Computers & Operations Research 78 (2017) 513–525

Contents lists available at ScienceDirect

Computers & Operations Research

journal homepage: www.elsevier.com/locate/caor

An integration of Lagrangian split and VNS: The case of the capacitated vehicle routing problem

Mouaouia Cherif Bouzid [a,b,*], Hacene Aït Haddadene [b], Said Salhi [c]

[a] Department of Logistic and Transportation Engineering, ENST, Dergana, Algiers, Algeria
[b] LaROMaD-Laboratory, Department of Operations Research, Faculty of Mathematics, USTHB-University, BP 32 El-Alia, Bab-Ezzouar 16111, Algiers, Algeria
[c] Centre for Logistics & Heuristic Optimisation, Kent Business School, The University of Kent, Canterbury, Kent CT2 7PE, UK

ARTICLE INFO

Available online 24 February 2016

Keywords:
Routing problems
Route-first cluster-second
Lagrangian relaxation
Subgradient method
Variable neighbourhood search
Hybridisation

ABSTRACT

In this paper, we propose an efficient and novel Lagrangian relaxation method which incorporates a new integer linear programming (ILP) formulation to optimally partition a giant tour in the context of a capacitated vehicle routing problem (CVRP). This approach, which we call *Lagrangian split* (Ls), is more versatile than the ILP which, in most cases, can be intractable using a conventional solver. An effective repair mechanism followed by a local search are also embedded into the process. The mathematical validity of the repair mechanism and its time complexity are also provided. An integration of Ls into a powerful variable neighbourhood search (VNS) is also presented. Computational experiments are conducted to demonstrate that Ls provides encouraging results when applied on benchmark instances and that the integration of Ls into a metaheuristic scheme produces good results when compared to those found by state-of-the-art methods.

1. Introduction

Routing problems consist in optimizing the visit of a set of customers by a fleet of vehicles based at a single or multiple depots with possible side constraints. The most studied problem is the travelling salesman problem (TSP) when one vehicle visits all customers in one single trip (see Laporte [1] and Applegate et al. [2]). A more practical problem known as the vehicle routing problem (VRP) is when a fleet of vehicles with a limited load capacity needs to deliver quantities of goods to a set of customers so that everyone's demand is met (see Toth and Vigo [3]).

One of the type of constructive heuristics proposed for the VRP is the route-first cluster-second heuristics. This principle, also known as the order-first split-second principle (see Prins et al. [4]), was formalised in 1983 by Beasley [5]. It consists in generating a giant tour, building a cost network then finding a shortest path to partition optimally the giant tour. Variations on the way the cost network is built permitted to partition a giant tour with regard to several routing variants. Golden et al. [6] proposed a similar approach to tackle the fleet size and mix VRP while Ulusoy [7] applied it to the fleet size and mix arc routing problem. More recently, the route-first cluster-second principle has been embedded into various heuristic schemes (see Prins [8], Imran et al. [9] and Villegas et al. [10]) and applied successfully to multi-attribute routing problems (see Vidal et al. [11]). An interesting and informative recent review on route-first cluster-second methods can be found in Prins et al. [4].

These methods perform efficiently in many cases but can be limited due to the presence of basic constraints such as fixing (not limiting) the number of vehicles or by considering further resources such as a heterogeneous fleet. A discussion about the efficiency and limitations of the existing route-first cluster-second methods can also be found in Prins et al. [4].

Recently, the partitioning of a giant tour for the multiple traveling salesman problem (mTSP) was formulated as an integer linear program (ILP) (see [12]). It was proven that the obtained formulation is solvable in polynomial time. An extension to the mTSP with limitations on the number of customers visited per vehicle was then presented and an integration to the variable neighbourhood search (VNS) provided interesting results.

In this study, we extend this idea to the case of the capacitated vehicle routing problem (CVRP) with a fixed number of vehicles following the route-first cluster-second principle. As the obtained ILP formulation reveals intractable on experiments for a conventional solver, we propose a Lagrangian relaxation method called *Lagrangian split* (Ls) to tackle the problem. As we will show later, the proposed ILP formulation is flexible and thus, Ls could be extended to deal with a variety of routing problems. Ls is a subgradient method which consists of a repair and a local search algorithms designed to repair and improve an infeasible partitioning of a giant tour. The mathematical validity of the repair mechanism and its time complexity are also provided. An integration of Ls into a VNS scheme is presented as an illustration of the use of Ls within a metaheuristic. Computa-

* Corresponding author.
E-mail address: cherifmouaouia.bouzid@gmail.com (M.C. Bouzid).

http://dx.doi.org/10.1016/j.cor.2016.02.009
0305-0548/

tional experiments are conducted to demonstrate that Ls provides encouraging results when applied on benchmark instances. In addition, the integration of Ls into a metaheuristic scheme, though it is still in its infancy, has shown to yield good results when compared to the state-of-the-art methods.

The remainder of the paper is organised as follows: the next section is devoted to the description of the split problem and its ILP formulation. In Section 3, the Lagrangian split method and its main components are presented with an illustrative example. In Section 4, Ls is embedded into a VNS scheme followed by experiments in Section 5. Finally, conclusions and suggestions are given in Section 6.

2. An ILP formulation for the splitting of a giant tour

2.1. Definitions

Consider a single-depot routing problem with $V'=\{1,\dots,n\}$ being the set of customers, 0 the depot and $V=\{0\}\cup V'$ the set of nodes. Let $c(i,j)$ be the travel cost from node i to node j. Let m be the number of vehicles located at the depot each having a resource capacity Q. Let $q(u)$ be the demand of customer u with $0\le q(u)\le Q$.

Given a positive integer a, we denote the term $[a]$ as the set $\{1,\dots,a\}$, and a *position* as an element of $[n]$. A *giant tour* is a Hamiltonian circuit over the elements of V'. A giant tour T can be denoted as a permutation $(T_1,\dots,T_n)$ where T_i belongs to V' for all positions i. Note that in our definition, the depot is not assumed to belong to T. Therefore, *there is neither a first nor a last customer in T*. The length of T is given by $L(T)=\sum_{k=1}^{n-1} c(T_k,T_{k+1})+c(T_n,T_1)$.

We define a summation operator which takes into account the circular nature of T. Given an array $w=(w_1,\dots,w_n)$ of values indexed on the set $[n]$, the *n-circular sum of the elements of w* is defined by

$$\textcircled{n}_{k=i}^{j} w_k=\begin{cases} w_i+\cdots+w_j & =\sum_{k=i}^{j} w_k \quad \text{if } i\le j \\ w_i+\cdots+w_n+w_1+\cdots+w_j & =\sum_{k=i}^{n} w_k+\sum_{k=1}^{j} w_k \quad \text{if } i>j \end{cases}, \quad i,j\in[n]$$

For example, $⑩_{k=3}^{5} w_k=w_3+w_4+w_5$ while $⑦_{k=5}^{2} w_k=w_5+w_6+w_7+w_1+w_2$.

Moreover, we consider the indexing to be circular throughout the paper (i.e., the next value after w_n is w_1 and the value before w_1 is w_n).

Let q_k be the demand of the customer T_k, $k\in[n]$ (i.e., $q_k=q(T_k)$). The cumulated demand over a subsequence $(T_i,\dots,T_j)$ of T is defined by $Q_i^j=\textcircled{n}_{k=i}^{j} q_k$.

A route $r=(0,r_1,r_2,\dots,r_{|r|},0)$ is a Hamiltonian circuit which covers the depot and a subset of customers where $|r|$ is the number of customers in r and $r_k\in V', \forall k\in[|r|]$. The cost of the route r is defined by $L(r)=c(0,r_1)+\sum_{k=1}^{|r|-1} c(r_k,r_{k+1})+c(r_{|r|},0)$. The cumulated demand of the customers belonging to the route r equals $Q(r)=\sum_{k=1}^{|r|} q(r_k)$. The route r is feasible if $Q(r)\le Q$. A routing $x=(r^1,r^2,\dots,r^m)$ is a set of m routes that cover all the customers exactly once. The cost of x is given by $L(x)=\sum_{k=1}^{m} L(r^k)$. The routing x is feasible if all its routes are feasible. The capacitated vehicle routing problem (CVRP) consists in finding a feasible routing x^* with a minimum total cost $L(x^*)$.

