available, the average (*Avg*) percentage gap with respect to the best known solution, the average raw run-time of a single run solving the considered LRP instances in seconds (t), the PassMark ®Software (2020) score as reported in the survey paper (P_{score}), and the scaled running time defined as the normalized single run-time $\times$ the number of conducted runs ($\hat{t}$). Run-times are normalized according to the processor used by Lopes, Ferreira, and Santos (2016). Finally, a graphic comparison of the algorithm performance is given in Figure 3.7.

Id	$\|V_c^1\|$	$\|V_d\|$	*Type*	*BKS*	*Best*	*Avg*	*Worst*	t_{10}
P111112	100	10	rd	**1467.68**	0.00	0.00	0.04	110
P111122	100	10	rd	1448.37	0.00	0.03	0.06	183
P111212	100	10	rd	**1394.80**	0.00	0.00	0.01	100
P111222	100	10	rd	1432.29	0.00	0.21	2.08	198
P112112	100	10	c (3)	**1167.16**	0.00	0.12	0.33	588
P112122	100	10	c (3)	1102.24	0.00	0.00	0.01	584
P112212	100	10	c (3)	**791.66**	0.00	0.01	0.15	124
P112222	100	10	c (3)	728.30	0.00	0.00	0.00	131
P113112	100	10	c (5)	1238.24	0.02	0.05	0.15	542
P113122	100	10	c (5)	1245.30	0.00	0.00	0.01	438
P113212	100	10	c (5)	**902.26**	0.00	0.00	0.00	433
P113222	100	10	c (5)	1018.29	0.00	0.00	0.00	499
P131112	150	10	rd	1892.17	0.56	1.08	2.08	704
P131122	150	10	rd	1819.68	0.25	1.06	2.14	770
P131212	150	10	rd	1960.02	0.24	0.27	0.38	420
P131222	150	10	rd	1792.77	0.00	0.26	0.82	688
P132112	150	10	c (3)	**1443.32**	0.00	0.22	0.31	808
P132122	150	10	c (3)	1429.30	0.86	1.24	1.50	875
P132212	150	10	c (3)	1204.42	0.10	0.20	0.46	699
P132222	150	10	c (3)	924.68	0.70	0.79	0.89	751
P133112	150	10	c (5)	1694.18	0.08	0.58	1.52	705
P133122	150	10	c (5)	1392.01	0.37	0.60	0.79	840
P133212	150	10	c (5)	1197.95	0.00	0.01	0.02	420
P133222	150	10	c (5)	1151.37	0.07	0.15	0.25	725
P121112	200	10	rd	2237.73	0.03	0.66	1.18	863
P121122	200	10	rd	2137.45	0.15	1.02	3.66	855
P121212	200	10	rd	2195.17	0.00	0.66	1.33	883
P121222	200	10	rd	2214.86	0.51	1.44	2.43	981
P122112	200	10	c (3)	2070.43	0.61	1.31	1.94	939
P122122	200	10	c (3)	1685.52	0.86	1.76	2.90	1039
P122212	200	10	c (3)	1449.93	1.20	1.44	1.56	936
P122222	200	10	c (3)	1082.46	0.02	0.07	0.25	937
P123112	200	10	c (5)	1942.23	1.38	1.64	1.76	965
P123122	200	10	c (5)	1910.08	0.09	0.53	1.79	1064
P123212	200	10	c (5)	1761.11	0.34	0.73	1.21	1081
P123222	200	10	c (5)	1390.86	-0.01	0.01	0.04	736
Mean					0.23	0.50	0.95	655.94

Table 3.3: Computations on LRP instances.
Optimality has been proved for the solutions in boldface by Baldacci, Mingozzi, and Wolfler Calvo (2011).
New best solutions (instance identifier, value): (P123222, 1390.74)

Algorithm	*Best*	*Avg*	*t*	P_{score}	$\hat{t}$
DLPP	1.40[a]		606.64	1200	317.55 × 5
YLLT	1.59[a]		826.47	1135	409.62 × 1
HCC (500K)	0.53	0.99	165.93	685	49.63 × 5
HCC (5000K)	0.39	0.60	1620.60	685	484.76 × 5
CCG	0.27	0.66	2589.53	1219	1378.44 × 10
ELT	1.24	1.24	392.33	776	132.94 × 1
TC	1.33[a]		202.08	546	48.18 × 10
ELBT	0.86	0.86	201.22	776	68.19 × 1
SL (speed)	0.31	0.57	65.75	1652	47.43 × 5
SL (quality)	0.04	0.15	1004.65	1652	724.75 × 5
LFS	0.96	1.55	86.02	2290	70.02 × 10
LFS+	0.80	1.19	363.61	2290	256.66 × 10
A&S	0.30	0.30	171.93	1878	141.00 × 1
AVXS	0.23	0.50	65.59	2704	77.45 × 10

Table 3.4: Comparison with state-of-the-art LRP algorithms.
[a]Data on average gaps not available.
The algorithm acronyms are: DLPP (Duhamel et al. (2010)), YLLT (Yu et al. (2010)), HCC (Hemmelmayr, Cordeau, and Crainic (2012)), CCG (Contardo, Cordeau, and Gendron (2014)), ELT (Escobar, Linfati, and Toth (2013)), TC (Ting and Chen (2013)), ELBT (Escobar et al. (2014)), SL (Schneider and Löffler (2019)), LFS (Lopes, Ferreira, and Santos (2016)), A&S (Florian (2018))

We conclude that our AVXS favourably compares with the best existing methods, achieving an excellent compromise between solution quality and running time, as shown by its presence on the Pareto frontier of Figure 3.7.

5.5 Testing on STTRPSD instances

The STTRPSD instances were proposed by Villegas et al. (2010); they are a set of 32 randomly generated Euclidean instances defined in a square grid of 100 × 100 units. The set contains randomly distributed and clustered instances including only truck customers and satellite depots. The size of the instances vary from 25 to 200 truck customers and from 5 to 20 satellites. The truck capacity is either 1000 or 2000, and the customer demand is uniformly distributed in the interval [1, 200]. The algorithm of Villegas et al. (2010) was implemented in Java. The experiments were run on a computer equipped with an Intel Pentium D 945 processor running at 3.4 GHz with 1024 MB of RAM. We were not able to find the exact CPU in PassMark ®Software (2020), so we referred to a similar model, namely the Intel Pentium D 940 working at 3.20 GHz, which has a single-thread rating of 758. Our CPU is thus 3.57 times faster. The authors presented four metaheuristics with various characteristics and levels of performance. In Table 3.5 we compare our algorithm with the algorithm MS-ILS of Villegas et al. (2010), which is the best-performing one with respect to the objective function (and just slightly more time-consuming than the second-best).

Table 3.5 includes several columns similar to those of previous result tables. Since Villegas et al. (2010) reported only the average computing time needed to perform a single run of their algorithm, to obtain the values of t_{10} we multiplied their times by ten. As can be seen from Table 3.5, AVXS was able to find seven new best solutions for the largest instances. Moreover, we note that instances 1-18, 21 and 22 have already been proved to be optimal by Belenguer et al. (2016); thus we improved 7 out of 12 of the remaining ones. The computational results show the effectiveness of our method, in terms of both solution quality and processing time (which is more than ten times shorter than the competing algorithm).

						MS-ILS				AVXS				
Id	$\|V_c^1\|$	$\|V_d\|$	Q_1	*Type*	*BKS*	*Best*	*Avg*	*Worst*	t_{10}	*Best*	*Avg*	*Worst*	t_{10}	$\hat{t}_{10}$
1	25	5	1000	c	**405.46**	0.00	0.00	0.00	114	0.00	0.00	0.00	10	35.67
2	25	5	2000	c	**374.79**	0.00	0.00	0.00	114	0.00	0.00	0.00	10	35.67
3	25	5	1000	rd	**584.03**	0.00	0.00	0.00	126	0.00	0.00	0.00	10	35.67
4	25	5	2000	rd	**508.48**	0.00	0.00	0.00	102	0.00	0.00	0.00	10	35.67
5	25	10	1000	c	**386.45**	0.00	0.00	0.00	132	0.00	0.00	0.00	10	35.67
6	25	10	2000	c	**380.86**	0.00	0.00	0.00	126	0.00	0.00	0.00	10	35.67
7	25	10	1000	rd	**573.96**	0.00	0.00	0.00	138	0.00	0.00	0.00	10	35.67
8	25	10	2000	rd	**506.37**	0.00	0.00	0.00	126	0.00	0.00	0.00	10	35.67
9	50	5	1000	c	**583.07**	0.00	0.00	0.00	708	0.00	0.00	0.00	10	35.67
10	50	5	2000	c	**516.98**	0.00	0.00	0.00	570	0.00	0.00	0.00	10	35.67
11	50	5	1000	rd	**870.51**	0.00	0.00	0.00	612	0.00	0.00	0.00	10	35.67
12	50	5	2000	rd	**766.03**	0.00	0.00	0.00	588	0.00	0.00	0.00	10	35.67
13	50	10	1000	c	**387.83**	0.00	0.00	0.00	822	0.00	0.00	0.00	10	35.67
14	50	10	2000	c	**367.01**	0.00	0.00	0.00	750	0.00	0.00	0.00	10	35.67
15	50	10	1000	rd	**811.28**	0.00	0.00	0.00	720	0.00	0.00	0.00	10	35.67
16	50	10	2000	rd	**731.53**	0.00	0.00	0.00	630	0.00	0.00	0.00	10	35.67
17	100	10	1000	c	**614.02**	0.00	0.06	0.21	3480	0.00	0.00	0.00	58	206.90
18	100	10	2000	c	**547.44**	0.00	0.02	0.12	3894	0.00	0.00	0.00	30	107.02
19	100	10	1000	rd	1275.76	0.33	0.55	0.81	2976	-0.31	-0.31	-0.31	72	256.84
20	100	10	2000	rd	1097.28	0.00	0.06	0.56	2604	0.00	0.00	0.00	52	185.5
21	100	20	1000	c	**642.61**	0.00	0.00	0.00	2892	0.00	0.00	0.00	65	231.87
22	100	20	2000	c	**581.56**	0.00	0.11	0.37	3600	0.00	0.00	0.00	40	142.69
23	100	20	1000	rd	1143.10	0.00	0.31	0.63	3012	0.00	0.00	0.00	58	206.9
24	100	20	2000	rd	1060.75	0.00	0.20	0.40	2946	0.01	0.24	0.27	91	324.62
25	200	10	1000	c	822.52	0.00	0.71	1.52	11544	-0.31	-0.31	-0.31	311	1109.42
26	200	10	2000	c	714.33	0.00	0.79	1.63	11682	-0.51	-0.44	-0.29	284	1013.11
27	200	10	1000	rd	1761.10	0.12	1.26	2.53	10314	-0.32	-0.32	-0.32	418	1491.12
28	200	10	2000	rd	1445.94	0.00	0.84	1.71	9288	0.00	0.00	0.00	255	909.66
29	200	20	1000	c	909.46	0.00	0.45	1.51	13746	-0.25	-0.25	-0.25	319	1137.96
30	200	20	2000	c	815.51	0.63	0.82	1.14	14646	-0.13	-0.13	-0.09	227	809.77
31	200	20	1000	rd	1614.18	0.00	1.08	2.16	12282	-0.22	-0.22	-0.22	341	1216.44
32	200	20	2000	rd	1413.32	0.00	0.81	1.81	12084	0.00	0.00	0.00	307	1095.16
Mean						0.03	0.25	0.54	3980.25	-0.06	-0.05	-0.05	96.50	344.24

Table 3.5: Computations on STTRPSD instances.
Optimality has been proved for the solutions in boldface by Belenguer et al. (2016).
New best solutions (instance identifier, value): (19, 1271.78) (25, 819.96) (26, 710.70) (27, 1755.44) (29, 907.17) (30, 814.42) (31, 1610.62)

5.6 Testing on STTRP instances

In order to assess the performances of AVXS on the STTRP, we first solved the small-scale instances proposed by Bartolini and Schneider (2018) and used in a branch-and-cut algorithm. They had selected subsets of customers with sizes of 30 and 40 from the TTRP instances proposed by Chao (2002) and Lin, Yu, and Chou (2010), deriving the set of 36 instances that we used for initial testing. Moreover, to comply with the multiple-vehicle versions of the original TTRP in which each vehicle can perform at most one route type among the ones defined by Chao (2002) (see Section 2), Bartolini and Schneider (2018) imposed the constraint of not having any sub-route directly starting from the depot even in the single-vehicle version. Since the XSTTRP does not allow sub-routes directly starting from the depot, we could capture the STTRP considered by Bartolini and Schneider (2018) as a special case of our problem. Table 3.6 contains the computational results of those instances. We computed the arc costs $c_{ij} = (\lfloor 10000 \cdot Euc(i,j) \rfloor)/10000$ as in the the original paper. By analyzing the computational results of the set of instances shown in Table 3.6, we can conclude that AVXS is able to solve small-scaled instances of this XSTTRP sub-problem effectively. In particular, the algorithm improved five best known solutions.

