Mathematical Programming Computation manuscript No.
(will be inserted by the editor)

An integrated local-search/set-partitioning refinement heuristic for the Capacitated Vehicle Routing Problem

Francesco Cavaliere · Emilio Bendotti · Matteo Fischetti

June 14, 2022

Abstract In this paper, an effective heuristic algorithm for large-scale instances of the Capacitated Vehicle Routing Problem is proposed. The technique consists in a local search method entangled with a restricted Set Partitioning problem optimization. Helsgaun's LKH-3 algorithm has been used for the local search phase, with a number of implementation improvements. The restricted Set Partitioning formulation is solved by means of an exact commercial Integer Liner Programming solver. The resulting algorithm is able to consistently improve the solutions obtained by a state-of-the-art heuristic from the literature, as well as some of the best-know solutions maintained by the CVRPLIB website.

Keywords Capacitated Vehicle Routing Problem, Heuristics, Local Search, Set Partitioning, Computational Analysis.

1 Introduction

Firstly introduced by Dantzig and Ramser [10], Vehicle Routing Problems (VRPs) are a class of problems calling for a minimum-cost set of vehicle routes to serve a given set of customers with known demands.

The Capacitated Vehicle Routing Problem (CVRP) is one of the most studied VRP versions, in which the transportation request consists of the distribution of goods from a single depot to a set of customers using homogeneous vehicles with a limited capacity. In the symmetric case, it can be defined on a complete undirected graph $G = (V, E)$ with edge costs c_e's and a special depot node d. Each customer

Francesco Cavaliere
Department of Electrical, Electronic, Information Engineering "Guglielmo Marconi" - DEI, University of Bologna, viale Risorgimento 2, 40136 Bologna, Italy, E-mail: f.cavaliere@unibo.it

Emilio Bendotti
AzzurroDigitale, via della Croce Rossa 42, 35129 Padova, Italy, E-mail: emilio.bendotti@azzurrodigitale.com

Matteo Fischetti
Department of Information Engineering, University of Padua, via Gradenigo 6/A, 35100 Padova, Italy, E-mail: matteo.fischetti@unipd.it

node $i \in N = V \setminus \{d\}$ is characterized by its demand $q_i \geq 0$ which represents the amount of goods requested, while each vehicle route must start and finish at d and has to visit a set of customers whose total demand does not exceed a given capacity C. The overall number of vehicles to be used is often fixed in advance.

Historically, many mathematical formulations have been proposed for this problem [26, 41]. Particularly relevant for our work is the so-called Set-Partitioning (SP) formulation, common to many other VRP variants. In the SP formulation, the objective is to find the best combination of feasible routes that partitions the customer nodes of the graph, minimizing the overall cost, i.e.:

$$\min \sum_{p \in \Omega} c_p \theta_p \tag{1a}$$

$$\sum_{p \in \Omega} \theta_p = k \tag{1b}$$

$$\sum_{p \in \Omega_i} \theta_p = 1, \quad \forall i \in N \tag{1c}$$

$$\theta_p \in \{0, 1\}, \quad p \in \Omega \tag{1d}$$

where Ω is the set of feasible routes for the CVRP, c_p is the cost associated to each route $p \in \Omega$, $\Omega_i \subset \Omega$ is the subset of routes that visit the customer $i \in N$, k is the required number of routes, and θ_p is a binary variable which is 1 if the route p is in the optimal solution, 0 otherwise.

An important aspect of the SP formulation is its generality, as it easily extends to all VRP variants where the additional constraints only affect the feasibility of the routes, hence they are implicitly represented by the route set Ω. However, a main drawback is represented by the cardinality of Ω, which grows exponentially with the number of customers. To tackle this issue, only a subset of potentially-relevant routes is explicitly generated, and optimization techniques like Column Generation [11, 14] or Branch and Price [17, 31] are used. Within these schemes, a Restricted SP (RSP) formulation is iteratively solved, containing only a subset of routes.

Although several advanced mathematical programming decomposition algorithms have been proposed in the last few decades, only relatively small instances—containing only few hundred customers—have been solved to optimality [41]. Problems encountered in real-life scenarios are often substantially larger, thus efficient heuristic algorithms are the only option available to obtain good-quality solutions within acceptable computing times.

The aim of our paper is to design a powerful (yet time consuming) refinement heuristic which is able to improve top-quality solutions. Thus, our method is meant to be used on top of a state-of-the-art heuristic, more than to replace it. This is very much is the spirit of other refinement heuristics from the literature, whose quality is certified by the capability of improving state-of-the-art solutions in a final post-processing step.

The paper is organized as follows. Previous literature on CVRP heuristics is sketched in Section 2. In Section 3, the comprehensive strategy of our algorithm is described, along with the modifications and improvements applied. Extensive computational results are reported in Section 4, showing that our method is able to

consistently improve the solutions obtained by a state-of-the-art heuristic from the literature, as well as some of the best-know solutions maintained in the CVRPLIB website [30]. Some conclusions are finally drawn is Section 5.

2 Previous work

A brief outline of the CVRP heuristics that are most relevant for our work follows.

Helsgaun's [20, 21] heuristic, LKH-3 (whose code can be found in the dedicated website [18]), is a penalty-based extension of the famous Lin and Kerninghan [27] heuristic (LK), able to tackle many VRP variants. Although less efficient with respect to other state-of-the-art CVRP heuristics, LKH-3 (from now on, just LKH) plays a prominent role in our work in that it is the building block of our local-search phase, so we next give a brief description of this method.

Originally designed for the Traveling Salesperson Problem (TSP), the LKH algorithm is based on the concept of r-Opt moves and r-optimality. In a r-Opt move, r edges from the current solution are replaced by other r edges in such a way that another solution is obtained [20]. A solution is said to be r-optimal if it is impossible to obtain a shorter tour by means of any r-Opt move [20]. It is also intuitive that, for $0 \leq r' \leq r$, an r-optimal tour is also r'-optimal, and for a tour of n city to be optimal, it must also be n-optimal. Furthermore, is also reasonable that the probability for a r-optimal tour to be optimal grows with r [20]. However, the number of possible r-Opt moves grows rapidly with the number of nodes of the graph, making it impossible to fully explore the available moves for large values of r. For this reason, r is usually set to 2 or 3, as the algorithm rapidly loses efficiency for larger numbers. To overcome this limit, the LK heuristic introduces a scheme where the r value is decided at run-time, iteration after iteration. Initially, r is set to 2, its minimal value, and then it is gradually increased searching for new potential pairs with the following rationale: starting from the most "out-of-place" pair, the algorithm iterates searching for the new most "out-of-place" pairs of the remaining set, repeating the search multiple times [27]. If an improvement is found, the search restarts from scratch, while it stops otherwise. For further information, the reader can refer to [20] for a brief explanation, or to the original Lin and Kernighan's paper [27].

Vidal et al. [43] propose HGS, a hybrid genetic algorithm combining the effectiveness of their population based method with the Local-Search exploration of neighborhoods defined from a set of operators.

Arnold and Sörensen's [5] knowledge-guided local search (KGLS) is an effective Local-Search heuristic which adopts three different neighborhood-defining operators along with a knowledge based penalization to avoid local optima.

Christiaens and Vanden Berghe [8] develop a simple yet effective algorithm named *Slack Induction by String Removals* (SISR), consisting in a ruin-and-recreate local search heuristic.

In their recent work, Accorsi and Vigo [2] propose FILO, a very efficient and effective iterated local search heuristic, which through the combination of acceleration and localization techniques is able to find state-of-the-art solutions for very large scale CVRP instances in a short computing time. The algorithm adopts a large number of operator-defined neighborhoods and a combination of a ruin-and-recreate scheme coupled with simulated annealing.

Sharing some similarities with the work presented in the present paper, Subramanian et al. [39] propose *Iterated Local Search with Set Partitioning* (ILS-SP), a hybrid algorithm merging the effectiveness of a competitive iterated local search heuristic along with the optimization a SP formulation that tries to heuristically find the best combination of the explored routes. The adoption of a SP optimization phase has been also studied for many other heuristic techniques, as in the works of Foster et al. [15], Ryan et al. [38], Rochat et al. [36], Kelly et al. [24], De Franceschi et al. [12], or Monaci and Toth [29] for the Bin-Packing Problem.

Finally, Queiroga et al. [35] propose a heuristic working as a refinement technique to improve the solution obtained by other heuristics. Exploring a large solution neighborhood, their algorithm is able to consistently improve near-optimal solutions. The adopted technique is POPMUSIC [40], a matheuristic [13] based on the VRPSolver [32, 33] exact solver for VRPs.

3 Algorithm Outline

The overall scheme of our approach can be subdivided into three main phases.

1. The LKH heuristics is executed, in parallel; from the solutions generated at the end of each "trial" of the core LK algorithm, routes are extracted to populate a pool (called the "route pool").
2. Considering the Linear Programming (LP) relaxation of the SP formulation, a column-generation pricing procedure is applied to "filter" the most meaningful routes from the pool.
3. The RSP formulation, considering only the selected routes, is solved with a given time limit.

The three phases above are iterated until a global time limit expires—or a maximum number of repetitions is reached.

The described algorithm has been called *Local Search - Column Generation Heuristic* (LS-CGH) since it uses the LKH heuristic to generate good candidate routes that are then fed to the RSP optimization.