2.2. Formulation of the split problem associated with the CVRP

Let $\delta_i=c(T_i,0)+c(0,T_{i+1})-c(T_i,T_{i+1})$ be the cost of inserting the depot after the node T_i into the giant tour T. In other words, T is split at the node T_i. Let y_i be a binary variable which equals 1 if T is split at the node T_i and 0 otherwise, $i\in[n]$. A *split position* is a position $i\in[n]$ for which $y_i=1$. In Fig. 1, the descending arrow means that the depot is inserted into T after the position i.

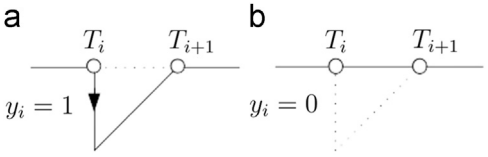


Fig. 1. (a) i is a split position. (b) i is not a split position.

Given a split position i, a position j is said to be *compatible with* i if $Q_{i+1}^j\le Q$. In other words, if a split occurs at position i, the subsequence $(T_{i+1},\dots,T_j)$ has a total demand not larger than Q. Thus, performing a split at position j would produce a feasible route $(0,T_{i+1},\dots,T_j,0)$. We call the set of all positions compatible with i, the *compatibility set of* i. This is defined by $\mathcal{C}(i)=\{j\in[n]: Q_{i+1}^j\le Q\}$.

We also define for every position $i\in[n]$, the set $\mathcal{C}^{-1}(i)=\{j\in[n]: i\in\mathcal{C}(j)\}$. This set corresponds to the positions of $[n]$ for which i is a compatible position.

The split problem can then be formulated as follows:

$$Split(T)\quad z^*=\min \sum_{i\in[n]}\delta_i y_i \tag{1}$$

s.t.

$$\sum_{i\in[n]} y_i=m \tag{2}$$

$$\sum_{j\in\mathcal{C}(i)} y_j\ge y_i,\quad \forall i\in[n] \tag{3}$$

$$y_i\in\{0,1\},\quad \forall i\in[n] \tag{4}$$

The objective is to minimize the sum of the insertion costs. The value $z^*+L(T)$ is the cost of the routing obtained by optimally partitioning T. The constraint (2) ensures that exactly m routes are used. The constraints (3) guarantee that in case a split occurs at a position i then at least one split must occur in one of the positions compatible with i. Indeed, if $y_i=1$ then $\sum_{j\in\mathcal{C}(i)} y_j\ge 1$, otherwise the constraint is redundant. The constraints (4) refer to the binary nature of the decision variables.

In case the number of vehicles is not limited but a fixed charge F is associated with the use of a vehicle, the objective becomes $\min\sum_{i\in[n]}(\delta_i+F)y_i$ and constraint (2) is exchanged with $\sum_{i=1}^{n} y_i\ge 1$ to ensure that there is at least one route. In this latter formulation, the number of vehicles can be minimized simply by setting the objective to $\sum_{i\in[n]} y_i$.

Depending on the definition of the compatibility set $\mathcal{C}(i)$, *Split* (T) can be extended to consider other constraints such as:

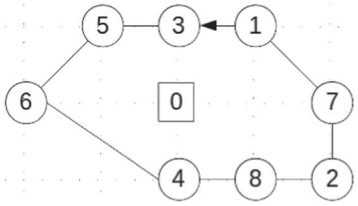
(i) *Distance restriction*: Let D be the route length limit and $D_{i+1}^j=c(0,T_{i+1})+\textcircled{n}_{k=i+1}^{j-1} c(T_k,T_{k+1})+c(T_j,0)$ be the length of the route $(0,T_{i+1},\dots,T_j,0)$. Then, $\mathcal{C}(i)=\{j\in[n]: D_{i+1}^j\le D\}$ for all positions i,

(ii) *Time windows*: Let $t(u,v)$, $[a_j,b_j]$ and s_j be the travel time from the customer u to the customer v $(u,v\in V')$, the visiting period and the service time of the customer T_j $(j\in[n])$ respectively. The visiting time of the customer T_j on the route containing the sequence $(0,T_{i+1},\dots,T_j)$ is given by $T_{i+1}^j=t(0,T_{i+1})$ if $j=i+1$ and $T_{i+1}^j=\max\{T_{i+1}^{j-1},a_{j-1}\}+s_{j-1}+t(T_{j-1},T_j)$ when $j\in[n]\setminus\{i+1\}$. Thus, $\mathcal{C}(i)=\{j\in[n]: T_{i+1}^j\le b_j\}$ for all positions i.

2.2.1. An illustrative example

Fig. 2 illustrates an instance of *Split*(T) with 8 customers and 3 vehicles having a load capacity $Q=5$. The traveling cost between two nodes is given by the Euclidean distance. Note that the

a



b

n=8, m=3, Q=5 and $L(T)$=10.06.

i	T_i	q_i	δ_i	$\mathcal{C}(i)$	$\mathcal{C}^{-1}(i)$
1	1	1	1.41	2,3	5,6,7,8
2	3	2	1.41	3,4	7,8,1
3	5	3	2.00	4,5	1,2
4	6	2	0.76	5,6,7	2,3
5	4	3	1.41	6,7,8,1	3,4
6	8	1	2.65	7,8,1	4,5
7	2	1	3.24	8,1,2	4,5,6
8	7	2	2.00	1,2	5,6,7

c

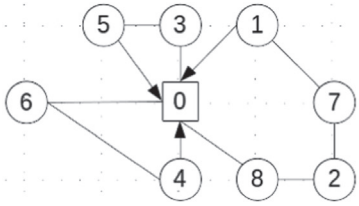


Fig. 2. Example of a *Split*(T) instance. (a) The giant tour. (b) Instance description and the corresponding *Split*(T) parameters. (c) The routing corresponding to the optimal solution of *Split*(T).

considered instance is Euclidean for an illustrative purpose only and that the proposed approach does not make any assumption on the cost matrix c. The length of the giant tour $T=(1,3,5,6,4,8,2,7)$ is $L(T)=10.06$. The problem is to split T into 3 feasible routes while minimizing the cost of inserting the depot into the giant tour. The compatibility set $\mathcal{C}(3)$ is equal to $\{4,5\}$. This means that if a split occurs at position 3 (which is occupied by customer 5), then a split must occur at one of the positions 4 or 5 in order to keep the route starting from position 4 feasible. The optimal solution of *Split*(T), found using the GLPK solver [13], is given by $y^*=(1,0,1,0,1,0,0,0)$ with a value of $z^*=4.82$. The obtained routing has a cost of $L(T)+z^*=14.88$.

Note that as *Split*(T) does not assume the presence of the depot in the giant tour T, the optimality of the splitting obtained is not subjected to the order in which T is taken. In addition, *Split*(T) permits to *fix* (not just limiting) the number of vehicles used in the routing. To the best of our knowledge, the existing split procedures permit only to limit the number of vehicles used in the routing (not fixing that number). This is achieved using a general form of Bellman's algorithm by limiting the number of arcs used during the building of the shortest path (see Prins et al. [4]).

3. Lagrangian split of a giant tour T

Split(T) can be solved for small size instances by a standard ILP software such as GLPK [13], however, its resolution becomes impractical for relatively larger instances.

The Lagrangian relaxation (LR) is a powerful technique for solving ILPs. Given a minimisation ILP with a set of constraints partitionable into "easily" satisfiable constraints and harder ones, LR consists in relaxing the hard constraints into the objective by multiplying them with a penalty factor called the *Lagrangian multiplier*. The resulting model is thus easier to solve than the original problem. The solution of the relaxed model, called the *lower-bound program*, gives a valuable data as it corresponds to a lower bound for the original ILP. If the relaxed constraints are suitably chosen, the optimal value to the lower-bound program provides a tight lower bound for the original ILP which can then be used for pruning the search tree in a branch and bound procedure. A feasible solution for the ILP can also be derived from the solution of the lower-bound program in a process called *Lagrangian heuristic*. Several techniques have been proposed in the literature to determine the Lagrangian multipliers which maximizes the lower bound program, among them the *sub-gradient method* which makes use of the convexity of the lower-bound program objective. An old but still topical overview of Lagrangian relaxation can be found in Fisher [14]. Lagrangian relaxation has been widely used for solving combinatorial optimisation problems, see for instance Cornuejols et al. [15], Kohl and Madsen [16] and more recently Nezhad et al. [17]. To the best of our knowledge, LR has never been applied to solve the problem of partitioning a giant tour in the context of routing problems.