After successfully solving the small-sized STTRP instances, we derived larger test instances from the TTRP ones proposed by Chao (2002) by setting the trailer capacity Q_2 equal to the sum of customer demands and adding a dummy satellite at the same location as the main depot, to allow an unlimited number of sub-routes starting from the depot location. These new instances have either randomly distributed or clustered customers, who number between 50 and 200, and the capacity of the truck Q_1 is either 100 or 150. Table 3.7 shows our computational results with columns similar to previous tables. The number of vehicle customers with parking facilities is denoted as $|\overline{V_c^2}|$, and BKS identifies the best known solution value found during the experiments.

From the table we note that AVXS provides results with very small differences between the best and the worst average gap, and the computing times to perform ten runs never exceed 13 minutes for the largest instances and are, on average, less than three minutes.

5.7 Testing on XSTTRP instances

We derived a set of XSTTRP instances from the TTRP ones proposed by Chao (2002) as follows. For each TTRP instance, for n_c the number of TTRP customers, we turned n_s randomly chosen customers into satellites. From the remaining $n'_c = n_c - n_s$ customers, we randomly chose $t \cdot n'_c$ to turn into vehicle customers without parking facilities. The possible values for n_s depend on the original TTRP instance size: $n_s = 5$ if $n_c \leq 99$, $n_s \in \{5, 10\}$ when $100 \leq n_c \leq 199$ and $n_s \in \{5, 10, 20\}$ if $200 \leq n_c$. The values for t are $t \in \{0.2, 0.8\}$. Moreover, every instance contains an additional satellite placed at the depot location.

Table 3.8 shows our computational results; in addition to the columns already defined for previous sub-problems we include a column for the number of vehicle customers without parking facilities ($|V_c^2 \setminus \overline{V_c^2}|$).

The results show that, as for the STTRP instances, the average difference between the best and the worst average gap remains small. This experimentally indicates that AVXS is able to provide stable results for XSTTRP instances. Moreover, the average computing time to perform a single run is always less than 80 seconds and on average slightly more that 19 seconds.

Id	$\lvert V_c^1 \rvert$	$\lvert \overline{V_c^2} \rvert$	Q_1	*Type*	*BKS*	*Best*	*Avg*	*Worst*	t_{10}
chao1.30	7	23	150	rd	**350.6**	0.00	0.00	0.00	10
chao2.30	15	15	150	rd	**371.6**	0.00	0.00	0.00	10
chao3.30	23	7	150	rd	**383.6**	0.00	0.00	0.00	10
chao4.30	7	23	150	rd	**351.7**	0.00	0.00	0.00	10
chao5.30	15	15	150	rd	**383.8**	0.00	0.00	0.00	10
chao6.30	22	8	150	rd	**435.8**	0.00	0.00	0.00	10
chao7.30	8	22	200	rd	**365.1**	0.00	0.00	0.00	10
chao8.30	15	15	100	rd	**419.5**	0.00	0.00	0.00	10
chao9.30	22	8	200	rd	**408.0**	0.00	0.00	0.00	10
chao10.30	8	22	100	rd	**411.5**	0.00	0.00	0.00	10
chao11.30	15	15	100	rd	**398.8**	0.00	0.00	0.00	10
chao12.30	22	8	100	rd	**471.8**	0.00	0.00	0.00	10
tai1	7	23	750	c	**508.9**	0.02	0.02	0.02	10
tai2	15	15	750	c	**711.3**	0.00	0.00	0.00	10
tai3	22	8	750	c	**757.4**	0.00	0.00	0.00	10
tai4	7	23	850	c	**310.6**	0.00	0.00	0.00	10
tai5	15	15	850	c	**333.9**	0.00	0.00	0.00	10
tai6	22	8	850	c	**359.1**	0.00	0.00	0.00	10
tai7	7	23	600	c	**491.1**	0.00	0.00	0.00	10
tai8	15	15	600	c	**538.5**	0.00	0.00	0.00	10
tai9	23	7	600	c	**585.4**	0.00	0.00	0.00	10
tai10	7	23	850	c	**608.8**	0.00	0.00	0.00	10
tai11	15	15	850	c	**667.6**	0.00	0.00	0.00	10
tai12	23	7	850	c	**719.1**	0.00	0.00	0.00	10
chao1.40	10	30	150	rd	399.2	0.03	0.03	0.03	10
chao2.40	20	20	150	rd	452.5	-2.48	-2.48	-2.48	10
chao3.40	30	10	150	rd	472.3	0.00	0.00	0.00	10
chao4.40	10	30	150	rd	415.2	-0.17	-0.17	-0.17	10
chao5.40	20	20	150	rd	455.5	0.00	0.00	0.00	10
chao6.40	30	10	150	rd	501.0	0.00	0.00	0.00	10
chao7.40	10	30	200	rd	423.1	-0.21	-0.21	-0.21	10
chao8.40	20	20	100	rd	461.5	0.00	0.00	0.00	10
chao9.40	30	10	200	rd	449.0	0.00	0.00	0.00	10
chao10.40	10	30	100	rd	502.2	-3.33	-3.33	-3.33	10
chao11.40	20	20	100	rd	500.7	0.00	0.00	0.00	10
chao12.40	30	10	100	rd	576.6	-0.36	-0.36	-0.36	10
Mean						-0.18	-0.18	-0.18	10

Table 3.6: Computations on STTRP instances proposed by Bartolini and Schneider (2018).
Optimality has been proved for the solutions in boldface by Bartolini and Schneider (2018), for the remaining ones *BKS* reports the upper-bound provided by Bartolini and Schneider (2018).
New best solutions (instance identifier, value): (chao2.40, 441.3) (chao4.40, 414.5) (chao7.40, 422.2) (chao10.40, 485.5) (chao12.40, 574.5)
The objective values from which we computed the above gaps were rounded to the first decimal place as in the original paper.

Id	$\lvert V_c^1\rvert$	$\lvert \overline{V_c^2}\rvert$	Q_1	*Type*	*BKS*	*Best*	*Avg*	*Worst*	t_{10}
1	12	38	100	rd	486.07	0.00	0.00	0.00	19
2	25	25	100	rd	548.14	0.00	0.00	0.00	20
3	37	13	100	rd	583.32	0.00	0.00	0.00	10
4	18	57	100	rd	617.13	0.00	0.01	0.09	41
5	37	38	100	rd	676.42	0.00	0.00	0.00	44
6	56	19	100	rd	769.88	0.00	0.00	0.00	30
7	25	75	100	rd	687.64	0.00	0.16	0.28	92
8	50	50	150	rd	745.19	0.00	0.00	0.00	102
9	75	25	150	rd	821.31	0.00	0.17	0.36	78
10	37	113	150	rd	790.54	0.00	0.13	0.23	377
11	75	75	150	rd	857.27	0.00	0.02	0.16	285
12	112	38	150	rd	936.00	0.06	0.21	0.39	249
13	49	150	150	rd	875.85	0.15	0.34	0.71	769
14	99	100	150	rd	950.80	0.19	0.63	1.15	620
15	149	50	150	rd	1049.97	0.18	0.26	0.43	476
16	30	90	150	c	579.29	0.00	0.08	0.18	140
17	60	60	150	c	611.30	0.00	0.00	0.00	136
18	90	30	150	c	698.57	0.00	0.00	0.00	73
19	25	75	150	c	541.87	0.00	0.00	0.00	65
20	50	50	150	c	582.62	0.00	0.00	0.00	75
21	75	25	150	c	676.13	0.00	0.00	0.00	61
Mean						0.03	0.10	0.19	179.14

Table 3.7: Computations on STTRP instances.

6 Algorithm components analysis

This section examines the most relevant design choices by providing a detailed analysis of the different components of AVXS, namely: *(i)* the construction procedure to build an initial feasible solution, *(ii)* the different neighborhoods explored in the RVND and *(iii)* the shaking procedures used in the improvement phase, *(iv)* the granular speedup strategy, and *(v)* the set-partitioning-based post-optimization phase. The results presented here refer primarily to the XSTTRP variant, the most general of those we studied. However, similar conclusions apply to the other variants considered in this chapter.

Table 3.8: Computations on XSTTRP instances

Id	$\lvert V_c^1\rvert$	$\lvert V_c^2 \setminus \overline{V_c^2}\rvert$	$\lvert \overline{V_c^2}\rvert$	$\lvert V_d\rvert$	Q_1	*Type*	*BKS*	*Best*	*Avg*	*Worst*	t_{10}
1	73	17	5	6	150	c	753.43	0.00	0.00	0.00	61
2	66	4	20	11	150	c	602.99	0.00	0.00	0.00	40
3	67	18	5	11	150	c	696.03	0.00	0.00	0.00	50
4	22	4	19	6	100	rd	505.23	0.00	0.00	0.00	10
5	21	19	5	6	100	rd	517.86	0.00	0.00	0.00	10
6	18	10	42	6	100	rd	586.87	0.00	0.00	0.00	40
7	18	41	11	6	100	rd	624.39	0.00	0.00	0.00	30
8	23	14	58	6	150	c	538.40	0.00	0.00	0.00	60
9	25	56	14	6	150	c	582.27	0.00	0.00	0.00	40
10	21	13	56	11	150	c	522.54	0.00	0.00	0.00	61
11	22	54	14	11	150	c	556.61	0.00	0.00	0.00	43
12	108	7	30	6	150	rd	916.39	0.00	0.10	0.19	224
13	108	29	8	6	150	rd	935.35	0.00	0.00	0.00	184
14	106	6	28	11	150	rd	906.58	0.02	0.12	0.29	365
15	103	29	8	11	150	rd	896.46	0.02	0.26	0.50	196
16	36	21	88	6	150	rd	776.73	0.00	0.12	0.36	315
17	35	88	22	6	150	rd	792.22	0.00	0.27	0.77	207
18	36	20	84	11	150	rd	766.28	0.03	0.12	0.31	308
19	35	84	21	11	150	rd	773.97	0.00	0.03	0.09	181