To better differentiate between the different types of iterations (one nested into the other), the following terms will be used:

– in accordance with the naming adopted by the LKH algorithm, the term "trial" refers to a single pass of the core Lin-Kerninghan algorithm, ending when no more improving r-Opt moves can be found.
– A "run" is a set of successive "trials", each starting from the perturbed solution of the previous one.
– The sequence of a single execution of LKH, followed by Column Generation filtering and the RSP optimization, has been named "round".

Our LS-CGH algorithm then consists in a number of "rounds", repeating the three-phase scheme multiple times. Each round is linked to the next one as it exploits the best solution found as its initial solution, and also because the route pool is maintained between rounds.

A high-level representation of the three main phases of the algorithm is given in Algorithm 1. In the pseudocode, the following functions are used:

– LKH: Calls the LKH-based heuristic described in Section 3.1 and in Algorithm 2. Returns the best solution found by the algorithm (S), along with a populated route pool (P).
– CGFilter: Applies the column-generation inspired filtering (described in Section 3.2) to the route pool.
– SolveRSP: Solves the restricted Set Partitioning formulation with a black-box Integer Linear Programming (ILP) solver; see Section 3.3.

Algorithm 1: High-level pseudocode for the LS-CGH algorithm.

```
  Input  : Initial solution S.
  Output: The best solution found.
1 FUNCTION LS-CGH(S)
2 begin
3 |  for Round ← 1 to n_Rounds do
4 |  |  S, P ←LKH(S);
5 |  |  P′ ← CGFilter(P);
6 |  |  S ← SolveRSP(P′, S)
7 |  end
8 |  return S;
9 end
```

3.1 Phase 1: Lin, Kernighan and Helsgaun Heuristic

To integrate the LKH algorithm with our LS-CGH scheme—which has been implemented as multi-thread C++ project—and also to improve its efficiency, a number of customizations have been applied to the original Helsgaun's code available at [18]. A summary of the most relevant changes are reported next.

– Due to the extensive use of global variables and non-reentrant primitives in the *C* code, the algorithm was not "out-of-the-box" ready to be encapsulates into a multi-thread scheme. Therefore we have systematically modified all global variables storage making them "thread local", and we have substituted all the non-reentrant *C* primitives with their corresponding reentrant versions. After these changes, we were able to synchronize the code by means of a step-by-step execution implemented upon *pthread* barrier.
– An improved synchronization has been implemented to equalize the duration of parallel "runs".
– The Jonker and Volgenant's mTSP-to-TSP transformation has been implemented to adapt solutions generated by the RSP optimisation and make them compatible with the current LKH instance.
– A basic control interface has been added to control the execution of the LKH algorithm and to let successive LKH calls execute one after the other with a reduced overhead.
– A route extraction function has been implemented to obtain a suitable amount of diversified routes to fill the route pool.
– The caching system already adopted within the algorithm has been extended and slightly improved.

– The CVRP penalty function has been redesigned, improving its speed while maintaining the exact same behaviour as the original one.
– A Simulated Annealing (SA) scheme has been added on top of the original solution acceptance test, to improve the performance of the original algorithm and to perturb the initial solution in the attempt of escaping from local optima.

For the sake of clarity, in what follows we will call "newLKH" our modified version of the LKH. To give a clearer idea of the structure of the newLKH algorithm and of the introduced changes, a sketch of this variant is given in Algorithm 2. The overall scheme resembles the original LKH, since most of its logic is not affected by our changes. The two main additions are the route-extraction step (ExtractRoutes), and the Simulated Annealing acceptance test (SATest) called on every solution returned by the LinKernighan function. To be more specific, the following functions appear in the pseudocode:

– Cost: Returns the cost of the input solution.
– Kick: Perturbs the input solution; see Section 3.1.3.
– LinKernighan: Calls Helsgaun's implementation of the Lin-Kernighan heuristic on the input solution, possibly refining it; see Algorithm 3 for a simplified overview of the main steps of this phase.
– ExtractRoutes: Given a (possibly infeasible) tour, returns all its feasible routes.
– SATest: Manages the current solution update according to the Simulated Annealing metaheuristic approach described in Section 3.1.3.
– TimeLimitReached: Simple test that returns true if the given time limit for the phase 1 of the LS-CGH has been reached, false otherwise.

Algorithm 2: High-level pseudocode for the LKH algorithm.

```
   Input  : Initial solution S_init.
   Output: The populated route pool P and the best solution found S*.
 1 FUNCTION LKH(S_init)
 2 begin
 3     for Run ← 1 to n_Runs do
 4         S* ← S ← S_init;
 5         for Trial ← 1 to n_Trials do
 6             S ← Kick(S);
 7             S ← LinKernighan(S);
 8             P ← ExtractRoutes(S);
 9             if Cost(S) < Cost(S*) then
10                 S* ← S
11             end
12             S ← SATest(S*, S);
13             if TimeLimitReached() then
14                 return S*, P
15             end
16         end
17     end
18     return S*, P
19 end
```

An overview of the LinKernighan function is provided in Algorithm 3, highlighting the positions of the "Penalty" and "Flip" functions (to be described in

Section 3.1.2 and Section 3.4.1, respectively). The functions that appear in the pseudocode are as follows.

- BestSpecialOptMove: Original LKH function which, given a solution, searches for a r-Opt move that improves it, considering a restrict set of moves specialized for routing problems. An array $M_{rOpt}[1..r]$ of 2-Opt moves and its size r are returned. The proposed move is thus represented as a sequence of r 2-Opt moves to be applied, in sequence, to produce the final r-Opt move; see Sections 3.1.2 and 3.4.1 for further details.
- Flip: Original (for CVRP) or modified (for asymmetric problems) function that applies a single 2-Opt move to a solution; see Section 3.4.1 for details.
- Penalty: Modified version of the original "Penalty" function that, given a solution, returns its infeasibility level; see Section 3.1.2 for details.

Algorithm 3: Simplified representation of the LinKernighan function inside the LKH algorithm

```
   Input  : Initial solution S.
   Output: The refined solution S.
1  function LinKernighan(S)
2  begin
3      P ← Penalty(S);
4      C ← Cost(S);
5      M_rOpt, r ←BestSpecialOptMove(S);
6      do
7          Improved ← false;
8          for t ← 1 to r do
9              S ← Flip (S, M_rOpt[t])
10         end
11         P′ ←Penalty(S);
12         C′ ← Cost(S);
13         if (P′ < P) OR (P′ = P AND C′ < C) then
14             P ← P′;
15             C ← C′;
16             Improved ← true
17         else
18             for t ← r downto 1 do
19                 S ←Flip(S, M_rOpt[t])
20             end
21         end
22     while Improved;
23     return S
24 end
```

Our newLKH version containing all the speed-related optimizations (namely: the new *Penalty* function, the caching system and the new *Flip* function) is freely available, for research purposes, at https://github.com/c4v4/LKH3.

3.1.1 Speed improvements

Some of the most relevant changes aimed at speeding up the execution of the original LKH code are outlined next.

Cost function: To reduce the overhead related to the computation of distances between vertices, the LKH algorithm uses, since its first version, a clever caching system proposed by Bentley [6]. This caching system works with two arrays of the same size: one array is used to save the used distances, while in the other one the smaller of the two node indices is saved as a signature. The position of each distance-signature pair in their respective arrays is chosen with a fast hash function. Thanks to this simple mechanism, both Helsgaun and Bentley report that the time with TSP problems can be halved or more [6, 20].

In the LKH original cost function, several checks are performed before calling the computationally expensive distance function. Indeed, depending on the VRP version and other internal parameters, the required distance might have already been stored by previous operations. Thus, before calling the distance function, all these fields are checked. The cache is checked as a last step, only if none of the fields contains the required value. Even though the performed checks are usually less expensive than a call to the distance function, searching all the places where the distance could have been stored (which are not located adjacently in memory) can be slower than a direct check of the cache which, very often, already contains the actual value required. For this reason, we have modified the original cost function moving the cache check ahead, in a small prologue (often inlined by the compiler even without *linking time optimization*, since it is defined in a shared header file) that first checks if the requested cost is already stored inside the cache. Only when this step fails, it proceeds by calling the remaining part of the cost function, performing all the field checks and, eventually, the final call to the distance function. Furthermore, since distance and signature are always accessed together, the subdivision into two distinct array have been modified into a single array containing the signature and its distance adjacent in memory, to improve the cache-locality of this system.

Forbidden function: The "Forbidden" function tells if a given edge is part or not of the given instance. A simple example of forbidden edges is the set of edges between depot copies—note that, in the Jonker and Volgenan's mTSP-to-TSP transformation [23], multiple copies of the depot are introduced. This function is heavily used by the algorithm, as shown by our profiling. Since the caching mechanism proved to be a really effective improvement for the cost function, we have implemented an analogous mechanism for the Forbidden function, using again a small prologue to possibly skip not only all the checks made by the original one, but also the function-call overhead.

Balanced workload: As previously described, we have modified the original LKH source code to make it reentrant. The reason for this extensive modification has been the need of enabling a parallel execution of multiple instances of the LKH algorithm. However, running different threads in parallel, synchronized only at the beginning and at the end of each LKH call, often leads to an unbalanced situation where some threads take less time than others. This difference varies randomly with the status of the algorithm. To avoid the waste of potential computational resources, all the threads are synchronized such that each parallel run ends only when the slowest one has ended. In this way, fast runs (which sometimes are even twice as faster as the slowest one), can carry on with their "trials", avoiding to reach the phtread barrier early and then wait for the others to finish.

Some utility procedures have also been implemented to connect LKH with the remaining part of our LS-CGH scheme. We next describe two main components of such an interface: the route pool and the Jonker and Volgenant's solution transformation.