In this section, we present a LR method for the resolution of *Split*(T) which we call *Lagrangian split* (Ls).

3.1. Overview of the Lagrangian split

Ls is a subgradient method that solves *Split*(T) based on a Lagrangian relaxation $LB(\lambda)$ of this model where λ is a Lagrangian multiplier (see Section 3.2). In a previous contribution (see [12]), $LB(\lambda)$ was proved to be solvable in polynomial time. Initially, the Lagrangian multiplier is set to a value λ^0. For each iteration t, $LB(\lambda^t)$ is solved resulting in a solution to *Split*(T). If that solution is not feasible for *Split*(T), it is repaired using a polynomial algorithm (see Section 3.3.1). If that solution is still not feasible then Ls returns that T is unsplittable using m vehicles (see Theorem 1). Otherwise, a local search is applied to improve the repaired solution (see Section 3.3.3). The combination of the repair mechanism with the local search is a Lagrangian heuristic (see Section 3.3). Once the Lagrangian heuristic is performed, the Lagrangian multiplier is updated and t is incremented. The overall process terminates whenever a number of non-improving iterations passed or when *Split*(T) is proven infeasible (see Section 3.4).

We now describe the main components of Ls which are the Lagrangian relaxation of *Split*(T), the Lagrangian heuristic and the subgradient method.

3.2. Lagrangian relaxation of Split(T)

We define a *split solution* as a vector $y\in\{0,1\}^n$ which satisfies the constraint (2) in *Split*(T). A split solution y is *feasible* if it satisfies the constraints (3) which are harder to tackle than (2). Therefore, we propose to attach Lagrangian multipliers $\lambda_j\geq 0$ to (3) and to relax them into the objective function to obtain the following lower bound program:

$$LB(\lambda)\ \ z_{LB}(\lambda)=\min \sum_{i\in[n]}\delta_i y_i+\sum_{i\in[n]}\lambda_i\Big(y_i-\sum_{j\in\mathcal{C}(i)}y_j\Big)$$
$$\text{s.t.}$$
$$(2),(4)$$

$z_{LB}(\lambda)$ is a lower bound for z^*. Indeed, z^* is greater or equal to the optimal value of $LB(\lambda)$ subject to (3) as $\lambda_i(y_i-\sum_{j\in\mathcal{C}(i)}y_j)\leq 0,\ \forall i\in[n]$, which is greater or equal to $z_{LB}(\lambda)$ as we are relaxing (3) in a minimization problem.

It follows that:

$$\sum_{i\in[n]}\delta_i y_i+\sum_{i\in[n]}\lambda_i\left(y_i-\sum_{j\in\mathcal{C}(i)}y_j\right)=\sum_{i\in[n]}(\delta_i+\lambda_i)y_i-\sum_{i\in[n]}\sum_{j\in\mathcal{C}(i)}\lambda_i y_j$$
$$=\sum_{i\in[n]}(\delta_i+\lambda_i)y_i-\sum_{j\in[n]}\sum_{i\in\mathcal{C}^{-1}(j)}\lambda_i y_j$$

$$= \sum_{i \in [n]} (\delta_i + \lambda_i)\, y_i - \sum_{i \in [n]} \sum_{j \in \mathcal{C}^{-1}(i)} \lambda_j y_i$$

$$= \sum_{i \in [n]} \left(\delta_i + \lambda_i - \sum_{j \in \mathcal{C}^{-1}(i)} \lambda_j \right) y_i$$

Let $\Delta_i = \delta_i + \lambda_i - \sum_{j \in \mathcal{C}^{-1}(i)} \lambda_j$ then $LB(\lambda)$ becomes:

$$LB(\lambda)\ \ z_{LB}(\lambda) = \min \sum_{i \in [n]} \Delta_i\, y_i$$
$$\text{s.t. } (2), (4)$$

which is a polynomially solvable 0–1 knapsack problem. Indeed, an optimal solution to $LB(\lambda)$ can be found simply by selecting the m smallest Δ_i, assigning 1 to the corresponding y_i and 0 to the remaining ones. This can be performed in $O(nm)$ (see [12]).

Note that $LB(\lambda)$ has the integral property, which means that the optimal value of $LB(\lambda)$ is not altered by dropping the integrality conditions on its variables (see Proposition 2.2 in [12]). Therefore, the maximum value attainable by $z_{LB}(\lambda)$ equals the optimal value of the LP relaxation of *Split*(T) when it is feasible (see Theorem 2 in Geoffrion [18]). Thus, we do not solve $LB(\lambda)$ to bound z^* but to build a split solution (possibly feasible) then to repair it to deduce an expected good feasible split solution for *Split*(T). This Lagrangian heuristic is described in the next section.

3.3. Lagrangian heuristic

Let $\bar{y}$ be an optimal solution of $LB(\bar{\lambda})$ w.r.t. $\bar{\lambda} \geq 0$ and assume that $\bar{y}$ does not satisfy (3) i.e. $\bar{y}$ is an infeasible split solution. Then, there exists a split position $i \in [n]$ such that $y_i > \sum_{j \in \mathcal{C}(i)} y_j$. As the decision variables are binary and $y_i = 1$ then $y_j = 0, \forall j \in \mathcal{C}(i)$. In other words, a split occurred at node T_i but no split occurred sufficiently early after that (i.e., not in the set $\mathcal{C}(i)$).

Fig. 3 represents an infeasible split solution with three consecutive split positions v_1, v_2 and v_3. The problem is that v_2 is too far from v_1 i.e., $v_2 \notin \mathcal{C}(v_1)$). The idea is to roll back v_2 until it takes a position belonging to $\mathcal{C}(v_1)$ while ensuring that v_3 remains in $\mathcal{C}(v_2)$. We call the set of positions which fulfil that condition a *fixing region* and we define it by $\mathcal{R}(v_1, v_3) = \{j \in [n] : j \in \mathcal{C}(v_1) \text{ and } v_3 \in \mathcal{C}(j)\} = \mathcal{C}(v_1) \cap \mathcal{C}^{-1}(v_3)$. In other words, $\mathcal{R}(v_1, v_3)$ is the set of positions compatible for v_1 and for which v_3 is compatible.

In the next two subsections, we describe a repair algorithm which uses the fixing region to fix an infeasible split solution followed by a local search algorithm to improve a feasible split solution.

3.3.1. The repair algorithm

In brief, the repair algorithm consists in moving the split positions around an infeasible giant tour T until every split position is compatible with the previous one. If it is not possible then *Split*(T) is infeasible. In order to present and justify this algorithm, we first introduce some additional concepts.

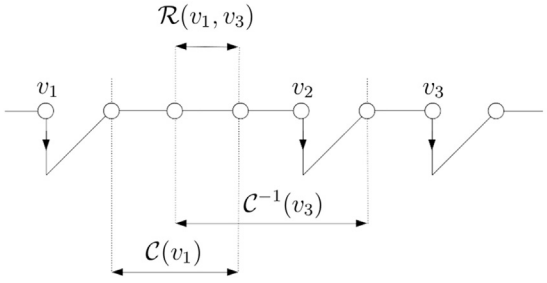


Fig. 3. An infeasible split with three consecutive split positions v_1, v_2 and v_3 and the fixing region $\mathrm{R}(v_1, v_3)$.