Table 3.8: Computations on XSTTRP instances

Id	$\lvert V_c^1\rvert$	$\lvert V_c^2 \setminus \overline{V_c^2}\rvert$	$\lvert\overline{V_c^2}\rvert$	$\lvert V_d\rvert$	Q_1	*Type*	*BKS*	*Best*	*Avg*	*Worst*	t_{10}
20	48	9	38	6	150	rd	735.26	0.00	0.00	0.01	90
21	48	37	10	6	150	rd	743.71	0.00	0.00	0.00	73
22	45	9	36	11	150	rd	700.54	0.00	0.00	0.00	95
23	45	36	9	11	150	rd	715.08	0.00	0.01	0.10	81
24	52	3	15	6	100	rd	716.31	0.00	0.00	0.00	30
25	52	14	4	6	100	rd	802.23	0.00	0.00	0.00	40
26	34	7	29	6	100	rd	652.27	0.00	0.00	0.00	41
27	34	28	8	6	100	rd	661.35	0.00	0.00	0.00	30
28	71	14	60	6	150	rd	849.31	0.05	0.16	0.40	290
29	73	57	15	6	150	rd	887.13	0.00	0.23	0.43	229
30	73	13	54	11	150	rd	820.05	0.02	0.13	0.24	264
31	71	55	14	11	150	rd	858.46	0.01	0.10	0.41	207
32	25	14	56	6	150	rd	667.99	0.00	0.03	0.05	80
33	24	56	15	6	150	rd	671.63	0.00	0.04	0.19	56
34	22	13	55	11	150	rd	627.53	0.00	0.01	0.04	90
35	23	53	14	11	150	rd	651.17	0.04	0.09	0.12	62
36	98	19	77	6	150	rd	948.90	0.43	0.54	0.89	582
37	98	76	20	6	150	rd	980.57	0.10	0.23	0.47	418
38	95	18	76	11	150	rd	931.48	0.00	0.24	0.60	566
39	93	76	20	11	150	rd	963.67	0.48	0.75	1.06	439
40	92	17	70	21	150	rd	900.43	0.02	0.21	0.39	525
41	88	72	19	21	150	rd	929.58	0.34	0.72	1.42	454
42	71	4	20	6	150	rd	787.71	0.00	0.03	0.27	90
43	72	18	5	6	150	rd	784.69	0.00	0.26	0.51	61
44	68	4	18	11	150	rd	758.79	0.01	0.05	0.12	92
45	67	18	5	11	150	rd	773.22	0.00	0.00	0.00	72
46	144	10	40	6	150	rd	1040.68	0.00	0.03	0.12	509
47	145	39	10	6	150	rd	1085.86	0.17	0.29	0.50	460
48	140	9	40	11	150	rd	1009.72	0.02	0.05	0.07	403
49	143	36	10	11	150	rd	1069.28	0.06	0.31	0.67	576
50	132	9	38	21	150	rd	971.67	0.05	0.28	0.70	477
51	135	35	9	21	150	rd	997.43	0.00	0.12	0.35	371
52	57	11	47	6	150	c	603.08	0.00	0.00	0.00	90
53	58	45	12	6	150	c	666.36	0.00	0.00	0.00	73
54	53	11	46	11	150	c	589.21	0.00	0.00	0.03	111
55	54	44	12	11	150	c	607.69	0.00	0.02	0.05	80
56	29	17	69	6	150	c	570.53	0.00	0.01	0.13	121
57	29	68	18	6	150	c	580.00	0.00	0.00	0.00	79
58	26	16	68	11	150	c	561.61	0.08	0.14	0.27	122
59	28	65	17	11	150	c	560.75	0.00	0.12	0.41	70
60	87	5	23	6	150	c	645.86	0.00	0.00	0.00	74
61	86	23	6	6	150	c	777.00	0.00	0.00	0.00	91
62	84	5	21	11	150	c	642.40	0.00	0.00	0.00	68
63	83	21	6	11	150	c	717.47	0.00	0.01	0.11	69
64	33	2	10	6	100	rd	557.56	0.00	0.00	0.00	10
65	34	8	3	6	100	rd	553.54	0.00	0.00	0.00	10
66	49	29	116	6	150	rd	869.02	0.12	0.24	0.35	622
67	47	117	30	6	150	rd	897.21	0.16	0.52	0.90	433
68	44	29	116	11	150	rd	858.61	0.03	0.39	0.65	726
69	48	112	29	11	150	rd	878.07	0.04	0.51	0.91	388
70	43	27	109	21	150	rd	825.73	0.16	0.63	0.85	725
71	45	107	27	21	150	rd	860.21	0.41	0.71	1.12	416
72	11	6	28	6	100	rd	456.01	0.00	0.00	0.00	18
73	12	26	7	6	100	rd	485.42	0.00	0.00	0.00	10
74	49	9	37	6	150	c	573.69	0.00	0.00	0.00	66
75	47	38	10	6	150	c	660.97	0.00	0.00	0.00	51
76	44	9	37	11	150	c	567.98	0.00	0.00	0.00	68
77	47	34	9	11	150	c	594.29	0.00	0.00	0.00	40

Table 3.8: Computations on XSTTRP instances

Id	$\lvert V_c^1\rvert$	$\lvert V_c^2 \setminus \overline{V_c^2}\rvert$	$\lvert\overline{V_c^2}\rvert$	$\lvert V_d\rvert$	Q_1	*Type*	*BKS*	*Best*	*Avg*	*Worst*	t_{10}
Mean								0.04	0.12	0.24	193.62

6.1 Assignment fitness functions

The quality of the initial solution is tightly linked to the definition of what we called the assignment fitness function, see Section 4.1. Due to the highly complex interactions between heuristic algorithm components, good starting points do not guarantee a final solution of superior quality. In this paragraph we experimentally analyze the effect of using different assignment fitness functions on the quality of the initial and final solutions, as well as on the search diversification achieved. To this end, we compared two general approaches to defining the function, using a range of different parameters. The first approach was the restricted GRASP-based function d-NEAR that we defined in Section 4.1. In contrast, the second was a rank-based function b-RANK, defined as $f^i(j) = 1 + (|\overline{V}_d^+| - r_j)^b$, where r_j is the position for j in a list including all $k \in V$ and sorted according to the assignment costs $\hat{c}$; b is the scaling factor. The b-RANK function can be classified as an empirical bias function. We refer the interested reader to the work by Grasas et al. (2017) for a general survey on biased randomized procedures with theoretical or empirical bias functions. More precisely, we ran algorithm AVXS for ten executions on all instances of the XSTTRP data set, and Table 3.9 shows, for each assignment function (*Function)*, the average gap of the starting solutions obtained after the construction phase averaged over the Δ restarts (*Start*), the best, average and worst final gap (*Best*, *Avg*, *Worst*) averaged over all instances, the average total computing time for the ten runs in seconds (t_{10}), the average percentage improvement provided by the polishing phase (*sp*), the percentage of time to perform the polishing phase with reference to the total time of the algorithm ($\%t_{sp}$), and the number of routes used in the set-partitioning model F ($|\mathcal{P}|$). From the table it is clear that neither the final solution gap nor the computing time depend on the initial gap, but the differences in computing time are strongly correlated with the post-optimization polishing phase. In fact, the standard deviation of the computing time among the different approaches without considering the set-partitioning resolution is less than five seconds. Moreover, some randomization on the initial assignment was beneficial compared to the deterministic assignment of the 1-NEAR function. In fact, all considered approaches (except the 1-NEAR) behaved in a similar way. Among the best performing functions we note that the 25-NEAR was the one that obtained good results while being conceptually simpler than the empirical bias functions of the b-RANK family. As an additional note, we report that using the 3-RANK we were able to find an additional best known solution

Function	*Start*	*Best*	*Avg*	*Worst*	t_{10}	*sp*	$\%t_{sp}$	$\lvert\mathcal{P}\rvert$
1-RANK	341.25	0.05	0.13	0.26	202.57	0.05	32.85	6862.04
2-RANK	281.65	0.04	0.12	0.24	196.01	0.05	30.60	6564.64
3-RANK	243.35	0.04	0.13	0.24	190.97	0.05	29.12	6328.70
4-RANK	217.20	0.04	0.13	0.24	186.70	0.05	28.56	6142.69
5-RANK	196.56	0.04	0.13	0.25	185.74	0.05	27.31	5986.62
1-NEAR	36.22	0.09	0.23	0.35	134.19	0.04	9.31	2591.15
2-NEAR	52.13	0.05	0.15	0.27	145.42	0.05	14.69	3764.73
5-NEAR	109.16	0.05	0.13	0.25	160.36	0.04	19.77	4973.16
10-NEAR	184.40	0.05	0.13	0.23	175.64	0.05	26.11	5859.38
25-NEAR	311.53	0.04	0.12	0.24	193.62	0.04	31.76	6738.41

Table 3.9: Computations on XSTTRP instances with different assignment fitness functions.

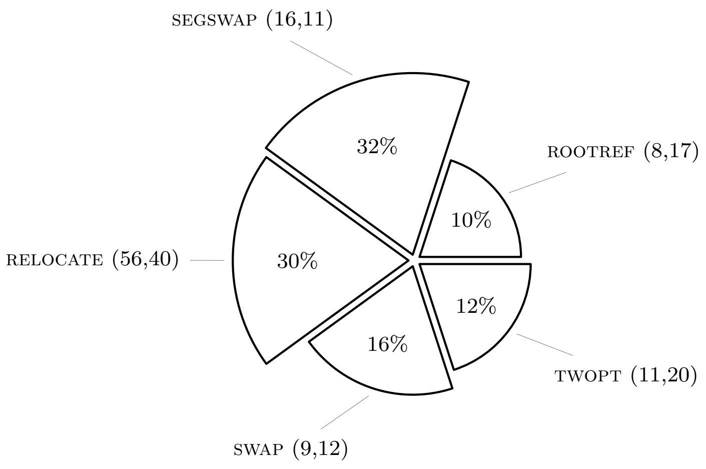


Figure 3.8: The percentage Relative Neighborhood Effectiveness Index for the XSTTRP instances.

for instance 28 of the STTRPSD with an objective value of 1445.05 (0.06% better than the current one). Other problem variants produced very similar results, with the exception of LRP in which the deterministic 1-NEAR turned out to be preferable, because of the strong influence of the fixed costs in the solution process (as was discussed in Section 4.5).

6.2 Local search

As extensively described in Section 4, our local search engine is based on an RVND scheme that searches five different neighborhoods in random order according to a randomized first-improvement strategy. In this section, we analyze the impact of the various neighborhoods we used. First of all, we observed that all the neighborhoods contributed significantly to the effectiveness of the local search step. More precisely, let **N** be the set of all the neighborhoods we considered, for each neighborhood $\mathcal{N} \in \mathbf{N}$ we stored the total improvement value $D(\mathcal{N})$ it produced and the number $I(\mathcal{N})$ of times a complete application of $\mathcal{N}$ resulted in an improvement. Then, for each neighborhood $\mathcal{N}$, we computed a *Relative Neighborhood Effectiveness Index* or RNEI($\mathcal{N}$), which defines the effectiveness of $\mathcal{N}$ with respect to the other neighborhoods. In particular, RNEI($\mathcal{N}$) is computed as RNEI($\mathcal{N}$) $= D(\mathcal{N})I_\Sigma / D_\Sigma I(\mathcal{N})$ where $D_\Sigma = \sum_{\mathcal{N}' \in \mathbf{N}} T(\mathcal{N}')$ and $I_\Sigma = \sum_{\mathcal{N}' \in \mathbf{N}} I(\mathcal{N}')$. In other words, RNEI($\mathcal{N}$) measures the successful improvement that any application of $\mathcal{N}$ would produce on an XSTTRP solution compared to the application of any other neighborhood in the set.