Route Pool: To store the routes extracted by the solutions generated by the LKH we have implemented a simple route pool. We have decided to use a data structure inheriting from C++ STL *std::unordered_set* to avoid duplicates while keeping the best version of each route within the same group of nodes. Every route is distinguished from the others by the set of visited customers (which are saved as a sorted list), while the actual customer sequence and the length of the routes are updated every time a better "duplicate" is found.

Jonker and Volgenant's solution transformation: An important transformation, proposed by Jonker and Volgenant [23] and applied in LKH, is the mTSP-to-TSP conversion which transforms an instance with m salespersons into a TSP instance with $m-1$ copies of the depot. This transformation is used to reduce the search space, decreasing the symmetry of mTSP and other problems with multiple routes (e.g., CVRP). It is easy to see that when $m-1$ identical copies of the depot are introduced into the graph, for each tour there exists $m!$ equivalent tours which only differ by the order of the depot copies. This transformation deletes part of the edge of the graph, by assigning to some selected nodes two depot copies to which they are allowed to be connected with, and by forbidding the edges to the other depot copies—thus reducing the number of possible route permutations.

A problem we encountered interfacing the RSP phase with the LKH one, concerns the compatibility of the CVRP solutions produced. Indeed, the combination of routes with the Set-Partitioning ILP optimization does not consider the Jonker and Volgenant's mTSP-to-TSP transformation [23] applied within the LKH algorithm. When the ILP optimization generates CVRP solutions, the transformation is applied to avoid the use of the forbidden edges. Our algorithm follows the general directives advised in the original Jonker and Volgenant's paper [23], namely:

1. Starting from a general CVRP solution, the routes are extracted and the depot is removed, obtaining a list of chains of customers.
2. The depot is copied, obtaining a number of depots equal to the number of vehicles.
3. All the chain endpoints (two for each chain) are considered. Accordingly to the transformation already in place within the current LKH instance, for each end point that results to be a *special* customer (in the sense of the Jonker and Volgenant's paper: a customer for which the transformation has assigned only two depot copies) the required depots are assigned.
4. Then the main cycle of the transformation begins. Starting from one, all the chains are concatenated one after the other, ensuring that all the *special* customers are not linked with forbidden depot copies.

3.1.2 New Penalty Function

Although quite effective in practice, the above improvements are of a minor theoretical relevance since they simply accelerate the algorithm without modifying its

original scheme—or provide an interface for other modules to interact with it more freely. On the other hand, the *Penalty* function modification has been characterized by a more prominent re-design of one of the main bottleneck functions. LKH is characterized by a hard division between the penalty value of a solution, which correlates to a measure of the "amount of constraint violation", and the actual cost of the objective function. At run-time, LKH gives higher priority to the improvement (i.e., decrease) of the penalty, considering the edge-cost gain achieved by the proposed r-Opt move only when the penalty variation is zero.

For any given solution, the *Penalty* function computes the penalty value with a computational complexity linear in the size of the CVRP solution. Inside LKH, such a solution is represented by a TSP tour containing a number of depot copies equal to the number of vehicles (following the Jonker and Volgenant [23] symmetry-breaking transformation). In what follows, the term "tour" will refer to this internal representation and it will not be a synonym for "route", which instead refers to the cycle covered by a singe vehicle.

The *Penalty* function is called inside the LKH to check a new proposed solution in the following way:

1. A new r-Opt move is found and stored (decomposed as a series of 2-Opt moves) within the LK function.
2. The move is applied to the best tour found in the current "trial", named *current* tour, obtaining a new *proposed* tour.
3. The penalty function is called to check the *proposed* tour.
4. If the *proposed* tour improves the penalty of the *current* tour, or keeps the penalty unchanged while improving its cost, it becomes the new *current* tour, otherwise the saved r-Opt move is reversed to obtain the original *current* tour.

Notice that, at any given time, the *proposed* and *current* tours are abstract concepts used to explain their role, while the tour stored in memory is actually one which is first modified and then eventually restored if it does not improve the previous one.

However, due to its strict policy requiring that the infeasibility level can never increase, the *Penalty* function frequently rejects new candidates solutions. As a matter of fact, in almost all our tests the function rejects the proposed tour more than 95% of the times, thus representing one of the main bottlenecks for the entire algorithm. This observation enabled us to optimize the original LKH scheme by speeding-up the frequent "rejecting" case, introducing a rarely executed "update" step, thus resulting in a significant performance improvement. Indeed, the main change to the original penalty function has been the restriction of the penalty checks to only the routes "touched" by the *proposed* r-Opt move. Since the penalty function is called at every new potential change of the tour, these are the only routes modified between successive calls of the penalty function.

As in the original code there is no route-related data structure, a basic one has been implemented to store the route penalty for the current tour. Then, for each node, a reference to its route-data is stored, in accordance to the current CVRP solution. Thanks to this additional information, one can efficiently retrieve the *current* penalties of the routes touched by the *proposed* r-Opt move, as they appear in the *current* tour.

As a further optimization, we observe that route penalties need to be stored only if the current tour penalty is not yet zero. Indeed, when a feasible CVRP solu-

tion has been found (and the current penalty is, therefore, zero), then the previous cumulative penalty of any subset of routes is also zero. Therefore the previously described step can be completely avoided to further speed up the function.

Finally, when a *proposed* tour is accepted, an update procedure needs to be executed to restore route-data consistency.

3.1.3 Simulated Annealing

To avoid to get stuck in local optima, the original LKH algorithm uses a so-called "kick" strategy, i.e., every time a "trial" of the core LK procedure cannot find any other move that improves the *current* solution, a random r-Opt move (usually a double bridge 4-Opt move [3, 4, 19, 28]) is applied to the current solution and the LK procedure is called again. As previously explained, a single iteration of such scheme is named "trial" in the LKH context. This technique has however two shortcomings:

- When LK is applied over a TSP instance that maps the VRP one, the additional constraints applied through the penalties make the search space very sparse. Therefore, although effective with true TSP instances, it can result to be not powerful enough to perturb the solution and move from the current VRP local-optima.
- When a warmstart is provided to the algorithm, LKH starts from a potentially very good local optimum from which it is not able to move (especially if such a warmstart has been produced by previous iterations of the LKH algorithm itself). Therefore, a perturbing strategy able to lead the search trajectory away from this starting point and to explore new solution neighborhoods is needed.

As in the recent FILO heuristic [2], we decided to integrate a Simulated Annealing (SA) [25] scheme into LKH, motivated also by the compatibility of the original penalty-based scheme with such a technique.

Two overlapping SA schemes have been implemented, one based on the number of "trials", and one based on the LKH time limit. During the execution, the temperature is decreased for both the SAs and the smaller one is considered for the actual SA acceptance test. In this way, when both the trial and the time limits are given, the algorithm can automatically adapt to fit the tighter of the two.

Inspired again by the SA implementation in FILO [2], we have set up our SA scheme as follows:

- The ratio between the initial temperature and the final one has been fixed to 100.
- Adopting the terminology introduced in Section 3.1, let z be the cost of the proposed solution, z' be the cost of the current solution used as a starting point, and T^t be the temperature at the "trial" t of the algorithm. The solution z is accepted as new current solution if

$$z - z' < T^t \cdot ln(U[0,1])$$

where $U[0,1]$ is a uniform random variable in the $[0,1]$ range.

– Two distinct temperatures are maintained during the execution, namely: T^t_{trial} which represent the trial-based SA temperature, and T^t_{time} which is the temperature of the time-based one. The actual temperature T^t is computed as the minimum of the two. Therefore, the update formulas are:

$$T^{t+1}_{trial} = 0.01^{1/MTRIAL} \cdot T^t_{trial}$$

$$T^{t+1}_{time} = 0.01^{\Delta t/TMAX} \cdot T^t_{time}$$

$$T^{t+1} = min\{\, T^{t+1}_{trial}, \, T^{t+1}_{time}\,\}$$

where $MTRIAL$ is the maximum number of "trials", Δt is the time lasted from "trial" t and "trial" $t+1$, and $TMAX$ is the time limit for the "run".
– Finally, the initial temperature is computed as the value of the best solution obtained after 50 "trials", multiplied by a factor c (say) defined as follows. As we aim for long runs, we have distinguished the initial part of the algorithm (where the objective is to find a good solution without getting stuck into local optima) from the second one (which tries to find improvements to the given initial solution). For the first part a factor c_z (say) has been used to scale the initial temperature when no initial solution is provided to the algorithm, while c_w (say) is the same factor when an initial solution is present—because provided externally or from previous rounds of the algorithm. After some preliminary computational tests, we have fixed $c_z = 2.5 \cdot 10^{-3}$ and $c_w = 5 \cdot 10^{-4}$.

3.2 Phase 2: Column Generation Filtering

The number of routes generated during the LKH execution is typically exceedingly large, hence a technique to select the best routes is essential for the efficiency of the whole algorithm.

Considering our heuristic context, we need to balance two aspects: efficiency of the column generation phase, and RSP optimization speed. To achieve the former, a set of policies built around the common objective of finding a good and relatively small subset of routes has been defined, from which the RSP optimization could start. The initial core set of candidate routes consists in the selection of the "best" $8,000$ routes from the ordered list of all routes, sorted by non-decreasing solution costs.

(Indeed, in our computational tests we have seen that values between $5,000$ and $10,000$ are adequate for fast runs where the Set-Partitioning phase needs to be fast to avoid introducing large slow-down for the whole LS-CGH algorithm.)