A *split vector* v is a circularly sorted vector of m components in $[n]$ without any duplicate value. Every component of v represents a split position. Formally, we define v as a vector of m components in $[n]$ such that there exists an index p in $[m]$ which satisfies $v_{p-1} > v_p$ and $v_i < v_{i+1}, \forall i \in [m] \backslash \{p-1\}$. The element v_p is called a *pivot* and p refers to its position. For example, $(5, 9, 15, 1, 3)$ is a split vector with a pivot equal to 1 ranked at $p = 4$ but $(9, 3, 4, 1, 5, 2)$ is not a split vector as it has no pivot. To every split vector v, we can associate a split solution y defined by $y_i = 1$ if $i \in \{v_1, \ldots, v_m\}$, 0 if $i \in [n] \backslash \{v_1, \ldots, v_m\}$. Similarly, to any split solution y, we can associate a split vector v defined by $v_1 = \min\{j \in [n] : y_j = 1\}$ and $v_i = \min\{j \in [n] \backslash \{v_1, \ldots, v_{i-1}\} : y_j = 1\}$ for $i \in [m] \backslash \{1\}$. A split vector v is *feasible* if its associated split solution y is feasible. The cost of a split vector v is given by $c(v) = \sum_{i \in [m]} \delta_{v_i}$. Hence, we can describe a split of T using either a split solution or a split vector.

Proposition 1 characterizes a feasible split vector using the compatibility sets.

Proposition 1. *Let v be a split vector. v is feasible if and only if $v_{i+1} \in \mathcal{C}(v_i)$ for all $i \in [m]$.*

Proof. Let v be a split vector and y be its associated split solution. On one hand, assume that v is feasible but there exists an $i \in [m]$ such that $v_{i+1} \notin \mathcal{C}(v_i)$. Then, $y_j = 0$ for all j in $\mathcal{C}(v_i)$ and $y_{v_i} = 1 > \sum_{j \in \mathcal{C}(v_i)} y_j = 0$ which is absurd. On the other hand, assume that $v_{i+1} \in \mathcal{C}(v_i)$ for all $i \in [m]$. By the definition of v and y, $\sum_{i \in [n]} y_i = m$. Given a $k \in [m]$, the following inequality holds $y_{v_k} \leq y_{v_{k+1}} + \sum_{j \in \mathcal{C}(v_k) \backslash \{v_{k+1}\}} y_j$ as the left term equals 1 and the right term equals at least 1. Also, $y_i = 0$ for all $i \in [n] \backslash \{v_1, \ldots, v_m\}$. Hence, $y_i \leq \sum_{j \in \mathcal{C}(i)} y_j$ holds for all $i \in [n]$ and y is feasible likewise v.□

We now define a function which measures the distance between two positions in a giant tour. Let $\eta_n(i, j)$ be the number of elements from the position $i+1$ to the position j on a giant tour of size n. For simplicity, $\eta_n(i, j)$ will be simply denoted $\eta(i, j)$ as there is no ambiguity on the size of the giant tours. Proposition 2 provides a way to calculate $\eta(i, j)$ and an injectivity-like property for this function. Let a and n be two integers with $n \neq 0$. The common residue of the division of a by n is given by $\mathrm{mod}(a, n) = a - \lfloor \frac{a}{n} \rfloor n$ where $\lfloor \frac{a}{n} \rfloor$ is the greatest integer less or equal to $\frac{a}{n}$. It is known that $0 \leq \mathrm{mod}(a, n) < n$.

Proposition 2.

(i) $\forall (i, j) \in [n]^2 :\ \eta(i, j) = \mathrm{mod}(j - i, n)$
(ii) $\forall (i, j, k) \in [n]^3$: $\eta(i, j) = \eta(i, k)$ *if and only if* $j = k$

Proof.

(i) It is clear that $|j - i| < n$. If $j \geq i$ then $0 \leq \frac{j-i}{n} < 1$ and $\mathrm{mod}(j - i, n) = j - i = \eta(i, j)$ as $\lfloor \frac{j-i}{n} \rfloor = 0$. Otherwise, if $j < i$ then $-1 < \frac{j-i}{n} < 0$ and $\mathrm{mod}(j - i, n) = (n - i) + j = \eta(i, j)$ as $\lfloor \frac{j-i}{n} \rfloor = -1$.

(ii) If $\mathrm{mod}(j - i, n) = \mathrm{mod}(k - i, n)$ then $j - k = \left(\lfloor \frac{j-i}{n} \rfloor - \lfloor \frac{k-i}{n} \rfloor \right) n$. The second term of this equation depends on the order of i, j and k. In any case, the second term is either equal to 0 providing the desired result or $\pm n$ which is absurd given the domain of j and k. The reciprocal is trivial.□

Let $\overline{\eta}$ and $\underline{\eta}$ be two functions defined on the set $[n]$ by $\overline{\eta}(i) = \mathrm{argmax}_{j \in \mathcal{C}(i)} \eta(i, j)$ and $\underline{\eta}(i) = \mathrm{argmax}_{j \in \mathcal{C}^{-1}(i)} \eta(j, i)$. Literally, $\overline{\eta}(i)$ is the furthest position compatible with the position i while $\underline{\eta}(i)$ is the furthest position for which i is compatible. *Tightening* a split vector v means to set $v_{i+1} := \overline{\eta}(v_i)$ for all i in $[m] \backslash \{p-1\}$. In that case, v is said to be *tight*. Given a split vector $\bar{v}$, a split vector u *is induced by*

$\overline{v}$ if $u_p = \overline{v}_p$ and $\eta(u_p, u_{p-1}) \leq \eta(\overline{v}_p, \overline{v}_{p-1})$. Note that any split vector induces itself.

Proposition 3 states that no feasible split vector can be induced by an infeasible tight split vector.

Proposition 3. *Let $\overline{v}$ be a tight split vector. If $\overline{v}_p \notin \mathcal{C}(\overline{v}_{p-1})$ then no feasible split vector can be induced by $\overline{v}$.*

Proof. Let $\overline{v}$ be a tight split vector such that $\overline{v}_p \notin \mathcal{C}(\overline{v}_{p-1})$. Assume v is a feasible split vector induced by $\overline{v}$ then $v_p = \overline{v}_p$, $\eta(v_p, v_{p-1}) \leq \eta(v_p, \overline{v}_{p-1})$ and $v_p \in \mathcal{C}(v_{p-1})$. Assume that $\eta(v_p, v_{p-1}) < \eta(v_p, \overline{v}_{p-1})$ then

$$q_{v_{p-1}+1} + \cdots + q_{\overline{v}_{p-1}} + q_{\overline{v}_{p-1}+1} + \cdots + q_{v_p} \leq Q \text{ as } \eta(v_p, v_{p-1}) < \eta(v_p, \overline{v}_{p-1}) \text{ and } v_p \in \mathcal{C}(v_{p-1})$$

$$q_{\overline{v}_{p-1}+1} + \cdots + q_{v_p} > Q \text{ as } v_p = \overline{v}_p \text{ and } \overline{v}_p \notin \mathcal{C}(\overline{v}_{p-1})$$

Subtracting the second inequality from the first one implies that $\sum_{k=v_{p-1}+1}^{\overline{v}_{p-1}} q_k < 0$ which is absurd as $q_k \geq 0$ for all k in $[n]$. By Proposition 2, the case where $\eta(v_p, v_{p-1}) = \eta(v_p, \overline{v}_{p-1})$ implies that $v_{p-1} = \overline{v}_{p-1}$. Therefore, $v_p \in \mathcal{C}(v_{p-1})$ as v is feasible and $v_p \notin \mathcal{C}(v_{p-1})$ because $\overline{v}_p \notin \mathcal{C}(\overline{v}_{p-1})$, $v_p = \overline{v}_p$ and $v_{p-1} = \overline{v}_{p-1}$, contradiction.□

Lemma 1 provides a sufficient condition for *Split*(T) to be infeasible. It states that if no tight split vector is feasible then the problem has no feasible solution.