Figure 3.8 shows an aggregate measure, averaged over ten runs, for all the XSTTRP instances, for each neighborhood $\mathcal{N}$ computed as $100 \cdot \text{RNEI}(\mathcal{N}) / \sum_{\mathcal{N}' \in \mathbf{N}} \text{RNEI}(\mathcal{N}')$. The values (X, Y) reported beside each neighborhood $\mathcal{N}$ name are $X = 100 \cdot D(\mathcal{N}) / D_\Sigma$ and $Y = 100 \cdot I(\mathcal{N}) / I_\Sigma$. For example, the values reported for SEGSWAP show that its application was less frequently successful with respect to other neighborhoods (11%) but the average improvement that it had produced (16%) during those applications made it the most effective operator, with an RNEI percentage score equal to 32%. However, Figure 3.8 clearly shows that all neighborhoods made a positive contribution to the overall local search effectiveness. We also note that because we used an RVND approach, the RNEI measure is not affected by the order in which the neighborhoods are examined.

Next, we compared our implemented scheme to a classical VND with fixed operators and moves ordering. In particular, we examined a naive reasonable order given by RELOCATE, SWAP, TWOPT, SEGSWAP, and ROOTREF and a best-RNEI order defined as SEGSWAP, RELOCATE,

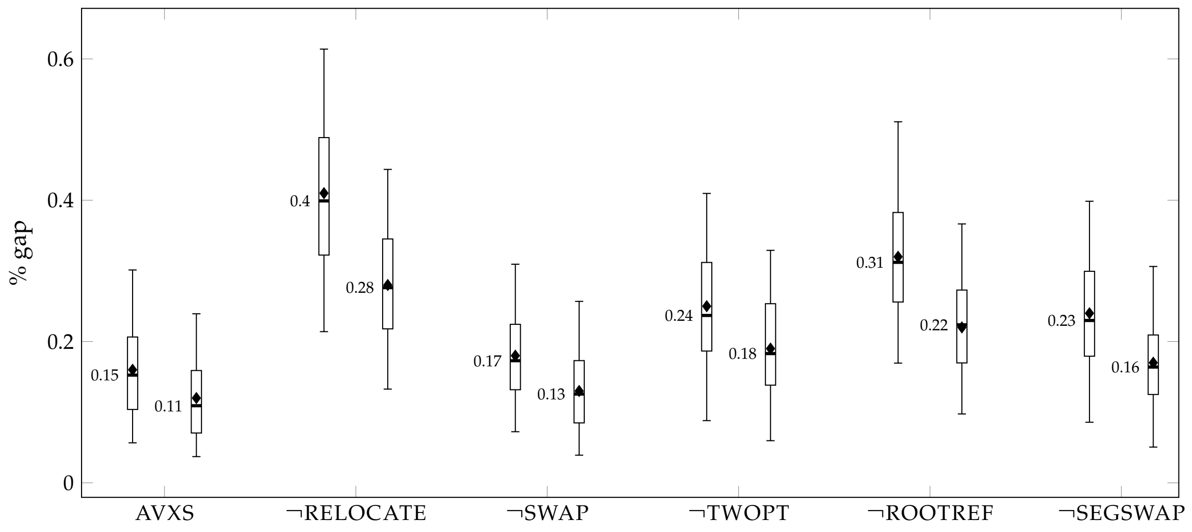


Figure 3.9: Computations on XSTTRP instances with the complete approach (AVXS) and disabling one local search operator at a time (¬OPERATOR). For each configuration, the left box-plot represents the results before the polishing phase and the right one, the results after the polishing phase. The median value is shown on the left of each box-plot.

SWAP, TWOPT, and ROOTREF. We executed ten runs of AVXS on all XSTTRP instances with both approaches. The results with the RVND were slightly better than those with the VND. In particular, the final gap obtained by the RVND over all instances was about 0.03%, resp. 0.01%, smaller when compared to the VND with naive, resp. best-RNEI, ordering. The computing time was similar for all the three approaches. Moreover, the solutions produced by the RVND appeared to be more diversified, because the set-partitioning polishing phase obtained better final results with less effort when fed with the RVND solutions rather than with those of the VND. Thus, we can conclude that the adoption of randomization in the selection of the next neighborhood (and of the next move within the neighborhood) may play an important role in improving the final solution quality without lengthening the computational time.

In addition, we evaluated what happens if a certain operator is removed entirely. The results are shown in Figure 3.9. On the one hand, none of the operators seems mandatory to obtain high quality final solutions. On the other hand, all of them play a role in determining the best possible result.

Finally, we analyzed the impact of the two shake operators we described in Section 4.3. We experimentally observed that both operators proved to be effective in inducing improvements in the solutions. In particular, over all XSTTRP instances, the percentage improvement associated with a local search step following a SUNLOAD shaking was equal to 39% of the total improvement, while that associated with SREM was 61%. Similar results were obtained with the other problem classes; therefore we can conclude that both shaking operators are required for maximum effectiveness.

6.3 Granular neighborhoods adoption

We studied the effect of granular neighborhoods by comparing the average solution quality and the computing time needed to run AVXS, using either the granular or the complete neighborhood exploration. The results confirmed that use of granular neighborhoods is an effective speedup technique, greatly improving computing time while only slightly decreasing solution quality. In particular, we observed that the average running time to execute ten runs over all

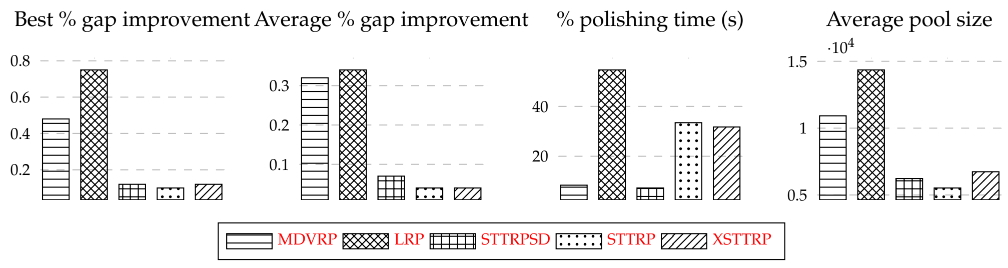


Figure 3.10: Polishing phase contribution

XSTTRP instances with complete neighborhoods exploration was 904.87 seconds and the average best gap was 0.02%. In contrast, using granular neighborhoods, as reported in Table 3.8, the average computing time was only 193.62 seconds and the average best gap was 0.04%. We can conclude that the adoption of granular neighborhoods make it possible to reach high-quality solutions with a very limited running time (almost five times shorter than that of an improvement phase considering complete neighborhoods).

6.4 Set-partitioning post-optimization

The polishing phase glues together the different restarts, making the overall algorithm more robust and stable. The bar diagram in Figure 3.10 shows the average polishing phase improvement, the average percent processing time with reference to that of the complete algorithm, and the average size of the pool $\mathcal{P}$ for the different problem variants we considered. The figure shows that for all variants, the polishing phase was effective. Moreover, with the exception of LRP (which had a considerably larger solution pool), the phase's computing time was always less than 35% of the total. As one can expect, the improvement is much more relevant for problems that required additional care to be modeled as XSTTRP. In fact, both the MDVRP and the LRP have additional constraints which are not explicitly considered by the XSTTRP model and the local search procedures. On the other hand, the core part of AVXS is alone able to generate high quality solutions for the STTRPSD and the STTRP which are straightforward XSTTRP special cases. However, by observing Figure 3.9, one can still see the beneficial effects of the polishing phase on the XSTTRP instances, in terms of improvement and stabilization of the results.

7 Conclusions

In this chapter we presented AVXS, an effective hybrid metaheuristic for a general variant of single-vehicle truck and trailer routing problems called the Extended Single Truck and Trailer Routing Problem (XSTTRP). The AVXS algorithm is based on a four-phase solution approach in which the main improvement phase consists of an iterated local search (ILS) incorporating two shaking procedures and a randomized variable neighborhood descent with granular speedup. A set of high-quality solutions generated during the ILS is stored in a pool, which is used for a final polishing phase based on the solution of a restricted mathematical formulation of the problem. Therefore, AVXS can also be classified as a matheuristic (see Maniezzo, Stützle, and Voß (2010)).

The AVXS algorithm was used to solve, with very short computing times, a large set of instances for several variants of the problem and proved able to reach state-of-the-art results for those variants already studied in the literature. In particular, AVXS was able to detect one new best solution for the extensively studied Location Routing Problem and seven new best solutions for the Single Truck and Trailer Routing Problem with Satellite Depots.

The study of the XSTTRP variant opens different research possibilities that could enrich the problem's definition such as, for example, considering time windows and extending it to multiple vehicles. Furthermore, the development of exact methods, which are seldom studied in the truck and trailer literature, might be a difficult but extremely interesting research direction.

8 Acknowledgments

This research was partially funded by Ministero dell'Istruzione dell'Università e della Ricerca, Italy under grant PRIN n. 2015JJLC3E_002 and by United States Air Force Office of Scientific Research under award number FA9550-17-1-0234.

Chapter 4

Guidelines for the Computational Testing of Machine Learning approaches to Vehicle Routing Problems

Despite the extensive research efforts and the promising results obtained by the ML community on Vehicle Routing Problems, most of the proposed techniques are still seldom employed by the OR community. With the current work, we highlight a number of challenges arising during the computational evaluation of heuristics for VRPs. The resulting guidelines aim at defining a common testing setup for the approaches designed by the two communities, thus promoting and strengthening the collaboration between them.

1 Introduction

Recently, several attempts to use machine learning (ML) to deal with combinatorial optimization problems (COPs) have been proposed. Indeed, ML promises to support the design of algorithms as well as the resolution process, by decreasing the need for hand-crafted and specialized solution approaches, which are notably known to require high expertise and a huge time investment for being developed. The incredible advances in deep learning (DP, Goodfellow, Bengio, and Courville, 2016) coupled with increasingly powerful hardware, have led to very promising results for a variety of COPs Bengio, Lodi, and Prouvost, 2021; Mazyavkina et al., 2020.

In this work, we focus on COPs arising in the area of vehicle routing. Vehicle routing problems (VRPs, Toth and Vigo, 2014) are, in fact, increasingly receiving attention in the ML community both because of their relevance in real-world applications and the computational challenges they pose Vesselinova et al., 2020. Indeed, VRPs have been the test-bed for novel neural network architectures such as pointer networks Vinyals, Fortunato, and Jaitly, 2015, graph embedding networks Dai, Dai, and Song, 2016; Khalil et al., 2017 and attention-based models Kool, Hoof, and Welling, 2019; Deudon et al., 2018 achieving stunning results on a variety of problems. Currently, most of the proposed approaches aim at learning constructive heuristics, which sequentially extend a partial solution, possibly employing additional procedures such as sampling and beam search (see e.g., Bello et al., 2017; Hottung, Kwon, and Tierney, 2021). Few others, such as Wu et al., 2020; Chen and Tian, 2019, instead, focus on learning improvement heuristics to guide the exploration of the search space and iteratively refine an existing solution.

Despite of the relevance of the problems, the increasing attention they are receiving in the ML community and the promising results achieved so far, these techniques have not yet become widespread in the Operations Research (OR) community. This may be because of their novelty and the nontrivial effort required to adopt them by the OR community that typically has a different background. But also, at the same time, the computational testing provided by some of the proposed ML methods may not be very convincing with respect to the standard practices in the OR and VRP community. In fact, despite working on the same problems, there is a very limited integration between the experimental results developed by both communities. The current work focuses on this second aspect by highlighting important points to consider during the computational testing and providing guidelines and methodologies that we believe are relevant from an OR perspective. Finally, we conclude by referring the reader to the excellent overview on experimental analysis by Johnson, 1999.