Starting from this core set, the following filtering techniques are applied:

1. The LP relaxation of the RSP containing only the initial set of route is iteratively solved using the dual simplex algorithm. At each iteration, the reduced costs of the routes still in the route pool are computed, saving the value of the most negative one, say $\bar{c}_{min} < 0$. At this point, the routes with a reduced cost less than $0.8 \cdot \bar{c}_{min}$ are added to the RSP, therefore inserting a number of potentially useful columns at each iteration. This pricing procedure stops when all reduced costs are nonnegative, or when a time limit is reached.

2. Since the previous policy often does not select enough routes, we also use a filtering criterion akin to the one proposed by Caprara et al. [7] for the solution of large-scale set covering problems. At every pricing iteration we also select, for each customer, the ten routes with smallest (possibly positive) reduced costs. The pricing procedure stops when the time limit is reached or when the cumulative sum of the reduced costs added during the previous iteration, becomes nonnegative.

To handle the case in which the pricing procedure selects too many routes, we have set as a hard bound value equal to $16,000$, i.e., twice the initial set size.

3.3 Phase 3: Restricted Set Partitioning Problem Optimization

The final step of our scheme consists in the solution of the RSP formulation. For this task we used a state-of-the-art commercial MIP solver (IBM ILOG CPLEX 12.10). Although this is an exact algorithm, it has been successfully integrated in our heuristic scheme by setting an aggressive time limit and by an early activation of its "polishing procedure" [37].

It is worth observing that, as an alternative to the SP formulation, a Set Covering formulation might be used, that would allow for route overlaps. (Note that multiple customer visits can be removed by a short-cut post-processing procedure, that for instances with costs satisfying the triangle property would even reduce the final solution cost.) However, as reported by Rochat and Taillard [36], and confirmed by our own computational tests, the Set Covering formulation is significantly slower to solve by our MIP solver, so we preferred to stay with the SP formulation.

3.4 VRP Taxonomy

To position our technique within the VRP scientific literature and to give a clearer idea of its applicability to other VRP variants, we make use of the Pillac et al. [34] VRP taxonomy. Broadly speaking, VRPs can be classified by the point of view of the instance data evolution, in this sense that we have *static* problems where all the information is known beforehand, vs. *dynamic* problems where the information regarding the instance is known only during the optimization. Then, we have *deterministic* vs. *stochastic* problems: in the former, all information is known exactly, while in the latter the input data is modelled in the form of random variables. From the product of this two classifications, one obtains four different classes:

– static and deterministic;
– dynamic and deterministic;
– static and stochastic;
– dynamic and stochastic.

The technique proposed in the present work specifically aims at problems of the first category: static and deterministic, as this is the nature of our local search and set partitioning phases.

More precisely, our scheme can readily be extended to all the VRP variants characterized by solutions with independent routes (i.e., variants that can be represented through the SP formulation, needed for the SP-phase of our algorithm) and supported by LKH. Here is a brief list of possible candidates:

- Multiple Travelling Salesman Problem (m-TSP)
- Capacitated Vehicle Routing Problem (CVRP)
- Capacitated Vehicle Routing Problem with Time Windows (CVRPTW)
- Vehicle routing problem with backhauls (VRPB)
- Vehicle routing problem with backhauls and Time Windows (VRPBTW)
- Vehicle routing problem with mixed pickup and delivery (VRPMPD)
- Vehicle routing problem with simultaneous pickup and delivery (VRPSPD)
- Vehicle routing problem with mixed pickup and delivery and time windowsb (VRPMPDTW)
- Vehicle routing problem with simultaneous pickup and delivery and time windows (VRPSPDTW)

Of course, for any such VRP variant one needs to implement a specialized feasibility check for the routes found in the LKH solutions, to ensure that only feasible routes are inserted into the route pool.

One could also extend our technique to other variants which are compatible with the TSP–tour representation and the LKH penalty system. In this case, the implementation would be more involved than in the previously cited variants (which are already supported by Helsgaun's algorithm) since, along with the definition of the Penalty function, also the the internal data structure should be modified and extended. Similarly, all prepossessing steps (including the instance file parsing, the application of potential reductions or other preprocessing operations that can simplify the search) should be revised to account for the new variant.

3.4.1 New Flip Function

Within LKH, most VRP variants undergo an ATSP-to-TSP transformation [22], hence in what follows we will use the *symmetric* and *asymmetric* terms not to refer to the cost of the arcs in the original problem formulation, but to the cost of the arcs of the LKH internal representation of the problem. For instance, a symmetric CVRPTW instance is converted to an asymmetric one so as to remove all the finite-cost arcs which are not feasible due to the time-window constraints. In this sense, among the above-mentioned variants, only the "m-TSP" and the "CVRP" variants are viewed as symmetric problems, while all the others are asymmetric.

Within the LKH algorithm, whenever a 2-Opt move is applied, a function named *Flip* is called to copy a portion (segment) of the TSP tour representation in its reversed order. The operation is part of every 2-Opt move, although sometimes it could be avoided by applying more complex r-Opt moves that maintain the orientation of every part of the solution. Within LKH, every r-Opt move is decomposed into a sequence of 2-Opt moves, hence every r-Opt move must go through different "flips". If naively implemented, each flip operation has a $O(n)$ complexity, and is often the main bottleneck of any r-Opt move-based algorithm.

To improve its overall performance, LKH exploits a clever data structure due to Fredman et al. [16]. Three versions of the *Flip* function are implemented, with

complexity $O(n)$ (naive doubled-linked list version), $O(\sqrt{n})$ (two-level tree), and $O(\sqrt[3]{n})$ (three-levels tree), respectively. In particular, the second one is usually adopted since it is able to maintain a good trade-off with the size of common instances.

We observed that most of the proposed r-Opt are rejected by the *Penalty* function. As the *Flip* function is called every time an r-Opt move is applied, in the very likely "rejection" case the solution undergoes two "flip" operations: one to produce a *proposed* tour, and another to restore the *current* tour. As a result, this function can be optimized by introducing an "update" step when a better solution is found, with a significant speedup for the most-common "rejection" case.

4 Computational Results

In the present section, we address the following questions:

- How effective are our improvements to the original LKH implementation, in particular in terms of speed?
- Is our overall refinement heuristic able to improve the solutions found, in long computing times, by a state-of-the-art CVRP heuristic such as FILO?
- Are we able to improve some best-known solutions from CVRPLIB library, thus providing an implicit comparison will the best methods from the literature—that arguably have been applied to the instances of this well-known library?

In the computational tests that follow, the Uchoa et al. [42] X dataset has been used. Following Queiroga et. al [35], this dataset was restricted to its largest 57 instances (called 57-X in what follows).

For speedup evaluation and for the final tests with the FILO heuristic, we also considered the XXL set [5] which contains 10 instances of size up to 30,000 customers.

All the tests have been performed on Intel Xeon E3-1220 V2 CPUs, using up to 4 threads. We will refer to the *Gap* of a solution with respect to the currently Best-Known Solution (BKS), defined as:

$$Gap := \frac{Solution_value - BKS_value}{BKS_value}.$$

When not available, an initial solution can be obtained by using one of several constructive methods that LKH provides. In its default setting, a pseudo-random procedure is selected that takes into account the possible presence of some restrictions on the edges of the graph, like the presence of "fixed" edges. Another useful constructive CVRP algorithm implemented within LKH is the Clarke and Wright (CW) saving algorithm [9]. Our computational experience shows that, for the X dataset, the final solution quality does not depend too much on the selected constructive heuristic. For the bigger XXL instances,instead, CW is often superior to the pseudo-random one, as it starts from a solution that, even when infeasible, is of better quality. Thus, for the single-thread speedup tests described in Section 4.1 we use CW for the initialization. For the comparison with the original LKH in Section 4.2, instead, we use CW for the first thread, while for the remaining threads we use the pseudo-random one to help increasing route pool variability. Notice that, for both newLKH and LS-CGH, only the very first round makes use

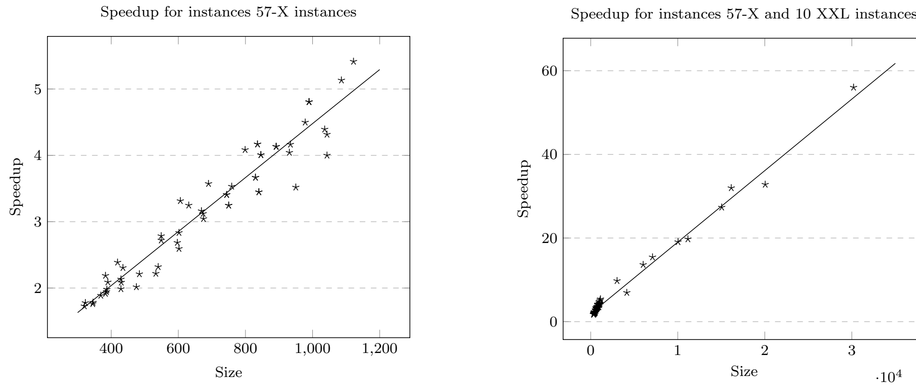

Fig. 1 On the left, the average speedup of the "New" LKH with respect to the original version for 5 random seeds on the 57-X set. On the right the same chart including the 10 XXL instances.

of such an initialization, while the best solution found is used in the other rounds.

4.1 Original LKH vs New LKH

In this section, the original LKH is compared with our modified version. The comparison only addresses the LKH phase of LS-CGH (i.e., without RSP and route extraction), both run in single-thread for the same number of "trials". As the implemented LKH changes do not alter the search trajectory between the original version and the new one (when run in single-thread mode and when the same random seed is used), the two versions visit the same solutions sequence and perform the same algorithmic steps, hence producing the same final solution.