Lemma 1. *Let $\overline{v}^1$, $\overline{v}^2$, …, $\overline{v}^n$ be n tight split vectors such that $\overline{v}^j_p = j$, $j \in [n]$. If $\overline{v}^j_p \notin \mathcal{C}(\overline{v}^j_{p-1})$ for all j in $[n]$ then Split(T) is infeasible.*

Proof. Assume that $\overline{v}^j_p \notin \mathcal{C}(\overline{v}^j_{p-1})$, $j \in [n]$ and that *Split*(T) is feasible. Let v be a feasible split vector then v is not induced by $\overline{v}^j$ for any j (see Proposition 3). However, there exists a k in $[n]$ equal to v_p such that $v_p = \overline{v}^k_p$ otherwise $v_p \notin [n]$ which is absurd. Assume without loss of generality that $k = p = 1$ and denote $\overline{v}^1$ by $\overline{v}$. In summary, $v_1 = \overline{v}_1 = 1$, v is feasible but not induced by $\overline{v}$ which is tight and infeasible. As v is not induced by $\overline{v}$ then $\eta(v_1, v_m) > \eta(v_1, \overline{v}_m)$. By Proposition 2, $\overline{v}_m \neq v_m$. If $\eta(v_1, v_{m-1}) \leq \eta(v_1, \overline{v}_{m-1})$ then $q_{v_{m-1}+1} + \cdots + q_{\overline{v}_{m-1}+1} + \cdots + q_{\overline{v}_m} + \cdots + q_{v_m} \leq Q$ as $v_m \in \mathcal{C}(v_{m-1})$. In particular, $q_{\overline{v}_{m-1}+1} + \cdots + q_{\overline{v}_m} + \cdots + q_{v_m} \leq Q$ which contradicts the fact that $\overline{v}_m = \overline{\eta}(\overline{v}_{m-1})$. Thus, $\eta(v_1, v_{m-1}) > \eta(v_1, \overline{v}_{m-1})$ and $\overline{v}_{m-1} \neq v_{m-1}$ by Proposition 2. Following a similar reasoning leads to $\eta(v_1, v_3) > \eta(v_1, \overline{v}_3)$ and $\overline{v}_3 \neq v_3$. By the definition of $\overline{\eta}(.)$, we have $\eta(v_1, v_2) \leq \eta(v_1, \overline{v}_2)$. It follows from $v_3 \in \mathcal{C}(v_2)$ that $q_{v_2+1} + \cdots + q_{\overline{v}_2+1} + \cdots + q_{\overline{v}_3} + \cdots + q_{v_3} \leq Q$. In particular, $q_{\overline{v}_2+1} + \cdots + q_{\overline{v}_3} + \cdots + q_{v_3} \leq Q$ which contradicts the fact that $\overline{v}_3 = \overline{\eta}(\overline{v}_2)$. Therefore, v cannot exist and *Split*(T) admits no feasible solution.□

3.3.2. The algorithm

This enables us to present a repair algorithm (see Algorithm 1) which either produces a feasible split vector starting from an infeasible one or determines the infeasibility of *Split*(T). Algorithm 1 requires initially a split vector v, the desired number of routes m and the *Split*(T) parameters namely δ_i, $\mathcal{C}(i)$ and $\mathcal{C}^{-1}(i)$ for all i in $[n]$. The algorithm returns a split vector v and a boolean *feas* which equals true when v has been repaired or false if *Split*(T) is proven infeasible. Initially, *feas* equals false, a split vector's index k is set to 1 and a counter t which equals $m-1$ when v is tight is set to 0 (line 1). In a first loop, the algorithm iterates until v is feasible or it is tight and infeasible (repeat-loop 2–10). Starting from the split position $j=k$, `detectIncompatibility`(v, k) returns the first index j in $[m]$ such that $v_{j+1} \notin \mathcal{C}(v_j)$ which means that v is infeasible because of an incompatibility of v_{j+1} toward v_j. If such an index does not exist then v is feasible and the algorithm stops (line 4). Otherwise, the problematic index is stored in k and the algorithm tests the fixing region $R(v_k, v_{k+2})$. If it is not empty then it splits at the fixing position having a minimum δ_j, the tightness counter t is reset to 0 and the fixing continues from v_{k+2} as no more incompatibility exists between the split positions v_k, v_{k+1} and v_{k+2} (line 6). If there is no fixing position for v_{k+1} (line 7) then the algorithm splits at the furthest position compatible with v_k and the tightness is incremented. In that case, v_{k+1} is present in $\mathcal{C}(v_k)$ but v_{k+2} is still outside $\mathcal{C}(v_{k+1})$. This is why the algorithm seeks an incompatibility from v_{k+1} at the next iteration (line 8). After the first loop, if t equals m then v is tight and infeasible. In that case, v satisfies $v_{i+1} = \overline{\eta}(v_i)$ for all i in $[m]\backslash\{k\}$. In order to find a feasible split vector, the algorithm generates all the remaining tight split vectors by varying the initial split position v_1. This value is set initially to the position coming immediately after v_{k+1} (line 12). The routine `tighten`(v) sets $v_{j+1} := \overline{\eta}(v_j)$ for all j in $[m-1]$. At each iteration of the second repeat-loop (lines 13–20), if a feasible tight split vector is produced then the algorithm terminates (lines 16–20), otherwise, it generates the next tight split vector. The loop stops either when a feasible split vector is reached or when all the tight split vectors have been examined without fixing the infeasibility (line 20) in which case the split problem is infeasible (see Lemma 1).

Algorithm 1. Repair algorithm.

Input : split vector v; integer m and *Split*(T) parameters.
Output : split vector v and a boolean *feas*.

1 *feas* := **false**, k := 1 and t := 0;
2 **repeat**
3 k := detectIncompatibility (v, k);
4 **if** k *does not exist* **then** *feas* := **true**;
5 **else if** $R(v_k, v_{k+2}) \neq \emptyset$ **then**
6 v_{k+1} := argmin $\{\delta_j : j \in R(v_k, v_{k+2})\}$, t := 0 and k := $k+2$;
7 **else**
8 $v_{k+1} := \overline{\eta}(v_k)$, t := $t+1$ and k := $k+1$;
9 **end if**
10 **until** *feas*= **true or** t = m;
11 **if** t = m **then**
12 α := v_{k+1} and i := $\alpha+1$;
13 **repeat**
14 v_1 := i and tighten (v);
15 **if** $R(v_m, v_2) \neq \emptyset$ **then**
16 v_1 := argmin $\{\delta_j : j \in R(v_m, v_2)\}$ and set *feas* := **true**;
17 **else**
18 i := $i+1$;
19 **end if**
20 **until** *feas* = **true or** j = α;
21 **end if**
22 **return** v and *feas*;

Theorem 1. *Algorithm 1 returns a feasible split vector or determines the infeasibility of Split(T) in O(n(m+h)) where* $h = \left\lceil \frac{Q}{\min_{i \in [n]} q_i} \right\rceil$.

Proof. Assume without loss of generality that the first incompatibility detected at line 3 is for $k=1$ which means that $v_2 \notin \mathcal{C}(v_1)$. The algorithm iterates on the first repeat-loop and fixes the incompatibilities until reaching a situation where $v_{j+1} \in \mathcal{C}(v_j)$ for all $j \in [m-1]$. If $\mathcal{R}(v_m, v_2) \neq \emptyset$ then setting v_1 to any position in $\mathcal{R}(v_m, v_2)$ produces a feasible split vector and terminates the algorithm. Otherwise, v_1 is set to $\overline{\eta}(v_m)$. Idem for v_2 and so on. If the algorithm reaches a position v_k such that $\mathcal{R}(v_k, v_{k+2}) \neq \emptyset$ then setting v_{k+1} to any position in $\mathcal{R}(v_k, v_{k+2})$ fixes the infeasibility

and the algorithm stops. If no such v_k is reached then the algorithm keeps iterating until v becomes tight and t equals $m-1$. On the next iteration, if $\mathcal{R}(v_m, v_2)$ is still empty then t is set to m and the first repeat-loop terminates. In that situation, the split vector v has been explored twice, it became an infeasible tight split vector and the algorithm enters the second repeat-loop as t equals m. All the remaining tight split vectors are generated until reaching a feasible one (lines 12–20). If no feasible tight split vector exists then the split problem is infeasible (see Lemma 1). This is the worst case and we now consider the induced complexity. The first loop ends after two examinations of v which is in $O(m)$. The worst case for detecting an incompatibility is to find one incompatibility after obtaining $m-1$ compatibilities then line 3 is in $O(m)$. The sizes of $\mathcal{C}(i)$ and $\mathcal{C}^{-1}(i)$ are the order of $O(h)$ for any position i as h corresponds to the maximum route size. Therefore, $\mathcal{R}(v_k, v_{k+2})$ is the order of $O(h)$ which is also the complexity of finding the j which minimizes δ_j for that set (line 6). Thus, the first repeat-loop (lines 2–10) is in $O(m(m+h))$. Regarding the second repeat-loop, there are n possible tight split vectors, one for every starting position. It takes $O(m)$ operations to tighten one split vector and $O(h)$ operations to find the minimizing split position in the fixing region. Therefore, the second repeat-loop (lines 12–20) takes $O(n(m+h))$. Finally, the overall complexity of the algorithm is $O(m(m+h)+n(m+h))$ which reduces to $O(n(m+h))$ as $m \leq n$.□