The remainder of the paper is organized as follows. Section 2 surveys crucial aspects faced during a computational testing such as the selection of proper benchmark instances and baseline algorithms, as well as techniques to properly compare different solution approaches. Section 3 provides concrete examples and pointers to representative computational studies found in recent papers. Finally, concluding remarks are given in Section 4.

2 Guidelines for a Computational Testing

In this section, we highlight important points an OR practitioner would consider crucial when examining the computational results section of a novel approach.

2.1 Benchmark Instances and Problem Definition

The training phase of a ML-based approach typically requires a large number of VRP instances sharing the same characteristics. A common way to generate these instances is by randomly defining their characteristics, sampled from arbitrarily defined distributions. In the following, we argue that adopting a common problem description as well as a common set of benchmark instances is extremely important to correctly assess the potentialities of a novel approach.

Problem description and objectives. The term VRP identifies a class of problems containing an enormous number of different variants. Among the most studied ones, we have the Capacitated Vehicle Routing Problem (CVRP), the Vehicle Routing Problem with Time Windows (VRPTW), and the Vehicle Routing Problem with Pickup and Deliveries. In the following, if not specified differently, we limit our treatment to the CVRP which is often taken as a reference problem in ML-based approaches and, despite its simple definition, is still a challenging problem for both traditional and innovative approaches. We refer to Chapter 1 of Toth and Vigo, 2014 for a thorough overview of VRPs. Specific VRPs have widely accepted formulations and precise problem definitions. For example, modern CVRP instances do not fix a-priori the number of routes a solution should have or heuristic approaches to the VRPTW typically consider a hierarchical objective: first minimize the number of vehicles and then the routing cost. It is thus clear that the specific definition of the problem has a crucial impact on the obtained results.

Test instances representativeness. Despite VRPs being $\mathcal{NP}$-hard, the actual challenge posed by a specific instance is highly dependant on several factors such as customers and depot location, the vehicle capacity and the customers demand distribution. The VRP community has thus identified, for each specific problem in the VRP class, a set of benchmark instances that is currently considered to be relevant for testing modern approaches. Thus, in addition to possible instances defined by the ML community, we highlight the importance of using, whenever possible, instances derived from these widely recognized, used and studied datasets.

As an example, in Uchoa et al., 2017, the authors introduced the $\mathbb{X}$ instances for the CVRP, thoroughly describing their generation process. This process can then be emulated to generate (possibly smaller-sized) more representative CVRP instances, as it was done in Kool et al., 2021 and in Hottung, Kwon, and Tierney, 2021 (appendix). In Wu et al., 2020, the authors directly used a subset of $\mathbb{X}$ instances along with distributions more commonly used in other ML works. Finally, because redefining ad-hoc generators may introduce inconsistencies between different studies, we encourage the widespread usage of the instance generator provided by Queiroga et al., 2022 for creating training and validation instances, as well as the usage of the 10,000 $\mathbb{X}$-like instances (also defined in Queiroga et al., 2022) as a common set for testing purposes.

As an additional example, consider the VRPTW for which the so-called Solomon instances and their extension proposed by Solomon, 1987 and Gehring and Homberger, 1999, respectively, are the current benchmark. Also for them, Solomon, 1987 describes the procedure used to define the time window constraints that can thus be considered when defining new instances.

Repositories of instances. Together with papers introducing or using certain sets of instances (whose authors could be contacted to retrieve), we mention some among the most popular repositories of VRP instances. Namely, VRP-REP collects instances for more than 50 different VRP variants, best known solution values, and references to papers obtaining these results. In addition, CVRPLIB contains instances and up-to-date best-known solutions of CVRP instances. Finally, small instances commonly used in the past or in exact methods can be found in TSPLIB and OR-LIBRARY.

2.2 Baseline Algorithms

The selection of a proper baseline algorithm is extremely important, failing in this task would hinder the objective evaluation of the potential of a novel approach. Indeed, since results (computing time, solution quality, and possibly more sophisticated measures as detailed in Section 2.3) are crucial to compare different solution approaches over a common set of problems and instances, a wrong baseline may distort their interpretation, undermining the whole validation process. Despite the purpose of ML-based approaches not being that of outperforming highly specialized solvers, but rather that of proposing versatile tools not requiring high-levels of manual engineering, the comparison should still occur against the best performing algorithms to better comprehend the tradeoff between data-driven and ad-hoc algorithms.

Include the best available algorithms. Along with simple baselines and competing ML-based methods, we argue that one should consider the inclusion of the best available algorithms proposed by the OR community for each specific VRP.

The selection of baseline algorithms is often guided by the availability of free-to-use or open-source reliable software packages. Fortunately, more and more researchers are publishing the source code of heuristic as well as exact state-of-the-art VRP solvers that can be freely used for research activities.

Heuristic solvers The widely used Google OR-Tools Perron and Furnon, 2019 is erroneously considered by most ML papers to be among the best open-source VRP solvers (see, e.g., Nazari et al., 2018) while achieving on the CVRP far-from-optimal results on the $\mathbb{X}$ instances (see Vidal, 2022). Much better open-source solvers for the CVRP are fortunately available. Along with LKH-3 Helsgaun, 2017, which is already widely used by the ML community for solving VRPs, we mention HGS-CVRP Vidal, 2022 and FILO Accorsi and Vigo, 2021 as highly effective and efficient open-source heuristic solvers for the CVRP that, on the widely studied $\mathbb{X}$ instances of the CVRP Uchoa et al., 2017, produce superior results compared with LKH-3 (see Cavaliere, Bendotti, and Fischetti, 2020a). Finally, we mention SISR Christiaens and Vanden Berghe, 2020, that, despite not being already available in terms of source code, is conceptually simple and easy to implement, yet providing state-of-the-art results on a great variety of VRPs.

Exact solvers Several papers solve small instances to optimality by using general-purpose exact optimization solvers such as Gurobi or CPLEX. Despite the noble attempt, trying to directly solve simple compact VRP formulations by using general-purpose solvers would soon turn out to be an extremely challenging task. Indeed, several ML papers report the ability of solving only very small instances (e.g., CVRPs with about 20 customers). Instead, VRPSolver Pessoa et al., 2020, a freely available (for academic purposes) exact solver specialized for routing problems, should be considered for serious and reliable testing of VRPs. Indeed, VRPSolver combines a branch-cut-and-price algorithm with other sophisticated techniques specifically designed for VRPs such as route enumeration, state-space relaxation, and others being able to consistently solve around 95% of randomly generated CVRP instances with up to 100 customers within one hour (a size used in most ML-based approaches proposed so far). Finally, as a general note, we discourage the usage of exact solvers for comparison reasons since their goal of proving optimality is deeply different from that of heuristics that aim at finding good quality solutions within acceptable computing time.

2.3 Algorithms Comparison

Comparing algorithms having a completely different nature is an extremely challenging task requiring a special attention to parameters tuning and hardware setup.

As to parameters tuning, traditional solution approaches proposed by the OR community are designed to handle set of instances having a broad range of different characteristics. Moreover, these approaches are typically tuned to achieve, with a single set of parameter values, results that are on average good over all tested instances. Instead, ML-based approaches generally need to treat every instance distribution separately, requiring a specific tuning and thus additional training (possibly taking up to several weeks of computing time). A fair comparison requires the usage of a single algorithm or model tested over the benchmark instances under examination. Moreover, hyper-parameter tuning should be conducted on a separate dataset. In addition, traditional OR algorithms are (almost) always executed on CPUs using a single thread, while ML-based approaches naturally benefit from running on massively parallel hardware architectures such as GPUs. Finally, the programming language may also play a role. In fact, traditional OR algorithms are usually implemented in highly efficient languages such as C++, whereas ML-based approaches typically use Python that mixes slow interpreted code with efficient native libraries.

Facilitate comparisons (i.e., run all solvers by using a common configuration). In production, algorithms should obviously make fully use of the best existing available technologies. In fact, even traditional algorithms may contain (portions of) embarrassingly parallel code that would thus benefit from being run on multiple threads. As an example, a common dynamic programming procedure employed in VRP exact solvers would greatly benefit from a GPU implementation compared to a traditional sequential version (Boschetti, Maniezzo, and Strappaveccia, 2017 report speedup of up to 40 times). Another well-known example is the parallel implementation of Branch & Bound algorithms (see, e.g., Crainic, Le Cun, and Roucairol, 2006). Finally, many heuristics can be easily parallelized with minimal synchronization overhead, for example by processing solutions in parallel within population-based approaches, or by running in parallel several searches from different starting points. Despite the clear time-savings of a parallel implementation, experimental evaluation asks to reduce at a minimum the different factors that would render comparisons among approaches unnecessarily more challenging. It is thus commonly accepted to consider single-threaded algorithms run on standard CPU architectures.

Parallelization can, however, be employed to speed up parameter tuning and model training. A very interesting information, that would promote a direct comparison of newly proposed algorithms with existing ones, consists in including also the computing time taken by the algorithm when run with the above defined settings. If running all the experiments both on GPU and on CPU with a single thread would be computationally prohibitive, we argue that one should consider adding a measure of the speedup associated with the model inference when run on a GPU rather than on a CPU together with the total algorithmic time (in which everything except the model inference is run on a CPU with a single thread), and the fraction of time spent doing inference on GPU. This way, the total computing time of the algorithm on a CPU with a single thread could be easily estimated. Another possibility sees the inclusion of a rough measure of the number of CPU cores needed to match the GPU capacity as it was done in Hottung, Kwon, and Tierney, 2021.

Support the reproducibility of the study. Open-source algorithmic implementations complemented with scripts that easily allow to reproduce the computational study would provide a great value to the entire scientific community promoting further exploration and extension of a proposed approach. Moreover, we encourage the sharing of detailed results (i.e., the solution quality and computing time for each individual instance solved) for future comparisons and additional analysis. These pieces of information could be stored in a companion table or a website.

Convert computing time to a common scale. When using results published on other papers, because the source code may not be available or running again the experiments may be too time consuming, a common practice consists in roughly scaling the computing time to a common base by using appropriate factors. A practice increasingly adopted in the OR community consists in using the single-thread rating defined by PassMark ®Software. So that, given a base processor, say an Intel Xeon CPU E3-1245 v5 having a single-thread rating of 2277, and a target processor, say an Intel Core2 Duo T5500 having a single-thread rating of 594, the computing time of an algorithm run on the target processor is reduced of a factor ≈ 3.83. The comparison is still rough, being the CPU just one among the several components affecting the overall performance. However, this is one of the possible approaches generally accepted as sound by the OR community. Note that, the above considerations assume all algorithms have been run on an unloaded system and on a CPU by using a single thread.

On the comparison with general-purpose solvers. General-purpose solvers such as Gurobi or CPLEX are often used as baseline algorithms to generate good quality solutions when approaching small-to-medium size instances without necessarily aiming at prove their optimality. When reporting the computing time spent by a solver whose aim is to prove the optimality of a solution, especially if its results are compared with a heuristic algorithm, it should be considered that the former may find very good quality solutions early during the run and then spend the majority of the time to prove optimality. Thus, if one wants to report results obtained by these solvers, an additional column could be added showing the computing time in which the last solution (i.e., the optimal one, if the solver is not prematurely stopped) was found. Despite being true that the solver does not have a termination criterion to know whether the solution at hand is optimal or not, being compared with heuristics, this approach would still provide a feeling on the convergence speed of the solver. We finally mention the primal integral introduced by Achterberg, Berthold, and Hendel, 2012, as a measure able to take into account the overall solution process in terms of convergence towards the optimal (or best known) solution over the entire solving time.