In Table 1 the speedup achieved by the new version is reported along with the size of the instance. (Since inside the LKH each solution in represented by a TSP tour of length equal to the number of customers plus the number of vehicles, we report this figure as the size of the instance.) Along with the 57-X test-bed, the 10 XXL instances of the Belgium data-set has been considered in order to evaluate the behaviour of the algorithm for a broader range of sizes.

For each test a single "run" of the LKH was executed, starting from a near-optimal warmstart. The number of "trials" has been set to $10,000$ and 5 random seeds where tried for each instance. The reported speedup is the average of the 5 speedups obtained by each seed.

Figure 1 (left) shows how the speedup scales with the size of the instance of the 57-X set. The linear increase of the speedup with the size of the instances is further confirmed in Figure 1 (right) where also the very large XXL instances are considered.

Table 1 Speedups of the modified LKH (newLKH) with respect to the original one. The size of the instances is computed as the number of customers plus the number of vehicles. Results are for the 57-X and XXL sets. (*) For the Flanders2 instance, the number of "trials" has been halved, since in the original algorithm 10,000 "trials" would have been computationally too expensive.

Instance	Size	LKH Time	newLKH Time	SpeedUp
X-n303-k21	323	86	49	1.78
X-n308-k13	320	69	40	1.73
X-n313-k71	384	732	335	2.19
X-n317-k53	369	165	87	1.89
X-n322-k28	349	166	89	1.87
X-n327-k20	346	73	41	1.78
X-n331-k15	345	51	29	1.76
X-n336-k84	421	788	330	2.39
X-n344-k43	386	163	84	1.94
X-n351-k40	390	377	180	2.09
X-n359-k29	387	153	77	1.98
X-n367-k17	383	91	48	1.92
X-n376-k94	469	210	96	2.19
X-n384-k52	436	606	263	2.31
X-n393-k38	430	146	70	2.08
X-n401-k29	429	231	108	2.13
X-n411-k19	429	109	55	1.99
X-n420-k130	549	262	94	2.79
X-n429-k61	490	398	166	2.39
X-n439-k37	475	64	32	2.02
X-n449-k29	477	400	170	2.36
X-n459-k26	484	182	82	2.21
X-n469-k138	607	894	270	3.31
X-n480-k70	549	275	103	2.66
X-n491-k59	549	518	190	2.72
X-n502-k39	540	108	47	2.32
X-n513-k21	533	60	27	2.22
X-n524-k153	678	812	206	3.94
X-n536-k96	631	1145	353	3.25
X-n548-k50	597	185	69	2.68
X-n561-k42	602	130	50	2.59
X-n573-k30	602	248	87	2.83
X-n586-k159	744	571	149	3.82
X-n599-k92	691	2184	611	3.57
X-n613-k62	674	397	127	3.12
X-n627-k43	669	322	102	3.16
X-n641-k35	675	398	131	3.04
X-n655-k131	785	211	66	3.20
X-n670-k130	802	944	231	4.08
X-n685-k75	759	499	141	3.53
X-n701-k44	744	328	96	3.41
X-n716-k35	750	417	128	3.25
X-n733-k159	892	343	83	4.13
X-n749-k98	846	845	211	4.01
X-n766-k71	836	863	207	4.17
X-n783-k48	830	528	144	3.67
X-n801-k40	840	234	68	3.45
X-n819-k171	990	1791	373	4.81
X-n837-k142	978	582	129	4.50
X-n856-k95	950	186	53	3.52
X-n876-k59	934	760	182	4.16
X-n895-k37	932	950	235	4.04
X-n916-k207	1122	811	150	5.41
X-n936-k151	1092	884	172	5.13
X-n957-k87	1043	239	60	4.00
X-n979-k58	1036	919	209	4.39
X-n1001-k43	1043	434	101	4.32
Antwerp1	6342	1052	77	13.62
Antwerp2	7119	1992	129	15.43
Brussels1	15511	4287	157	27.34
Brussels2	16181	7660	240	31.98
Flanders1	20683	7037	215	32.80
Flanders2	30255	17166	306	56.01
Ghent1	10484	2028	106	19.07
Ghent2	11109	4469	226	19.75
Leuven1	3202	670	69	9.77
Leuven2	4045	710	102	6.94

4.2 Original LKH vs new LKH vs LS-CGH

In order to asses the effectiveness of the proposed scheme, three different variants have been compared. All the tests have been executed with the same time limit of 200 minutes, using 4 threads for both the LKH "runs" and the CPLEX solver (when used).

In Table 2 we compare the original LKH, the new LKH and our LS-CGH methods, and report the best gap reached (w.r.t the BKS) after 200 minutes. The "LKH" columns give the performance of the original LKH algorithm, without the proposed improvements and executed without the "round" subdivision adopted in our scheme. Four parallel threads with "runs" of $10,000$ "trials" have been executed, until the time limit was reached. The "newLKH" columns give instead the solutions obtained by our new LKH scheme, without the SP phase. All the improvements applied to the original algorithm have been activated and the LKH "runs" (with $50,000$ "trials" each and a time limit of 2000 seconds for each "run") have been subdivided into "rounds" of 4 parallel "runs", providing each round with the best solution found by the previous one. Finally, in the "LS-CGH" columns the results for our complete LS-CGH algorithm are reported, thus including the same setup as in the newLKH columns with the addition of the SP phase.

Both newLKH and LS-CGH show a significant decrease in the average gap, as well as a consistently lower gap for each instance in the 57-X set.

It is worth noting that the LKH algorithm involves a large number of parameters to tune: in our tests, we used the default values provided in the scripts available in Helsgaun's website. In Table 2, a significant improvement is shown already by our own version of LKH (namely, newLKH). This is due to three main factors.

- The improved time performance of the algorithm allowed for the exploration of a larger number of r-Opt moves with respect to the original LKH.
- The SA in the first round, applied with a high initial temperature, takes better advantage of a large number of "trials". The search descent is therefore less steep (w.r.t. the number of "trials"), and also less prone to get stuck into local optima.
- The adopted "round" subdivision, in which the best solution obtained is used as warmstart for the next "round", greatly improves the efficacy of the algorithm to refine the solutions in long runs.

Finally, with the addition of CG filtering and RSP optimization, further improvements have been obtained.

4.3 Statistical analysis of LS-CGH

A statistical analysis of percentage gaps obtained for multiple runs on a representative subset of the studied instances has been carried out. From the 57-X dataset, we have chosen seven representative instances selected as suggested by Queiroga et al. [35] so as to cover all the different characteristics considered during the generation of the whole X dataset. As to the Belgium data set, two (Antwerp1 and Flanders1) out of the ten instances have been randomly chosen. For these two instances, simulated annealing has been disabled because, for these sizes, the

Table 2 Comparison between the solution obtained in 200 minutes runs by: the original LKH algorithm, Helsgaun's LKH with our changes and inserted in our scheme (newLKH), and our final LS-CGH algorithm (i.e., newLKH followed by RSP optimization). The best result for each instance is highlighted in boldface.