3.3.3. The local search algorithm

We now propose a first improvement-based local search which uses the above ideas to improve a given feasible split vector v. Algorithm 2 requires a feasible split vector v, the desired number of routes m and the *Split*(T) parameters. It returns a feasible split vector of a quality equal or better than the initial split vector. The algorithm starts by setting a split vector's index k to 1 (line 1). Then, it iterates over v in a cyclic fashion while the split is improved. For every split position v_i, the algorithm tests whether there exists a better position for the next split position v_{i+1} by analysing the set $\mathcal{R}(v_i, v_{i+2})$ of candidates positions for v_{i+1} (lines 5–7). If so, the move is performed and the search restarts from v_{i+1} (line 8). The cost of v never increases in the algorithm. The process terminates whenever a complete examination of v is performed without any improvement.

Algorithm 2. Local search for a feasible split vector.

```
   Input  : feasible split vector v; integer m and Split(T) parameters.
   Output : feasible split vector v.
 1 k :=1, improv := true ;
 2 while improv do
 3 |  improv := false, i := k;
 4 |  repeat
 5 |  |  l :=argmin {δ_j : j ∈ R (v_i,v_i+2 )};
 6 |  |  improv := δ_l < δ_v_i+1 ;
 7 |  |  if improv then
 8 |  |  |  v_i+1 := l and k := i +1;
 9 |  |  end if
10 |  |  i := i +1;
11 |  until i = k or improv;
12 end while
13 return v;
```

A best improvement version was also tested but provided the same solution quality while requiring a slightly greater CPU time. Therefore, we decided to present only the first improvement version.

3.3.4. An illustrative example for the Lagrangian heuristic

Consider the *Split*(T) instance depicted in the top of Fig. 4. For simplicity, the positions corresponding to the customers are given by $T_i = i$ for all i in $[n]$. The traveling cost between two nodes is given by the Euclidean distance. Assume we are given an infeasible split vector $v^0 = (2, 5, 7)$ (see Fig. 4c). Applying Algorithm 1 on v^0 permits to detect that $5 \notin \mathcal{C}(2) = \{3, 4\}$. Hence, k is set to 1 and a fixing position is searched for v^0_{k+1}. The fixing region is $\mathcal{R}(v^0_1, v^0_3) = \mathcal{C}(2) \cap \mathcal{C}^{-1}(7) = \{3, 4\} \cap \{4, 5, 6\} = \{4\}$. Thus, v_2 is set to 4. Algorithm 1 terminates as the obtained split vector $v^1 = (2, 4, 7)$ is feasible. The value of v^1 is $\delta_2 + \delta_4 + \delta_7 = 5.89$. Algorithm 2 is applied on v^1 to find a better split vector. Starting from $k=1$, no improvement is possible at $i=1$ but more choices are available for $i=2$. Indeed, $\mathcal{R}(v^1_i, v^1_{i+2}) = \mathcal{R}(v^1_2, v^1_1) = \mathcal{C}(4) \cap \mathcal{C}^{-1}(2) = \{5, 6, 7, 1\} \cap \{6, 7, 1\} = \{6, 7, 1\}$. As mentioned in Section 2.1, the indexing is circular, therefore, the element v^1_{i+2} is v^1_1 for $i=2$. The split position l which minimizes δ_j for $j \in \mathcal{R}(4, 2)$ is 6. As $\delta_6 < \delta_7$, an improvement is possible and $v^1_{i+1} = v^1_3$ is set to 6. The new feasible split vector $v^2 = (2, 4, 6)$ has a value of 3.41. Algorithm 2 starts a new loop from $k=3$. An improvement is possible for $i=1$. Indeed, in the region $\mathcal{R}(v^2_1, v^2_3)$ which is given by $\{3, 4\}$, $\delta_3 < \delta_4 = \delta_{v_{i+1}}$. Hence, v^2_2 is set to 3. The split vector becomes $v^* = (2, 3, 6)$ with a value of 2.99 which is optimal in this case. The routing corresponding to the split of T according to v^* is represented in the lower-right part of Fig. 4. The cost of this routing is given by $L(T) + \sum_{i=1}^{m} \delta_{v^*_i} = 10.65 + 2.99 = 13.64$.

3.4. The subgradient method

Algorithm 3 is a subgradient method designed to solve *Split*(T). It requires the desired number of routes m, an integer p and the *Split*(T) parameters. It returns a split vector v and a boolean *feas* which equals true if v is feasible and false if *Split*(T) is proven infeasible for the fleet size m. During the initialisation (line 1), the values $\overline{z}$ and $\underline{z}$ corresponding to the values of the best feasible split vector and the best lower bound for z^* are set to $+\infty$ and $-\infty$ respectively. The boolean *feas*, a counter t and a scalar π are set to true, 0 and 2 respectively. The Lagrangian multiplier λ^t corresponding to the t^{th} iteration of the method is initialized to a value λ^0. At each iteration of the repeat-loop, $LB(\lambda^t)$ is solved based on λ^t and the lower bound $\underline{z}$ is updated (lines 4–5). If $\underline{z}$ has not increased after p iterations then π is halved (line 6). If the optimal split vector v^t is infeasible, a reparation is attempted (line 7). If v^t is unrepairable then *Split*(T) is infeasible and the algorithm stops (line 13). Otherwise (line 8), a local search is applied on v^t, the upper bound is updated and the next Lagrangian multiplier vector is calculated using the commonly used update formula

$$\lambda_i^{t+1} := \max\left\{\lambda_i^t + \pi \frac{\overline{z} - z_{LB}(\lambda^t)}{\|s(\lambda^t)\|^2} s_i(\lambda^t), 0\right\}, \quad \forall i \in [n], t \geq 0$$

where $s(\lambda^t) \in \mathbb{R}^n$ is the subgradient vector associated with λ^t defined by:

$$s_i(\lambda^t) = y_i^t - \sum_{j \in \mathcal{C}(i)} y_j^t, \quad \forall i \in [n]$$

$\|s(\lambda^t)\|$ is the Euclidean norm of $s(\lambda^t)$. The algorithm terminates either when π is sufficiently small in which case the best feasible split vector v^* is returned, or, when the split problem is proven infeasible.