Consider the statistical relevance of the results. In the past, it was common practice to use just average percentage errors with respect to the best known solution to compare the performance of different algorithms. However, more recently, the inclusion of simple statistical tests to objectively assess the differences among algorithms is increasingly becoming popular in the VRP

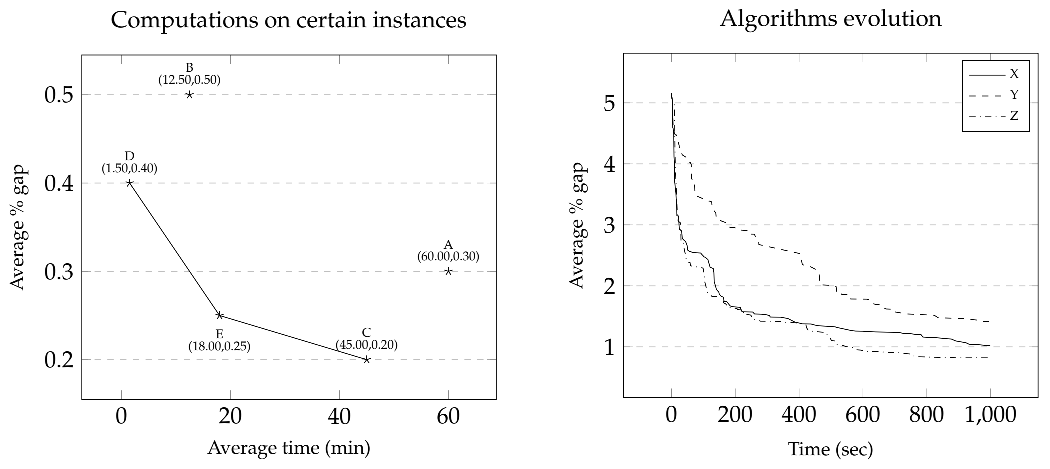


Figure 4.1: On the left side, the performance chart showing for algorithms A, B, C, D, and E the average normalized time and solution quality obtained in a number n of runs with different seeds. Algorithms C, D, and E dominate A and B. On the right side, the search trajectory defined by the average gap found at a given time instant for algorithms X, Y and Z.

community since differences in solution quality between solvers have decreased over time. A common practice consists in using a one-tailed Wilcoxon signed-rank test (see Wilcoxon, 1945) possibly coupled with correction methods when multiple comparisons involving the same data are performed (e.g., the Bonferroni correction, see Dunn, 1961). These tests are used to determine whether two sets of paired observations, for which no assumption can be done on their distribution, are statistically different, and thus, whether two algorithms are considered to provide equivalent results. A common tool for performing these statistical tests is the R language (R Core Team), which allows to execute them with just a few lines of code. An example of a test application can be found in Section 3.3.

Always compare averages. When experimenting with algorithms containing randomized components, we argue that one should make comparison by considering the average results obtained over a reasonable number of runs. Typically, the used number of runs are 10 or 50, that, despite not being statistically significant, allow to qualitatively identify whether an algorithm provides stable or highly variable results.

Always report the complete computing time. A common mistake consists in reporting the computing time spent to find the best found solution instead of the total computing time of the algorithm. The difference between these times can be surprisingly large and the reader may think that additional time would have allowed an improved solution quality. We thus encourage to clearly report the complete computing time.

Include charts. Charts allow to visually compare several algorithms at once, thus providing a fast and effective way to view the results, that, when coupled with tables and statistical tests, give a satisfying and thorough picture of the computational studies for a set of instances. Among the most used charts, we mention

- the performance chart, relating the average normalized computing time with the average solution quality (e.g., the percentage gap with respect to a reference value, which is typically that of the optimal or best known solution) of each algorithm in the comparison, see Figure 4.1 (left);
- the convergence profile chart, showing for each time instant the average solution quality for the compared algorithms, see Figure 4.1 (right).

The performance chart clearly identifies Pareto optimal algorithms and dominated ones. In general, an algorithm dominates another one if it performs better in terms of both computing time and solution quality. The convergence profile chart shows algorithms convergence speed when executed for a specific period of time. Convergence profile charts can be used for improvement heuristics but also for simpler constructive heuristics provided that they include additional iterative procedures used to further improve the solution quality (e.g., the active search of Bello et al., 2017).

On the choice of the programming language. (Baseline) Algorithms should be implemented efficiently. This may include using appropriate data structures as well as programming languages producing directly executable machine code. An efficient implementation of a competing algorithm should not be considered negatively.

Consider the analysis of the various algorithm components. Several papers include additional analysis on the components of the proposed algorithm. This may include the behavior of the algorithm when some parameters are changed as well as the contribution of individual components to the overall final results (e.g., the average improvement of different local search operators analyzed throughout the algorithm execution). This kind of analysis is both considerably appreciated in the VRP community and extremely important to grasp insights on the overall contribution and usefulness of the different components of an algorithm.

3 Examples of a Computational Studies

To make the above suggestions more concrete, in this section, we report extracts of and pointers to representative computational studies found in recent papers on the CVRP. We will consider two scenarios: in the first one, we assume to have all source codes available, whereas in the second one, we assume at least one of the competing algorithm source code is not available to the tester.

3.1 Scenario 1: all algorithms source codes are available

This is the ideal setting, since all algorithms can be run on exactly the same platform and for the same amount of time. In this context the convergence profile chart perfectly shows the evolution of each algorithm, making evident its speed to reach certain quality levels.

Table 4.1 reports a rearranged extract of Tables 1-3 proposed in Vidal, 2022 comparing three of the above mentioned algorithms (more specifically HGS-CVRP, SISR, and OR-Tools) run on a common platform for the same computing time and over the same $\mathbb{X}$ instances of the CVRP. In particular, the table shows for each competing algorithm the average solution value (Avg) and the associated average gap (Gap) computed considering a certain number of runs of the algorithm when it includes randomized components. The gap is computed with respect to a reference best known solution value (BKS) as Gap = $100 \cdot (\text{Avg} - \text{BKS})/\text{BKS}$. Additional useful columns could include the worst and best gap obtained, so as to better examine the variability of the quality for the algorithm on the dataset. Finally, we refer to Figures 3 and 4 of Vidal, 2022 for examples of convergence profile charts for the above-mentioned algorithms.

3.2 Scenario 2: at least one algorithm source code is not available

This scenario, which is still unfortunately frequent in the VRP community, makes the comparison of results much more challenging. The common case sees only the presence of tables showing the performance of a proposed algorithm over a set of benchmark instances when run on a certain platform. First of all, an assumption is required when reviewing these tables reporting

		HGS-CVRP		SISR		OR-Tools	
Instance	BKS	Avg	Gap	Avg	Gap	Avg	Gap
X-n101-k25	27591	27591.0	0.00	27593.3	0.01	27977.2	1.40
X-n106-k14	26362	26381.4	0.07	26380.9	0.07	26757.5	1.50
X-n110-k13	14971	14971.0	0.00	14972.1	0.01	15099.8	0.86
...							
X-n979-k58	118987	119247.5	0.22	119108.2	0.10	123885.2	4.12
X-n1001-k43	72359	72748.8	0.54	72533.1	0.24	78084.7	7.91
Mean			0.11		0.19		4.01

Table 4.1: Minimal table showing the solution quality obtained by algorithms run for the same amount of time on a common platform.

the computational results. In particular, we have to assume that the results published in the paper, which are inevitably obtained with a specific tuning of parameters, are the values on which the authors desire to compare with other approaches. We shall note that these parameters do include the termination criterion, which was then selected as a design choice, and considered to provide competitive results (otherwise a different criterion would have been selected). Since data on the convergence of the algorithm are typically not available, the comparison can then be performed over the published results by first normalizing the computing time in the best possible way (see Section 2.3) and then analyzing the bi-objective perspective provided by the performance chart shown in Figure 4.1 (left) showing non dominated algorithms for a fixed configuration of their parameters.

3.3 Statistical validation of the results

In both the above scenarios (and provided that the average solution quality obtained by an algorithm is available for each instance in the dataset), the (final) solution quality could be assessed with simple statistical tests. This procedure is useful especially when the proposed methods do not clearly dominate the others, for example because they obtain similar average percentage gaps on the considered benchmark instances.

As an example, in the following we compare HGS-CVRP, SISR, and OR-Tools on the $\mathbb{X}$ instances. Data have been taken from Tables 1 and 3 of Vidal, 2022 and is summarized in the boxplots of Figure 4.2.

Similarly to what was done in Christiaens and Vanden Berghe, 2020, we can assess the results

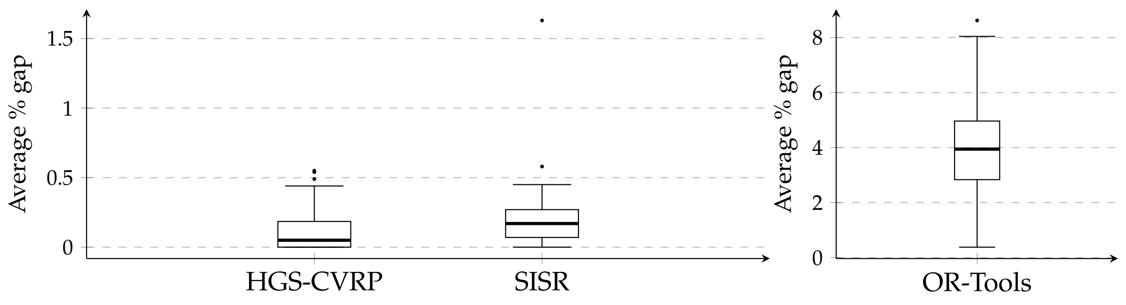


Figure 4.2: Average gaps obtained on the $\mathbb{X}$ instances by HGS-CVRP, SISR and OR-Tools. Note the different y-axis for OR-Tools. The thick line identifies the median value.

	SISR	OR-Tools
H_0	8.27934e-06	3.95591e-18
H_1	4.13967e-06	1.97796e-18

Table 4.2: p-values obtained when comparing HGS-CVRP with SISR and OR-Tools.

obtained by our proposed algorithm, say HGS-CVRP, against the remaining ones, by conducting a one-tailed Wilcoxon signed-rank test in which we consider a null hypothesis H_0

$$H_0 : \textsc{AvgSolCost}(HGS - CVRP) = \textsc{AvgSolCost}(X),$$

and an alternative hypothesis H_1

$$H_1 : \textsc{AvgSolCost}(HGS - CVRP) > \textsc{AvgSolCost}(X),$$

where X can be SISR and OR-Tools. A hypothesis is rejected when its p-value is lower than a significance level α. In particular, we have that

- failing to reject H_0 means that the average results of the two methods are not statistically different;
- whereas, when H_0 is not rejected, the average results are statistically different and the alternative hypothesis H_1 can be tested to find whether they are greater than those of a competing method. Rejecting H_1 thus implies that HGS-CVRP performs better than the competing method.

Moreover, as mentioned in Section 2.3, 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 adjusted to lower values. Bonferroni correction (Dunn, 1961) is a simple method that can be used for this purpose. In particular, given n comparisons, the significance level is set to α/n.

In the above comparison we tested a total number of $n = 2$ hypothesis corresponding to the partitioning of instances (1, all $\mathbb{X}$ instances together) and to the two hypothesis (2, H_0 and H_1). Assuming an initial significance level $\alpha_0 = 0.025$, the adjusted value becomes $\alpha = 0.025/2 = 0.0125$. In general, given n_p, the number of partitions of the instances (e.g., $n_p = 3$ if we split a single dataset into three subsets containing, say, small, medium and large instances), and n_h, the number of hypothesis we want to test for every partition, we have $n = n_p \cdot n_h$.