Instance	LKH		newLKH		LS-CGH	
	Sol	Gap	Sol	Gap	Sol	Gap
X-n303-k21	21877	0.65%	**21803**	**0.31%**	21805	0.32%
X-n308-k13	25995	0.53%	**25900**	**0.16%**	25919	0.23%
X-n313-k71	96097	2.18%	95330	1.37%	**94604**	**0.60%**
X-n317-k53	78409	0.07%	78361	0.01%	**78355**	**0.00%**
X-n322-k28	30061	0.76%	29968	0.45%	**29850**	**0.05%**
X-n327-k20	27800	0.97%	27640	0.39%	**27619**	**0.32%**
X-n331-k15	31289	0.60%	**31103**	**0.00%**	**31103**	**0.00%**
X-n336-k84	143175	2.92%	142122	2.16%	**141194**	**1.50%**
X-n344-k43	42417	0.87%	42201	0.36%	**42156**	**0.25%**
X-n351-k40	26343	1.73%	26133	0.92%	**26016**	**0.46%**
X-n359-k29	51807	0.59%	51652	0.29%	**51579**	**0.14%**
X-n367-k17	22955	0.62%	22824	0.04%	**22814**	**0.00%**
X-n376-k94	147807	0.06%	147720	0.00%	**147713**	**0.00%**
X-n384-k52	67082	1.73%	66403	0.71%	**66389**	**0.68%**
X-n393-k38	38519	0.68%	38335	0.20%	**38260**	**0.00%**
X-n401-k29	66485	0.50%	66481	0.49%	**66373**	**0.33%**
X-n411-k19	19890	0.90%	19780	0.34%	**19756**	**0.22%**
X-n420-k130	108247	0.42%	107946	0.14%	**107798**	**0.00%**
X-n429-k61	66135	1.05%	65742	0.45%	**65460**	**0.02%**
X-n439-k37	36559	0.46%	**36402**	**0.03%**	36422	0.09%
X-n449-k29	56118	1.60%	55569	0.61%	**55363**	**0.24%**
X-n459-k26	24508	1.53%	24226	0.36%	**24176**	**0.15%**
X-n469-k138	223542	0.77%	222320	0.22%	**222021**	**0.09%**
X-n480-k70	90031	0.65%	89698	0.28%	**89566**	**0.13%**
X-n491-k59	67355	1.31%	**66739**	**0.39%**	66894	0.62%
X-n502-k39	69317	0.13%	69254	0.04%	**69226**	**0.00%**
X-n513-k21	24428	0.94%	24268	0.28%	**24275**	**0.31%**
X-n524-k153	154662	0.04%	154616	0.01%	**154605**	**0.01%**
X-n536-k96	95924	1.14%	95224	0.40%	**95032**	**0.20%**
X-n548-k50	87031	0.38%	86836	0.16%	**86762**	**0.07%**
X-n561-k42	42998	0.66%	42854	0.32%	**42794**	**0.18%**
X-n573-k30	51053	0.75%	50835	0.32%	**50799**	**0.25%**
X-n586-k159	191487	0.62%	190593	0.15%	**190482**	**0.09%**
X-n599-k92	115113	6.14%	111324	2.65%	**110475**	**1.87%**
X-n613-k62	60467	1.57%	60136	1.01%	**59736**	**0.34%**
X-n627-k43	63000	1.34%	62395	0.37%	**62356**	**0.31%**
X-n641-k35	64551	1.36%	64205	0.82%	**64109**	**0.67%**
X-n655-k131	106943	0.15%	106857	0.07%	**106780**	**0.00%**
X-n670-k130	147052	0.49%	146812	0.33%	**146407**	**0.05%**
X-n685-k75	69310	1.62%	68554	0.51%	**68474**	**0.39%**
X-n701-k44	82933	1.23%	82521	0.73%	**82344**	**0.51%**
X-n716-k35	44186	1.87%	43637	0.61%	**43603**	**0.53%**
X-n733-k159	137622	1.05%	136477	0.21%	**136359**	**0.13%**
X-n749-k98	78682	1.83%	77863	0.77%	**77738**	**0.61%**
X-n766-k71	115728	1.15%	114910	0.43%	**114776**	**0.31%**
X-n783-k48	73497	1.53%	72822	0.60%	**72704**	**0.44%**
X-n801-k40	73976	0.92%	**73469**	**0.22%**	73484	0.24%
X-n819-k171	161871	2.37%	159287	0.74%	**159101**	**0.62%**
X-n837-k142	195666	1.00%	194453	0.37%	**194269**	**0.27%**
X-n856-k95	89473	0.57%	**89036**	**0.08%**	89102	0.15%
X-n876-k59	100297	1.01%	**99930**	**0.64%**	99986	0.69%
X-n895-k37	56497	4.90%	54827	1.80%	**54575**	**1.33%**
X-n916-k207	331620	0.74%	330093	0.28%	**329643**	**0.14%**
X-n936-k151	134163	1.09%	133169	0.34%	**133146**	**0.32%**
X-n957-k87	86197	0.86%	85606	0.16%	**85526**	**0.07%**
X-n979-k58	120354	1.16%	119977	0.84%	**119685**	**0.60%**
X-n1001-k43	74142	2.47%	**72820**	**0.64%**	72966	0.84%
Average		**1.16%**		**0.48%**		**0.32%**

time limit is not enough to get stuck into local optima. Thus, the use of simulated annealing would only make local search slower without the benefit of the broader exploration that would happen with a much longer time limit. For each instance, ten runs with different random seeds have been executed, and the corresponding box-plots are reported in Figure 2.

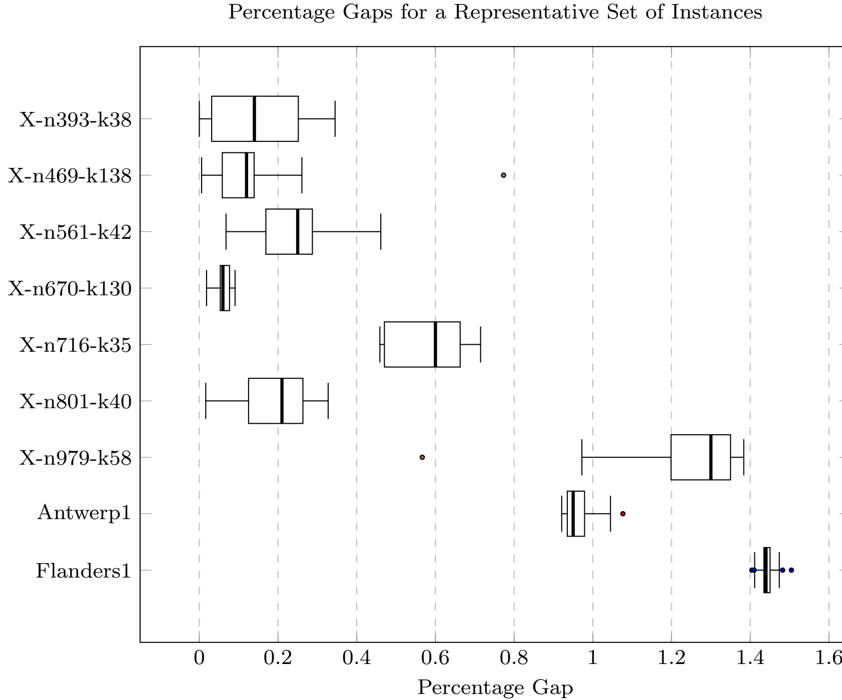

Fig. 2 Statistical analysis of percentage gap w.r.t. the best-known solution, on a representative subset of the 57-X and Belgium datasets.

According to the plot, a low variation is experienced for the Belgium instances. This can be explained by the fact that, for these very large problems, the 200-minute time limit is quite restrictive, hence the algorithm had less time to find local optima in which getting stuck. For the seven instances from the 57-X dataset, instead, the computing time allowed let the algorithm reach several local optima, hence the higher variance due to implemented diversification mechanisms—exceptional cases being the X-n469-k138 and X-n979-k58 instances with their outliers.

Figures 3 and 4 report a similar analysis for the ten instances in Table 2 for which LS-CGH got the best and worst relative gaps, respectively.

4.4 LS-CGH as a refinement tool for FILO

To asses the ability of improving the solution obtained by state-of-the-art heuristic algorithms, our proposed scheme has been tested starting from the best solution

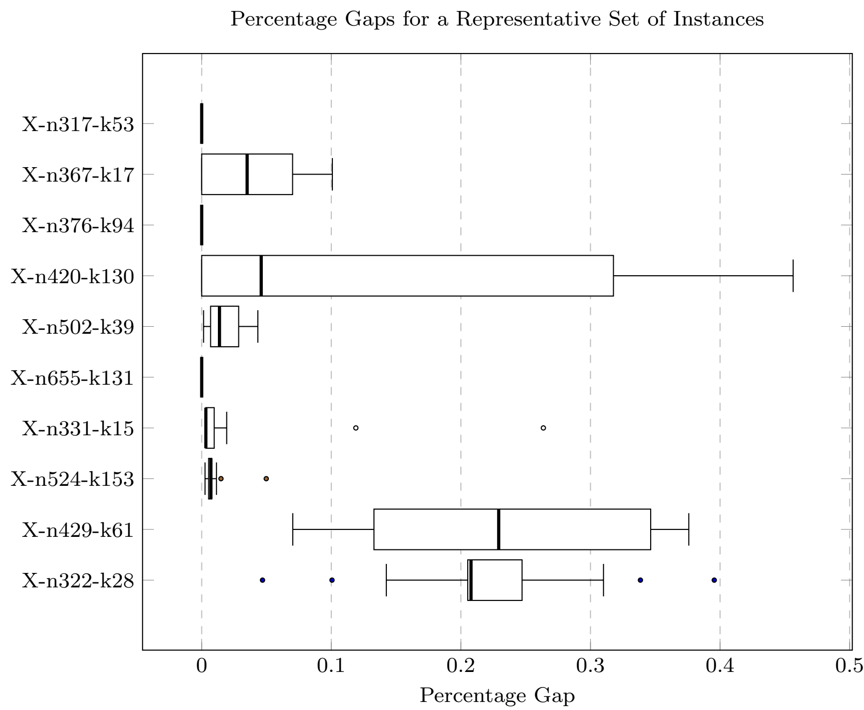

Fig. 3 Statistical analysis of percentage gap w.r.t. the best-known solution, on the ten instances of Table 2 where LS-CGH performed best in terms of relative gap.

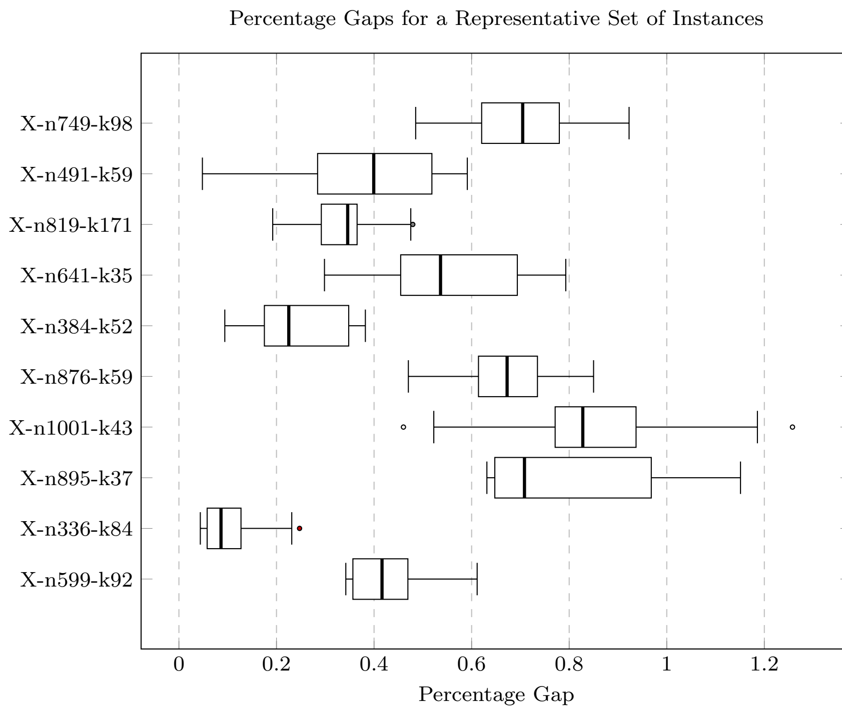

Fig. 4 Statistical analysis of percentage gap w.r.t. the best-known solution, on the ten instances of Table 2 where LS-CGH performed worst in terms of relative gap.

obtained by FILO [2]. As previously described, FILO is a recent fast and effective heuristic, especially designed for instances of very large size as those in the XXL dataset. The solutions obtained by FILO on a very large number of instances from the literature are available online [1].