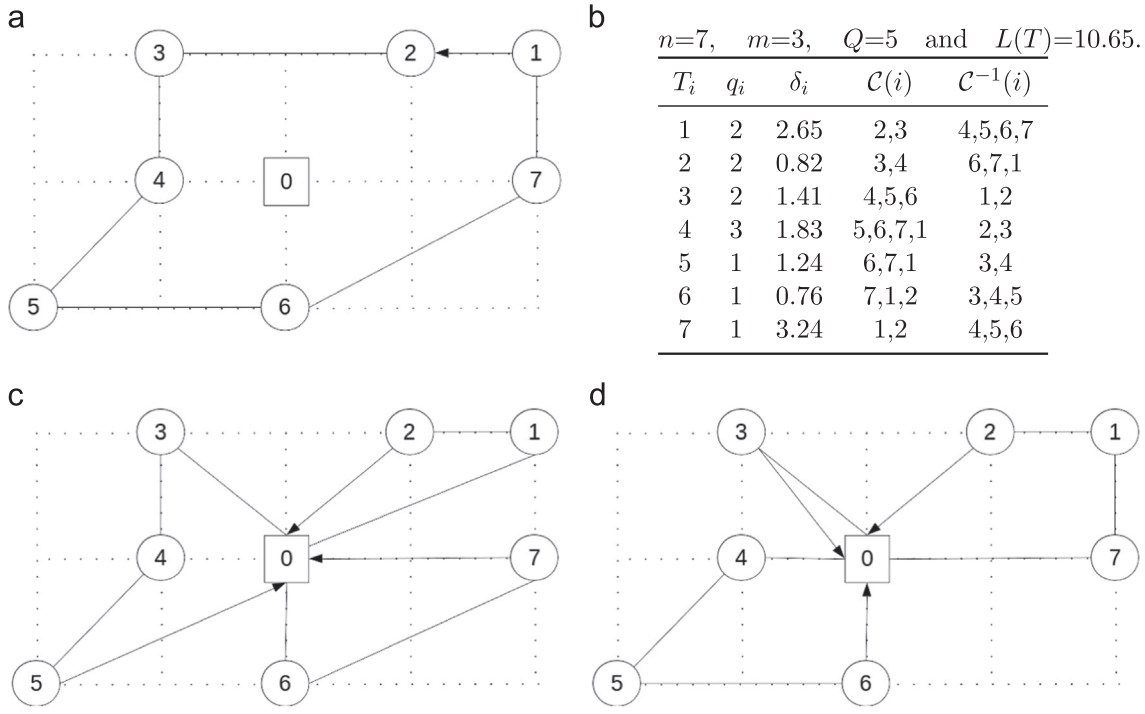


Fig. 4. Illustration of the Lagrangian heuristic on an infeasible split for a giant tour of 7 customers. (a) The giant tour. (b) Instance description and the corresponding *Split*(*T*) parameters. (c) The infeasible split. (d) The repaired and improved split.

Algorithm 3. Lagrangian split (Ls).

Input : integers m, p; *Split*(T) parameters.
Output: split vector v and a boolean *feas*.

1 $\overline{z} := +\infty$, $\underline{z} := -\infty$, *feas* := **true**, $t := 0$, $\pi := 2$, $\varepsilon := 0.01$ and initialize λ^0;
2 **repeat**
3 Calculate Δ_i for all $i \in [n]$;
4 Solve $LB(\lambda^t)$ with m vehicles. Let v^t be the optimal split vector;
5 **if** $z_{LB}(\lambda^t) > \underline{z}$ **then** $\underline{z} := z_{LB}(\lambda^t)$;
6 **if** $\underline{z}$ *has not increased after p iterations* **then** $\pi := \frac{\pi}{2}$;
7 **if** v^t *is not feasible* **then** repair v^t using Algorithm 1;
8 **if** v^t *is feasible* **then**
9 Improve v^t using Algorithm 2;
10 **if** $c(v^t) < \overline{z}$ **then** $v^* := v^t$ and $\overline{z} := c(v^*)$;
11 Calculate λ^{t+1};
12 **else**
13 *feas* := **false**;
14 **end if**
15 $t := t+1$;
16 **until** $\pi \le \varepsilon$ **or not** *feas*;
17 **return** v and *feas*

For every iteration of Algorithm 3, the two arrays Δ and s need to be recalculated (see lines 3 and 11). The following proposition provides two recurrence relations that speed-up this task.

Proposition 4. *The following assertions hold*:

$$(i)\ \begin{cases} s_1 = y_1 - \sum_{j \in \mathcal{C}(1)} y_j \\ s_{i+1} = s_i - y_i + 2\,y_{i+1} - \overset{\overline{\eta}(i+1)}{\underset{j=\overline{\eta}(i)+1}{(n)}} y_j, \quad \forall i \in [n-1] \end{cases}$$

$$(ii)\ \begin{cases} \Delta_n = \delta_n + \lambda_n - \sum_{j \in \mathcal{C}^{-1}(n)} \lambda_j \\ \Delta_i = \Delta_{i+1} - \delta_{i+1} - \lambda_{i+1} + \delta_i + 2\,\lambda_i - \overset{\eta(i+1)-1}{\underset{j=\underline{\eta}(i)}{(n)}} \lambda_j, \quad \forall i \in [n-1] \end{cases}$$

Proof.

$$(i)\ s_i - y_i + 2\,y_{i+1} - \overset{\overline{\eta}(i+1)}{\underset{j=\overline{\eta}(i)+1}{(n)}} y_j$$

$$= y_i - \sum_{j \in \mathcal{C}(i)} y_j - y_i + 2\,y_{i+1} - \overset{\overline{\eta}(i+1)}{\underset{j=\overline{\eta}(i)+1}{(n)}} y_j \quad \text{(by the definition of } s_i\text{)}$$

$$= 2y_{i+1} - \left(\overset{\overline{\eta}(i)}{\underset{j=i+1}{(n)}} y_j + \overset{\overline{\eta}(i+1)}{\underset{j=\overline{\eta}(i)+1}{(n)}} y_j \right)$$

(by the definition of $\mathcal{C}(i)$ and $\overline{\eta}(i)$)

$$= y_{i+1} - \overset{\overline{\eta}(i+1)}{\underset{j=i+2}{(n)}} y_j = y_{i+1} - \sum_{j \in \mathcal{C}(i+1)} y_j = s_{i+1}$$

$$(ii)\ \Delta_{i+1} - \delta_{i+1} - \lambda_{i+1} + \delta_i + 2\,\lambda_i - \overset{\eta(i+1)-1}{\underset{j=\underline{\eta}(i)}{(n)}} \lambda_j$$

$$= \delta_i + 2\,\lambda_i - \sum_{j \in \mathcal{C}^{-1}(i+1)} \lambda_j - \overset{\eta(i+1)-1}{\underset{j=\underline{\eta}(i)}{(n)}} \lambda_j$$

(by the definition of Δ_i)

$$= \delta_i + 2\,\lambda_i - \left(\overset{\eta(i+1)-1}{\underset{j=\underline{\eta}(i)}{(n)}} \lambda_j + \overset{i}{\underset{j=\underline{\eta}(i+1)}{(n)}} \lambda_j \right)$$

(by the definition of $\mathcal{C}^{-1}(i)$ and $\underline{\eta}(i)$)

$$= \delta_i + 2\,\lambda_i - \overset{i}{\underset{j=\underline{\eta}(i)}{(n)}} \lambda_j$$

$$= \delta_i + \lambda_i - \overset{i-1}{\underset{j=\underline{\eta}(i)}{(n)}} \lambda_j = \delta_i + \lambda_i - \sum_{j \in \mathcal{C}^{-1}(i)} \lambda_j = \Delta_i \quad \square$$

Furthermore, we introduce a relaxed version of the Lagrangian split (see Algorithm 4) which allows the variation of m between a lower bound $m_{\min}$ and n. Algorithm 4 starts by setting m to $m_{\min}$ then attempts to split the giant tour T. If the partitioning is impossible for this fleet size, m is incremented. The algorithm terminates whenever a split is found, the extreme case being when a single vehicle is used for every customer ($m=n$).

Algorithm 4. Relaxed Lagrangian split (rxLs).

Input: giant tour T; integer p.
Output: split vector v.
1 Calculate the *Split*(T) parameters: δ_i, $\mathcal{C}(i)$ and $\mathcal{C}^{-1}(i)$ for all $i \in [n]$;
2 $feas$:=**false**;
3 m:=$m_{\min}$;
4 **while not** $feas$ **and** $m \leq n$ **do**
5 $v, feas$:=Ls$(m, p, \delta, \mathcal{C}, \mathcal{C}^{-1})$;
6 m:=$m+1$;
7 **end while**
8 **return** v

4. Integration of Ls into a VNS.

As a single split may provide poor results, split procedures are usually embedded into metaheuristics where several giant tours are partitioned during the search (see Prins [8] and Vidal et al. [19]). In this section, we present the main components of our integration of Ls into a variable neighbourhood search (VNS).

VNS is a powerful metaheuristic proposed by Mladenović and Hansen [20]. It consists in exploring gradually the neighbourhoods of an incumbent solution while applying a local search at each step. If an improving solution is found, it is assigned to the incumbent solution and the search restarts. The process terminates when all the neighbourhoods have been explored. A comprehensive and informative review of the VNS methods and their applications can be found in Hansen et al. [21].

4.1. Capacitated Randomised Nearest Neighbour (CRNN)

Initially, giant tours are built using a capacitated randomised version of the nearest neighbour algorithm for the TSP (see Algorithm 5). This version is used to favour the production of giant tours with an easily splittable structure. Initially, a customer u is chosen and a variable C which represents a cumulated demand is set to $q(u)$. The following operations are then repeated while a customer remains unassigned to T:

Build a restricted candidate list (RCL) of maximum size rn which contains the nearest unrouted customers to u such that their demand added to C does not exceed the capacity Q. If RCL is not empty then a customer v^* is chosen randomly from RCL, it is added at the end of T and u set to v^*. Otherwise C is reset to 0 and the process restarts.