We can compute the p-values, which are shown in Table 4.2, for our analysis for example by using the R language. For both SISR and OR-Tools we have that the associated p-value is lower than the significance level α. We can thus reject the hypothesis H_0 that the average results of HGS-CVRP are similar to those of SISR and OR-Tools. Finally, by testing H_1 we again are able to reject the hypothesis, concluding that HGS-CVRP average results are statistically better than those obtained by SISR and OR-Tools on the $\mathbb{X}$ instances.

4 Conclusions

With this work we highlighted several challenges arising when comparing traditional and machine learning-based solution approaches for vehicle routing problems. Our aim was that of providing a set of reference guidelines that would help the machine learning community to produce computational studies that would be better appreciated by the operations research, and especially the vehicle routing, communities. We believe that cross fertilization of algorithmic techniques from ML and OR define very promising and potentially game-changing avenues for effective solutions of routing problems. However, we believe that such a cross fertilization needs to grow on a shared computational understanding. We hope that this paper is a step forward in this direction.

5 Acknowledgments

We gratefully thank a referee for its valuable comments and suggestions that greatly improved the quality of the manuscript.

Chapter 5

Conclusions and Future Directions

In this thesis we studied techniques for the effective and efficient resolution of Vehicle Routing Problems (VRPs). These techniques, when embedded into carefully designed heuristic solution approaches, allowed to achieve state-of-the-art quality solutions in a relatively short computing time for several well known classical problems.

In Chapter 2 we proposed a metaheuristic called FILO for the solution of large-scale instances of the Capacitated Vehicle Routing Problem (CVRP). The main ingredient of FILO is a sophisticated local search engine using several acceleration techniques that allow for the application of several of local search operators, including complex ones such as the ejection chain, without any substantial speed penalty. FILO is, in fact, able to achieve solutions of similar quality of existing state-of-the-art algorithms but in a fraction of the running time. An interesting future research direction consists in studying whether the acceleration techniques used in FILO could be extended to handle VRPs including more realistic constraints, such as the VRP with Pickup and Delivery, the VRP with Time Windows and the Heterogeneous VRP.

In Chapter 3 we study the Extended Single Truck and Trailer Routing Problem (XSTTRP). The XSTTRP is a general problem arising from the combination of several existing well known problems such as the Multiple Depot Vehicle Routing Problem, the Location Routing Problem, the Single Truck and Trailer Routing Problem with Satellite Depots, and the Single Truck and Trailer Routing Problem. More precisely, the XSTTRP models a broad class of VRPs that use a single vehicle, composed of a truck and a detachable trailer, to serve a set of customers with known demand and accessibility constraints ruling whether certain customers can be visited by the whole vehicle or just by the truck. To solve the XSTTRP we proposed a matheuristic called AVXS which combines an iterative core part based on a multi-start Iterated Local Search, in which routes that define high-quality solutions are stored in a pool. Eventually, a set-partitioning based post-optimization selects the best combination of routes that forms a feasible solution from the pool. Algorithm AVXS shown to be competitive with existing state-of-the-art solution approaches that were design to solve specialization of the XSTTRP. The natural continuation of this work consists in the extension of algorithm AVXS to handle multiple vehicles. Moreover, some techniques introduced in Chapter 2 (whose realization happened chronologically after the design of AVXS) may be extended to the XSTTRP to possibly further speed up the resolution process.

In Chapter 4 we highlighted a number of practices which are common way of proceeding in the computational evaluation of VRP papers produced by the Operations Research community. In fact, recently, due to the incredible advances in the Machine Learning (ML) techniques as well as in the hardware technology, several algorithms have been proposed by the ML to solve VRPs. However, despite the remarkable results obtained by this algorithms, ML techniques are still seldom employed by the OR community. In our opinion, this may be based on the different approaches to the computational evaluation of the proposed methods. In this chapter we provide some guidelines that will hopefully help promoting the collaboration between the OR and ML communities. In this regard, an interesting research direction we have already started exploring, consists in extending FILO to include a simple classifier to identify routes of the current solution that would most benefit from a re-optimization. This classifier could be used to bias the selection of the area to disrupt during the shaking procedure. This direction opens up a number of challenges that we think are worth investigating.

Bibliography

Accorsi, Luca (2017). "Validi Lower Bound al Single Truck and Trailer Routing Problem". MA thesis. Università di Bologna.

Accorsi, Luca and Daniele Vigo (2020). "A Hybrid Metaheuristic for Single Truck and Trailer Routing Problems". In: *Transportation Science* 54.5, pp. 1351–1371.

— (2021). "A Fast and Scalable Heuristic for the Solution of Large-Scale Capacitated Vehicle Routing Problems". In: *Transportation Science* 55.4, pp. 832–856.

Achterberg, Tobias, Timo Berthold, and Gregor Hendel (2012). "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, pp. 71–76.

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

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

Baldacci, Roberto and Aristide Mingozzi (2008). "A unified exact method for solving different classes of vehicle routing problems". In: *Mathematical Programming* 120.2, p. 347.

Baldacci, Roberto, Aristide Mingozzi, and Roberto Wolfler Calvo (2011). "An Exact Method for the Capacitated Location-Routing Problem". In: *Operations Research* 59.5, pp. 1284–1296.

Baldacci, Roberto, Paolo Toth, and Daniele Vigo (2007). "Recent advances in vehicle routing exact algorithms". In: *4OR* 5.4, pp. 269–298.

Bartolini, E. and M. Schneider (2018). "A two-commodity flow formulation for the capacitated truck-and-trailer routing problem". In: *Discrete Applied Mathematics*.

Beek, Onne et al. (2018). "An Efficient Implementation of a Static Move Descriptor-based Local Search Heuristic". In: *Computers & Operations Research* 94, pp. 1 –10.

Belenguer, José-Manuel et al. (2016). "A Branch-and-Cut Algorithm for the Single Truck and Trailer Routing Problem with Satellite Depots". In: *Transportation Science* 50.2, pp. 735–749.

Bello, Irwan et al. (2017). *Neural Combinatorial Optimization with Reinforcement Learning*.

Bengio, Yoshua, Andrea Lodi, and Antoine Prouvost (2021). "Machine learning for combinatorial optimization: A methodological tour dhorizon". In: *European Journal of Operational Research* 290.2, pp. 405–421.

Bergstra, James and Yoshua Bengio (2012). "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13.10, pp. 281–305.

Bodin, L. and L. Levy (2000). "Scheduling of local delivery carrier routes for the united states postal service". In: *In M. Dror, editor, Arc Routing: Theory, Solutions and Applications*, pp. 419–442.

Boschetti, Marco A. et al. (2009). "Matheuristics: Optimization, Simulation and Control". In: *Hybrid Metaheuristics*. Ed. by María J. Blesa et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 171–177.

Boschetti, Marco Antonio, Vittorio Maniezzo, and Francesco Strappaveccia (2017). "Route relaxations on GPU for vehicle routing problems". In: *European Journal of Operational Research* 258.2, pp. 456–466.

Caramia, Massimiliano and Francesca Guerriero (2010). “A Milk Collection Problem with Incompatibility Constraints”. In: *Interfaces* 40.2, pp. 130–143.

Cavaliere, Francesco, Emilio Bendotti, and Matteo Fischetti (2020a). *An integrated local-search/set-partitioning heuristic for the Capacitated Vehicle Routing Problem*. Tech. rep. University of Padova.

— (2020b). “Personal communication”.

Chao, I-Ming (2002). “A tabu search method for the truck and trailer routing problem”. In: *Computers & OR* 29.1, pp. 33–51.

Chen, Xinyun and Yuandong Tian (2019). “Learning to Perform Local Rewriting for Combinatorial Optimization”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc.

Christiaens, Jan and Greet Vanden Berghe (2020). “Slack Induction by String Removals for Vehicle Routing Problems”. In: *Transportation Science* 54.2, pp. 417–433.

Clarke, G. and J. W. Wright (1964a). “Scheduling of Vehicles from a Central Depot to a Number of Delivery Points”. In: *Operations Research* 12.4, pp. 568–581.

— (1964b). “Scheduling of Vehicles from a Central Depot to a Number of Delivery Points”. In: *Operations Research* 12.4, pp. 568–581.

Contardo, Claudio, Jean-François Cordeau, and Bernard Gendron (2014). “A GRASP + ILP-based metaheuristic for the capacitated location-routing problem”. In: *Journal of Heuristics* 20.1, pp. 1–38.

Cordeau, Jean-François, Michel Gendreau, and Gilbert Laporte (1997). “A tabu search heuristic for periodic and multi-depot vehicle routing problems”. In: *Networks* 30.2, pp. 105–119.

CPLEX (2017). “IBM ILOG CPLEX Optimizer 12.8 Callable Library”. In.

Crainic, Teodor Gabriel, Bertrand Le Cun, and Catherine Roucairol (2006). “Parallel Branch-and-Bound Algorithms”. In: *Parallel Combinatorial Optimization*. John Wiley & Sons, Ltd. Chap. 1, pp. 1–28.

Cuda, R., Gianfranco Guastaroba, and Maria Grazia Speranza (2015). “A survey on two-echelon routing problems”. In: *Computers & OR* 55, pp. 185–199.

CVRPLIB (2020). *Capacitated Vehicle Routing Problem Library*. visited on 2020-07-19.

Dai, Hanjun, Bo Dai, and Le Song (2016). “Discriminative Embeddings of Latent Variable Models for Structured Data”. In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML’16. New York, NY, USA: JMLR.org, pp. 2702–2711.

Dantzig, G. B. and J. H. Ramser (1959). “The Truck Dispatching Problem”. In: *Management Science* 6.1, pp. 80–91.

Deudon, Michel et al. (2018). “Learning Heuristics for the TSP by Policy Gradient”. In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Ed. by Willem-Jan van Hoeve. Cham: Springer International Publishing, pp. 170–181.

Drexl, Michael (2011). “Branch-and-price and heuristic column generation for the generalized truck-and-trailer routing problem”. In: *Journal of Quantitative Methods for Economics and Business Administration* 12, pp. 5–38.

— (2014). “Branch-and-cut algorithms for the vehicle routing problem with trailers and transshipments”. In: *Networks* 63.1, pp. 119–133.

Duarte, Abraham et al. (2018). “Variable Neighborhood Descent”. In: *Handbook of Heuristics*. Ed. by Rafael Martí, Panos M. Pardalos, and Mauricio G. C. Resende. Cham: Springer International Publishing, pp. 341–367.

Duhamel, Christophe et al. (Nov. 2010). “A GRASP$\times$ELS Approach for the Capacitated Location-routing Problem”. In: *Comput. Oper. Res.* 37.11, pp. 1912–1923.

Dunn, Olive Jean (1961). “Multiple Comparisons Among Means”. In: *Journal of the American Statistical Association* 56.293, pp. 52–64.

Escobar, John Willmer, Rodrigo Linfati, and Paolo Toth (2013). "A two-phase hybrid heuristic algorithm for the capacitated location-routing problem". In: *Computers & Operations Research* 40.1, pp. 70 –79.

Escobar, John Willmer et al. (2014). "A Granular Variable Tabu Neighborhood Search for the capacitated location-routing problem". In: *Transportation Research Part B: Methodological* 67, pp. 344 –356.

Etiemble, Daniel (2018). *45-year CPU evolution: one law and two equations*.

Florian, Arnold (2018). "Efficient Heuristics for Routing and Integrated Logistics". PhD thesis. University of Antwerp, Faculty of Applied Economics.

Frank, Eibe, Mark A. Hall, and Ian H. Witten (2016). *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*. 4th ed. Morgan Kaufmann.