Our test consisted in a long run (200 minutes) of our algorithm starting from the best solutions obtained by the 10M-iteration runs of FILO. For each instance, we selected the best solution among those produced by FILO in 10 runs with different random seeds.

As shown in Tables 3 and 4, our LS-CGH algorithm is consistently able to improve many of the solution produced by FILO, lowering the average gap to 0.076% for the largest 57 instances of the X data-set, and to 0.079% for the XXL data-set.

4.5 CVRPLIB best-known solution improvements

During the months preceding the writing of the paper, our LS-CGH algorithm was consistently and repeatedly able to improve the best-known solutions (BKSs) for a number of instances from the literature, competing with many other algorithms developed by different groups around the world. The current BKSs are maintained in the CVRPLIB website [30], where the history of the obtained improvements is also reported. As stated in the website, everyone can submit new BKSs, without a description of the applied techniques. This fact has enabled a number of different "competitors" to submit many improvements, especially for the difficult instances of the X and XXL datasets. Different techniques have been applied to these instances, both refining heuristic starting from the previous BKS, and "standalone" ones starting from scratch.

In our case, for 30 large-scale well-studied instances from the CVRPLIB, we have been able to improve the BKSs from literature several times, providing a total of 105 improved BKSs. At the time of writing (March 2021), 14 BKSs produced by our LS-CGH heuristic are still unbeaten; see Table 5. After an initial testing phase where the ensemble of proposed techniques was still incomplete, all the new BKS have been obtained using the same parameter setting, with the only exception of the overall time limit which was set to infinity. Thus, for each instance we "manually" monitored the time lasted from the last improvement, and aborted the code when no improvement was found in the least 24 hours.

5 Conclusions

In this work a new CVRP refining heuristic, LS-CGH, has been proposed. We use a custom parallel and optimized version of the Lin-Kerninghan-Helsgaun heuristic to generate a large pool of feasible CVRP routes, and exploit an LP-based pricing procedure to "filter" the most meaningful ones to feed a Set Partitioning model producing the final CVRP solution. Our optimized version of the LKH heuristic is available, for research purposes, at https://github.com/c4v4/LKH3.

The LS-CGH algorithm succeeded in improving several of the best solutions obtained by a recent state-of-the-art heuristic (FILO) in 10M iterations. In addition, a log of the best-known solutions obtained in the past months by our method

Table 3 Best result for 10 runs of FILO with 10 million iterations for the largest 57 instances of the X data-set along with the improvement obtained after 200 minuted by our LS-CGH algorithm. For the XXL instances, SA was disabled due to their extremely large size. Entries in boldface highlight the cases where LS-CGH was able to improve the FILO solution.

Instance	**FILO-10M**		**LS-CGH**	
	Sol	**Gap**	**Sol**	**Gap**
X-n303-k21	21744	0.037%	21744	0.037%
X-n308-k13	25862	0.012%	25862	0.012%
X-n313-k71	94084	0.044%	94084	0.044%
X-n317-k53	78355	0.000%	78355	0.000%
X-n322-k28	29854	0.067%	29854	0.067%
X-n327-k20	27556	0.087%	27556	0.087%
X-n331-k15	31103	0.003%	31103	0.003%
X-n336-k84	139249	0.099%	**139195**	**0.060%**
X-n344-k43	42064	0.033%	42064	0.033%
X-n351-k40	25936	0.154%	**25922**	**0.100%**
X-n359-k29	51507	0.004%	**51505**	**0.000%**
X-n367-k17	22814	0.000%	22814	0.000%
X-n376-k94	147713	0.000%	147713	0.000%
X-n384-k52	66024	0.130%	**65996**	**0.088%**
X-n393-k38	38287	0.071%	**38269**	**0.024%**
X-n401-k29	66187	0.050%	66187	0.050%
X-n411-k19	19756	0.223%	**19755**	**0.218%**
X-n420-k130	107825	0.025%	**107798**	**0.000%**
X-n429-k61	65502	0.081%	**65455**	**0.009%**
X-n439-k37	36395	0.011%	36395	0.011%
X-n449-k29	55312	0.143%	**55280**	**0.085%**
X-n459-k26	24141	0.008%	**24140**	**0.004%**
X-n469-k138	222363	0.243%	**222038**	**0.096%**
X-n480-k70	89471	0.025%	**89457**	**0.009%**
X-n491-k59	66529	0.069%	**66491**	**0.012%**
X-n502-k39	69227	0.001%	**69226**	**0.000%**
X-n513-k21	24201	0.000%	24201	0.000%
X-n524-k153	154607	0.009%	**154605**	**0.008%**
X-n536-k96	95343	0.524%	**95278**	**0.453%**
X-n548-k50	86707	0.008%	**86704**	**0.005%**
X-n561-k42	42751	0.080%	42751	0.080%
X-n573-k30	50736	0.124%	50736	0.124%
X-n586-k159	190694	0.199%	**190686**	**0.194%**
X-n599-k92	108612	0.148%	**108609**	**0.145%**
X-n613-k62	59618	0.139%	**59572**	**0.062%**
X-n627-k43	62189	0.040%	**62184**	**0.032%**
X-n641-k35	63740	0.088%	**63735**	**0.080%**
X-n655-k131	106780	0.000%	106780	0.000%
X-n670-k130	147066	0.502%	**146481**	**0.102%**
X-n685-k75	68339	0.196%	**68318**	**0.165%**
X-n701-k44	81951	0.034%	**81950**	**0.033%**
X-n716-k35	43424	0.118%	**43417**	**0.101%**
X-n733-k159	136274	0.064%	**136265**	**0.057%**
X-n749-k98	77430	0.208%	**77399**	**0.168%**
X-n766-k71	114638	0.193%	114638	0.193%
X-n783-k48	72464	0.108%	**72457**	**0.098%**
X-n801-k40	73311	0.008%	**73307**	**0.003%**
X-n819-k171	158734	0.388%	**158703**	**0.367%**
X-n837-k142	193967	0.119%	**193948**	**0.109%**
X-n856-k95	89001	0.040%	**88966**	**0.001%**
X-n876-k59	99412	0.114%	99412	0.114%
X-n895-k37	53906	0.085%	**53898**	**0.071%**
X-n916-k207	329789	0.185%	**329660**	**0.146%**
X-n936-k151	133019	0.229%	**132999**	**0.214%**
X-n957-k87	85467	0.002%	85467	0.002%
X-n979-k58	119043	0.056%	119043	0.056%
X-n1001-k43	72414	0.082%	**72405**	**0.069%**
Average		**0.101%**		**0.076%**

Table 4 Best result for 10 runs of FILO with 10 million iterations for the XXL dataset along with the improvement obtained after 200 minuted by our LS-CGH algorithm.

Inst	**FILO-10M**		**LS-CGH**	
	Sol	**Gap**	**Sol**	**Gap**
Antwerp1	477619	0.072%	**477598**	**0.067%**
Antwerp2	291528	0.054%	**291493**	**0.042%**
Brussels1	502278	0.102%	**502217**	**0.090%**
Brussels2	345747	0.056%	**345706**	**0.044%**
Flanders1	7248491	0.106%	**7246624**	**0.080%**
Flanders2	4382341	0.163%	**4380571**	**0.122%**
Ghent1	469894	0.077%	**469860**	**0.070%**
Ghent2	258118	0.122%	**258090**	**0.111%**
Leuven1	192915	0.035%	**192909**	**0.032%**
Leuven2	111544	0.130%	**111541**	**0.127%**
Average		**0.092%**		**0.079%**

Table 5 CVRPLIB best-known solution improvements by date. For the current best at the time of writing (March 2021), the following code identifies the authors of the algorithm: (1) Francesco Cavaliere, Emilio Bendotti, and Matteo Fischetti; (2) Eduardo Queiroga, Eduardo Uchoa, and Ruslan Sadykov; (3) Vinícius R. Máximo and Mariá C.V. Nascimento; (4) Thibaut Vidal; (5) Quoc Trung Dinh, Dinh Quy Ta, Duc Dong Do.