Gehring, Hermann and Jörg Homberger (1999). "A parallel hybrid evolutionary metaheuristic for the vehicle routing problem with time windows". In: *Proceedings of EUROGEN99*. Vol. 2. Citeseer, pp. 57–64.

Gerdessen, JC. (1996). "Vehicle routing problem with trailers". In: *European Journal of Operations Research* 93, pp. 135–147.

Glover, Fred (1989). "Tabu SearchPart I". In: *ORSA Journal on Computing* 1.3, pp. 190–206.

— (1996). "Ejection chains, reference structures and alternating path methods for traveling salesman problems". In: *Discrete Applied Mathematics* 65.1. First International Colloquium on Graphs and Optimization, pp. 223 –253.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Grasas, Alex et al. (2017). "Biased randomization of heuristics using skewed probability distributions: A survey and some applications". In: *Computers & Industrial Engineering* 110, pp. 216 –228.

Gutin, Gregory and Abraham P Punnen (2006). *The traveling salesman problem and its variations*. Vol. 12. Springer Science & Business Media.

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

Hemmelmayr, Vera C., Jean-François Cordeau, and Teodor Gabriel Crainic (2012). "An adaptive large neighborhood search heuristic for Two-Echelon Vehicle Routing Problems arising in city logistics". In: *Computers & Operations Research* 39.12, pp. 3215 –3228.

Hottung, André, Yeong-Dae Kwon, and Kevin Tierney (2021). *Efficient Active Search for Combinatorial Optimization Problems*.

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

Johnson, David S. (1999). "A theoretician's guide to the experimental analysis of algorithms". In: *Data Structures, Near Neighbor Searches, and Methodology*.

Jünger, Michael, Gerhard Reinelt, and Giovanni Rinaldi (1995). "Chapter 4 The traveling salesman problem". In: *Network Models*. Vol. 7. Handbooks in Operations Research and Management Science. Elsevier, pp. 225–330.

Khalil, Elias et al. (2017). "Learning Combinatorial Optimization Algorithms over Graphs". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc.

Kirkpatrick, S., C. D. Gelatt, and M. P. Vecchi (1983). "Optimization by Simulated Annealing". In: *Science* 220.4598, pp. 671–680.

Kool, Wouter, Herke van Hoof, and Max Welling (2019). "Attention, Learn to Solve Routing Problems!" In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Kool, Wouter et al. (2021). *Deep Policy Dynamic Programming for Vehicle Routing Problems*.

Kytöjoki, Jari et al. (2007). “An efficient variable neighborhood search heuristic for very large scale vehicle routing problems”. In: *Computers & Operations Research* 34.9, pp. 2743 –2757.

Lin, Shih-Wei, Vincent F. Yu, and Shuo-Yan Chou (2010). “A note on the truck and trailer routing problem”. In: *Expert Systems with Applications* 37.1, pp. 899 –903.

Lopes, Rui Borges, Carlos Ferreira, and Beatriz Sousa Santos (2016). “A simple and effective evolutionary algorithm for the capacitated locationrouting problem”. In: *Computers & Operations Research* 70, pp. 155 –162.

Lourenço, Helena R., Olivier C. Martin, and Thomas Stützle (2003). “Iterated Local Search”. In: *Handbook of Metaheuristics*. Boston, MA: Springer US, pp. 320–353.

Maniezzo, Vittorio, Thomas Stützle, and Stefan Voß, eds. (2010). *Matheuristics - Hybridizing Metaheuristics and Mathematical Programming*. Vol. 10. Annals of Information Systems. Springer.

Martello, Silvano and Paolo Toth (1990). *Knapsack Problems: Algorithms and Computer Implementations*. USA: John Wiley & Sons, Inc.

Matsumoto, Makoto and Takuji Nishimura (Jan. 1998). “Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator”. In: *ACM Trans. Model. Comput. Simul.* 8.1, pp. 3–30.

Mazyavkina, Nina et al. (2020). *Reinforcement Learning for Combinatorial Optimization: A Survey*.

Mladenovi, N. and P. Hansen (1997). “Variable neighborhood search”. In: *Computers & Operations Research* 24.11, pp. 1097 –1100.

Nazari, Mohammadreza et al. (2018). *Reinforcement Learning for Solving the Vehicle Routing Problem*.

OR-LIBRARY (2021). *OR-Library is a collection of test data sets for a variety of Operations Research (OR) problems.* visited on 2021-07-03.

PassMark ®Software (2020). *Professional CPU benchmarks*. visited on 2020-07-19.

Pecin, Diego et al. (2014). “Improved Branch-Cut-and-Price for Capacitated Vehicle Routing”. In: *Integer Programming and Combinatorial Optimization*. Ed. by Jon Lee and Jens Vygen. Cham: Springer International Publishing, pp. 393–403.

— (2017). “Improved branch-cut-and-price for capacitated vehicle routing”. In: *Mathematical Programming Computation* 9.1, pp. 61–100.

Perron, Laurent and Vincent Furnon (July 19, 2019). *OR-Tools*. Version 7.2. Google.

Pessoa, Artur et al. (2020). “A generic exact solver for vehicle routing and related problems”. In: *Mathematical Programming* 183.1, pp. 483–523.

Queiroga, Eduardo, Ruslan Sadykov, and Eduardo Uchoa (2020). *A modern POPMUSIC matheuristic for the capacitated vehicle routing problem*. Tech. rep. L-2020-2. Niterói, Brazil: Cadernos do LOGIS-UFF, p. 28.

Queiroga, Eduardo et al. (2022). *10,000 optimal CVRP solutions for testing machine learning based heuristics*.

R Core Team (2020). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria.

Santini, Alberto et al. (2021). *Decomposition strategies for vehicle routing heuristics*.

Schneider, Michael and Michael Drexl (2017). “A survey of the standard location-routing problem”. In: *Annals of Operations Research* 259.1, pp. 389–414.

Schneider, Michael and Maximilian Löffler (2019). “Large Composite Neighborhoods for the Capacitated Location-Routing Problem”. In: *Transportation Science* 53.1, pp. 301–318.

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

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

— (Apr. 2000b). “Record Breaking Optimization Results Using the Ruin and Recreate Principle”. In: *J. Comput. Phys.* 159.2, pp. 139–171.

Semet, F. (1995). "A two-phase algorithm for the partial accessibility constrained vehicle routing". In: *Annals of Operations Research* 45.11, pp. 1233–1246.

Semet, Frédéric and Éric D. Taillard (1993). "Solving real-life vehicle routing problems efficiently using tabu search". In: *Annals OR* 41.4, pp. 469–488.

Solomon, Marius M. (1987). "Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints". In: *Operations Research* 35.2, pp. 254–265.

Subramanian, Anand, Eduardo Uchoa, and Luiz Satoru Ochi (2013). "A hybrid algorithm for a class of vehicle routing problems". In: *Computers & OR* 40.10, pp. 2519 –2531.

Taillard, Éric et al. (1997a). "A Tabu Search Heuristic for the Vehicle Routing Problem with Soft Time Windows". In: *Transportation Science* 31.2, pp. 170–186.

Taillard, Éric et al. (1997b). "A Tabu Search Heuristic for the Vehicle Routing Problem with Soft Time Windows". In: *Transportation Science* 31.2, pp. 170–186.

Taillard, Éric D. and Stefan Voss (2002). "Popmusic — Partial Optimization Metaheuristic under Special Intensification Conditions". In: *Essays and Surveys in Metaheuristics*. Boston, MA: Springer US, pp. 613–629.

Talbi, El-Ghazali (2009). *Metaheuristics: from design to implementation*. Vol. 74. John Wiley & Sons.

Tan, Kay Chen, Y. H. Chew, and L. H. Lee (2006). "A hybrid multi-objective evolutionary algorithm for solving truck and trailer vehicle routing problems". In: *European Journal of Operational Research* 172.3, pp. 855–885.

Ting, Ching-Jung and Chia-Ho Chen (2013). "A multiple ant colony optimization algorithm for the capacitated location routing problem". In: *International Journal of Production Economics* 141.1. Meta-heuristics for manufacturing scheduling and logistics problems, pp. 34 –44.

Toth, P. and D. Vigo (2002). "Branch-and-Bound Algorithms for the Capacitated VRP". In: *The Vehicle Routing Problem*. Ed. by P. Toth and D. Vigo. Monographs on Discrete Mathematics and Applications. Philadelpia, PA: S.I.A.M., pp. 29–51.

Toth, Paolo and Andrea Tramontani (2008). "An Integer Linear Programming Local Search for Capacitated Vehicle Routing Problems". In: *The Vehicle Routing Problem: Latest Advances and New Challenges*. Ed. by Bruce Golden, S. Raghavan, and Edward Wasil. Boston, MA: Springer US, pp. 275–295.

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

— (2014). *Vehicle Routing*. Ed. by Daniele Vigo and Paolo Toth. Philadelphia, PA: Society for Industrial and Applied Mathematics.

TSPLIB (2021). *TSPLIB is a library of sample instances for the TSP (and related problems) from various sources and of various types.* visited on 2021-07-03.

Tuzun, Dilek and Laura I. Burke (1999). "A two-phase tabu search approach to the location routing problem". In: *European Journal of Operational Research* 116.1, pp. 87 –99.

Uchoa, Eduardo (2020). "Personal communication".

Uchoa, Eduardo et al. (2017). "New benchmark instances for the Capacitated Vehicle Routing Problem". In: *European Journal of Operational Research* 257.3, pp. 845 –858.

Vesselinova, Natalia et al. (2020). "Learning Combinatorial Optimization on Graphs: A Survey With Applications to Networking". In: *IEEE Access* 8, pp. 120388120416.

Vidal, Thibaut (2022). "Hybrid genetic search for the CVRP: Open-source implementation and SWAP* neighborhood". In: *Computers & Operations Research* 140, p. 105643.

Vidal, Thibaut et al. (2012). "A Hybrid Genetic Algorithm for Multidepot and Periodic Vehicle Routing Problems". In: *Operations Research* 60.3, pp. 611–624.

Vidal, Thibaut et al. (2013). "Heuristics for multi-attribute vehicle routing problems: A survey and synthesis". In: *European Journal of Operational Research* 231.1, pp. 1–21.

Villegas, Juan G. et al. (2010). "GRASP/VND and multi-start evolutionary local search for the single truck and trailer routing problem with satellite depots". In: *Eng. Appl. of AI* 23.5, pp. 780–794.

Villegas, Juan G. et al. (2011). "A GRASP with evolutionary path relinking for the truck and trailer routing problem". In: *Computers & OR* 38.9, pp. 1319–1334.

— (2013). "A matheuristic for the truck and trailer routing problem". In: *European Journal of Operational Research* 230.2, pp. 231–244.

Vinyals, Oriol, Meire Fortunato, and Navdeep Jaitly (2015). "Pointer Networks". In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes et al. Vol. 28. Curran Associates, Inc.

Voudouris, Christos and Edward Tsang (1999). "Guided local search and its application to the traveling salesman problem". In: *European Journal of Operational Research* 113.2, pp. 469 –499.

VRP-REP (2021). *Collaborative open-data platform for sharing vehicle routing problem benchmark instances and solutions*. visited on 2021-06-11.

Wilcoxon, Frank (1945). "Individual Comparisons by Ranking Methods". In: *Biometrics Bulletin* 1.6, pp. 80–83.

Wu, Yaoxin et al. (2020). *Learning Improvement Heuristics for Solving Routing Problems*.

Yu, Vincent F. et al. (2010). "A simulated annealing heuristic for the capacitated location routing problem". In: *Computers & Industrial Engineering* 58.2, pp. 288 –299.

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