Instance	Prev.BKS	3-May-20	18-May-20	17-June-20	30-July-20	8-Aug-20	20-Aug-20	10-Oct-20	15-Dec-20	30-Jan-21	BKS	Authors
X-n351-k40	25928	25919									25896	2
X-n384-k52	65943	65941									65938	3
X-n459-k26	24141						24140				24139	4
X-n536-k96	94950	94921									94846	2
X-n561-k42	42722	42717									42717	1
X-n573-k30	50717	50708	50673								50673	1
X-n641-k35	63737	63723							63684		63684	1
X-n670-k130	146446	146332									146332	1
X-n685-k75	68252	68245									68205	4
X-n716-k35	43414	43412							43373		43373	1
X-n766-k71	114487		114456						114417		114417	1
X-n783-k48	72393								72386		72386	1
X-n801-k40	73311						73305				73305	1
X-n819-k171	158249	158247									158121	2
X-n876-k59	99331	99330	99303								99299	4
X-n895-k37	53946	53935	53928				53870				53860	4
X-n936-k151	132907	132881	132812								132715	2
X-n957-k87	85478	85474									85465	4
X-n979-k58	119008	118996									118976	2
X-n1001-k43	72402	72397	72369								72355	2
Antwerp1	479021	478775	478674	478091	478019	477535					477277	2
Antwerp2	294319	293953	293802	292597	292511	291468	291450	291400	291387	291371	291371	1
Brussels1	504392	504175	504023	503407	503350	502144	501916	501854	501767		501767	1
Brussels2	353285	352658	352012	349602	348740	345627	345616	345565	345553		345551	5
Flanders1	7273695	7272444	7270362	7256529	7256400	7245214	7242182	7241290	7240845		7240845	1
Flanders2	4480972	4469477	4455217	4405678	4402841	4378434	4377986	4377626	4377524	4375193	4375193	1
Ghent1	471084	470902	470818	470329	470306	469838	469602	469586			469532	4
Ghent2	261676	260987	260553	259712	259486	258010	258002	257958	257954	257802	257802	1
Leuven1	193343	193244	193220	193092	193059	192894					192848	4
Leuven2	112751	112378	112280	111860	111794	111499		111489	111447	111399	111399	1

is publicly available on the CVRPLIB website [30], witnessing its ability to improve 105 solutions obtained by the best CVRP heuristics internationally competing on the same testbed.

In future work, our proposed method can be adapted to other routing problems, including the Capacitated Vehicle Routing Problem with Time Windows (CVRPTW), the Capacitated Arc Routing Problem (CARP), the Vehicle Routing Problem with Backhauls (VRPB), and many others. Since LKH itself is able to address some of these VRP variants, it can be used as route generator as suggested in the present work.

Acknowledgements

This work was partially supported by MiUR, Italy. We thank three anonymous referees for their constructive comments.

References

1. Accorsi, L., Vigo, D.: FILO repository. https://github.com/acco93/filo
2. Accorsi, L., Vigo, D.: A fast and scalable heuristic for the solution of large-scale capacitated vehicle routing problems. Transportation Science **55**(4), 832–856 (2021)
3. Applegate, D., Bixby, R., Chvátal, V., Cook, W.: Finding tours in the TSP. Tech. rep., University of Bonn, Germany (1999)
4. Applegate, D., Cook, W., Rohe, A.: Chained Lin-Kernighan for large traveling salesman problems. INFORMS Journal on Computing **15**(1), 82–92 (2003)
5. Arnold, F., Gendreau, M., Sörensen, K.: Efficiently solving very large-scale routing problems. Computers & Operations Research **107**, 32–42 (2019)
6. Bentley, J.L.: K-d trees for semidynamic point sets. In: Proceedings of the sixth annual symposium on Computational geometry, pp. 187–197 (1990)
7. Caprara, A., Fischetti, M., Toth, P.: A heuristic method for the set covering problem. Operations Research **47**(5), 730–743 (1999)
8. Christiaens, J., Vanden Berghe, G.: Slack induction by string removals for vehicle routing problems. Transportation Science **54**(2), 417–433 (2020)
9. Clarke, G., Wright, J.W.: Scheduling of vehicles from a central depot to a number of delivery points. Operations Research **12**(4), 568–581 (1964)
10. Dantzig, G.B., Ramser, J.H.: The truck dispatching problem. Management Science **6**(1), 80–91 (1959)
11. Dantzig, G.B., Wolfe, P.: The decomposition algorithm for linear programs. Econometrica: Journal of the Econometric Society pp. 767–778 (1961)
12. De Franceschi, R., Fischetti, M., Toth, P.: A new ILP-based refinement heuristic for vehicle routing problems. Mathematical Programming **105**(2), 471–499 (2006)
13. Fischetti, M., Fischetti, M.: Matheuristics. In: R. Martí, P.M. Pardalos, M.G.C. Resende (eds.) Handbook of Heuristics, pp. 121–153. Springer International Publishing, Cham (2018)
14. Ford Jr, L.R., Fulkerson, D.R.: A suggested computation for maximal multicommodity network flows. Management Science **5**(1), 97–101 (1958)
15. Foster, B.A., Ryan, D.M.: An integer programming approach to the vehicle scheduling problem. Journal of the Operational Research Society **27**(2), 367–384 (1976)
16. Fredman, M.L., Johnson, D.S., McGeoch, L.A., Ostheimer, G.: Data structures for traveling salesmen. Journal of Algorithms **18**(3), 432–479 (1995)
17. Fukasawa, R., Longo, H., Lysgaard, J., De Aragão, M.P., Reis, M., Uchoa, E., Werneck, R.F.: Robust branch-and-cut-and-price for the capacitated vehicle routing problem. Mathematical Programming **106**(3), 491–511 (2006)
18. Helsgaun, K.: LKH-3. http://akira.ruc.dk/ keld/research/LKH-3

19. Helsgaun, K.: An effective implementation of k-opt moves for the Lin-Kernighan TSP heuristic. Ph.D. thesis, Roskilde University. Department of Computer Science (2006)
20. Helsgaun, K.: General k-opt submoves for the Lin–Kernighan TSP heuristic. Mathematical Programming Computation **1**(2), 119–163 (2009)
21. Helsgaun, K.: An extension of the Lin-Kernighan-Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems (2017). DOI 10.13140/RG.2.2.25569.40807
22. Jonker, R., Volgenant, T.: Transforming asymmetric into symmetric traveling salesman problems: erratum. Operations Research Letters **5**(4), 215–216 (1986)
23. Jonker, R., Volgenant, T.: An improved transformation of the symmetric multiple traveling salesman problem. Operations Research **36**(1), 163–167 (1988)
24. Kelly, J.P., Xu, J.: A set-partitioning-based heuristic for the vehicle routing problem. INFORMS Journal on Computing **11**(2), 161–172 (1999)
25. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. science **220**(4598), 671–680 (1983)
26. Laporte, G., Nobert, Y., Desrochers, M.: Optimal routing under capacity and distance restrictions. Operations Research **33**(5), 1050–1073 (1985)
27. Lin, S., Kernighan, B.W.: An effective heuristic algorithm for the traveling-salesman problem. Operations Research **21**(2), 498–516 (1973)
28. Martin, O., Otto, S.W., Felten, E.W.: Large-step markov chains for the TSP incorporating local search heuristics. Operations Research Letters **11**(4), 219–224 (1992)
29. Monaci, M., Toth, P.: A set-covering-based heuristic approach for bin-packing problems. INFORMS Journal on Computing **18**(1), 71–85 (2006)
30. Pecin, D., Pessoa, A., Poggi, M., Uchoa, E.: CVRPLIB - Updates. http://vrp.atd-lab.inf.puc-rio.br/index.php/en/updates
31. Pecin, D., Pessoa, A., Poggi, M., Uchoa, E.: Improved branch-cut-and-price for capacitated vehicle routing. Mathematical Programming Computation **9**(1), 61–100 (2017)
32. Pessoa, A., Sadykov, R., Uchoa, E., Vanderbeck, F.: VRPSolver. https://vrpsolver.math.u-bordeaux.fr/
33. Pessoa, A., Sadykov, R., Uchoa, E., Vanderbeck, F.: A generic exact solver for vehicle routing and related problems. Mathematical Programming **183**, 483–523 (2020)
34. Pillac, V., Gendreau, M., Guéret, C., Medaglia, A.L.: A review of dynamic vehicle routing problems. European Journal of Operational Research **225**(1), 1–11 (2013)
35. Queiroga, E., Sadykov, R., Uchoa, E.: A POPMUSIC matheuristic for the capacitated vehicle routing problem. Computers & Operations Research **136**, 105475 (2021)
36. Rochat, Y., Taillard, É.D.: Probabilistic diversification and intensification in local search for vehicle routing. Journal of heuristics **1**(1), 147–167 (1995)
37. Rothberg, E.: An evolutionary algorithm for polishing mixed integer programming solutions. INFORMS Journal on Computing **19**(4), 534–541 (2007)
38. Ryan, D.M., Hjorring, C., Glover, F.: Extensions of the petal method for vehicle routeing. Journal of the Operational Research Society **44**(3), 289–296 (1993)

39. Subramanian, A., Uchoa, E., Ochi, L.S.: A hybrid algorithm for a class of vehicle routing problems. Computers & Operations Research **40**(10), 2519–2531 (2013)
40. Taillard, É.D., Helsgaun, K.: POPMUSIC for the travelling salesman problem. European Journal of Operational Research **272**(2), 420–429 (2019)
41. Toth, P., Vigo, D.: Vehicle routing: problems, methods, and applications. SIAM, Philadelphia, PA (2014). URL https://epubs.siam.org/doi/abs/10.1137/1.9781611973594
42. Uchoa, E., Pecin, D., Pessoa, A., Poggi, M., Vidal, T., Subramanian, A.: New benchmark instances for the capacitated vehicle routing problem. European Journal of Operational Research **257**(3), 845–858 (2017)
43. Vidal, T., Crainic, T.G., Gendreau, M., Lahrichi, N., Rei, W.: A hybrid genetic algorithm for multidepot and periodic vehicle routing problems. Operations Research **60**(3), 611–624 (2012)