In Algorithm 5, `nearestNeighbor`(u, RCL) is a function which returns the nearest unrouted neighbour of u not present in RCL. The function `add`(RCL, v) inserts the customer v at the back of RCL while `remove`(RCL, i) removes the i-th element of RCL. `rand`(a,b) returns a pseudo-randomly generated number in $\{a, \ldots, b\}$ with $a \leq b$.

Algorithm 5. Capacitated Randomised Nearest Neighbour (CRNN).

Input: integer rn.
Output: giant tour T.
1 Build an empty giant tour T;
2 cnt:=1, u:=`rand`(1, n) and T_{cnt}:=u;
3 S:=$V'\setminus\{u\}$, C:=$q(u)$ and cnt:=$cnt+1$;
4 **while** $cnt \leq n$ **do**
5 RCL :=$\emptyset$, sl:=`min`($rn, n-cnt$);
6 **for** k:=1 **to** sl **do**
7 v:=`nearestNeighbor`(u, RCL);
8 `add`(RCL, v);
9 **end for**
10 i:=1;
11 **while** $i < |RCL|$ **do**
12 **if** $q(RCL(i))+C > Q$ **then**
13 `remove`(RCL, i), i:=1;
14 **else**
15 i:=$i+1$;
16 **end if**
17 **endwhile**
18 **if** $RCL \neq \emptyset$ **then**
19 v^*:=RCL(rand(0, |RCL|));
20 T_{cnt}:=v^*, cnt := $cnt+1$, u:=v^* and C:=$C+q(v^*)$;
21 **else**
22 C:=0;
23 **end if**
24 **end while**
25 **return** T

4.2. Shaking procedure

Our shaking procedure, prototyped `shake`(T, k, r, TL), aims to perturb a giant tour T locally. It consists in choosing randomly a position j then swapping k pairs of customers in the subsequence $(T_{j-r}, \ldots, T_j, \ldots, T_{j+r})$ where r is an integer which represents the radius in which the pairs are swapped around the position j. Also, the procedure ensures that the returned giant tour does not belong to a list TL of giant tours. This list is a long term memory which contains the giant tours that have already been considered for partitioning during the search. We use it as a tabu mechanism to avoid re-examining the same giant tour unnecessarily.

4.3. Concatenation procedure

The concatenation procedure, prototyped `concat`(x), consists in disconnecting the depot from the routes of a routing x then reconnecting the resulting sequences in a nearest-neighbour fashion. The terminal endpoint u of a sequence is connected to the nearest endpoint u' of another sequence. If u' is a terminal endpoint then its sequence is reversed and concatenated to the sequence of u. This is to favour the appearing of new, and possibly better, split solutions. Indeed, reconcatenating a routing without taking into consideration the distances may produce undesirable links between the sequences that would lead an eventual split procedure to produce a nearly similar routing.

4.4. Split search

In order to fully exploit the potential of a giant tour T, we present the intensification mechanism called *split search* (see Algorithm 6).

This alternates between the giant tours' space and the routings' space using the relaxed Lagrangian split (see Algorithm 4), the concatenation procedure (see Section 4.3), a variable neighbourhood descent (VND) and a *gradual* large neighbourhood search (LNS) described below. Note that similar searches were proposed in the literature (see Fig. 2 in Prins et al. [4]), however, to the best of our knowledge, no one used LNS in an attempt to escape from local optima.

The VND permits to intensify the search by exploring the descent neighbourhoods of a routing solution in a specified order restarting the search every time an improvement is made. The resulting solution is thus a local optimum with respect to all the descent neighbourhoods (see Hansen et al. [21]). Our VND explores successively the following descent neighbourhoods: 2-opt intra-route (see Croes [22]), cross-exchange (see Savelsbergh and Goetschalckx [23]), (1,0) node-relocation inter-route, (1,1) node-exchange inter-route and (1,0) node-relocation intra-route (see Van Breedam [24]).

Given a routing x, the gradual LNS, prototyped $\texttt{gLNS}(x, l, \rho)$, consists in partitioning the set V' into l levels $V_1, \ldots, V_l$ such that the customers in the level V_k are closer to the depot than those in the level V_{k+1}, $k = 1, \ldots, l-1$. Starting from $k=1$, $\rho|V_k|$ customers are removed from each level V_k of the routing x with ρ being a parameter in $]0, 1[$. The customers of the level V_k are then reinserted to x using a cheapest insertion procedure before moving to the level V_{k+1}. The search stops when all the levels have been considered for removal and reinsertion.

Initially, Algorithm 6 assigns to x the routing obtained by splitting T using rxLs (see Algorithm 4) and the function $\texttt{toSol}(T, v)$ which returns the routing obtained by splitting the giant tour T according to the split vector v. The routing x is improved using VND, the giant tour T is added to the list TL (see Section 4.2) and T receives the concatenation of x using $\texttt{concat}(x)$ (see Section 4.3). The following operations are then performed while there is an improvement: Split a copy T' of T using rxLs, apply a VND on the obtained routing x' and add T' to TL. If x' improves x, the routine $\texttt{update}(x, x', T)$ assigns x' to x and the concatenation of x to T, then it returns true and the search restarts with the new giant tour T. Otherwise, $\texttt{update}(x, x', T)$ returns false and a gradual LNS is applied on x' followed by another VND in an attempt to unblock the search.

Algorithm 6. Split search (splitSearch).

Input: giant tour T; integers $k'_{\max}$, p, l; real ρ; list of giant tours TL.
Output: solution x.
1 x:=$\texttt{toSol}(T, \texttt{rxLs}(T, p))$;
2 x:=$\texttt{VND}(x, k'_{\max})$;
3 TL:=TL $\cup$ $\{T\}$;
4 T:=$\texttt{concat}(x)$;
5 **repeat**
6 T':=T;
7 x':=$\texttt{toSol}(T', \texttt{rxLs}(T', p))$;
8 x':=$\texttt{VND}(x', k'_{\max})$;
9 TL :=TL $\cup$ T';
10 **if not** $\texttt{update}(x, x', T)$ **then**
11 x':=$\texttt{gLNS}(x', l, \rho)$;
12 x':=$\texttt{VND}(x', k'_{\max})$;
13 $\texttt{update}(x, x', T)$;
14 **end if**
15 **until** *no improvement*
16 TL:=TL $\cup$ T;
17 **return** x

4.5. Reduction tests

Salhi and Sari [25] proposed a mechanism called *reduction tests* which reduces the size of a routing problem instance by eliminating edges which would be unlikely present in the optimal routing. It consists in defining a boolean $pos(i,j)$ which equals true if inserting the node j next to the node i is allowed in the search, false otherwise, i and j being two elements of V. In our method, this boolean is calculated initially as $pos(i,j) =$ true if $c(i,j) < \beta\overline{c}(i)$, false otherwise, where i and j are two customers, β is a positive number and $\overline{c}(i)$ is the average cost of travelling from customer i to other customers in the graph. $pos(i,j)$ is set to false if j equals i and set to true if i or j is the depot.

4.6. Starter

In order to obtain a good initial routing quickly, we use a starter algorithm (see Algorithm 7) which generates a giant tour using the CRNN algorithm (see Section 4.1) then examines the obtained giant tour using the split search (see Section 4.4). The split search used does not include yet the tabu list TL (see line 4). This process is performed ns times and the best routing obtained is returned.

Algorithm 7. Starter (starter).

Input: integers $k'_{\max}$, p, l, ns, rn; real ρ.
Output: solution x_{best}.
1 $c(x)$:=M;
2 **for** i:=1 **to** ns **do**
3 T:=$\texttt{CRNN}(rn)$;
4 x:=$\texttt{splitSearch}(T, k'_{\max}, p, l, \rho)$;
5 **if** $c(x) < c(x_{best})$ **then** x_{best}:=x
3 **end for**
7 **return** x_{best}

4.7. Main algorithm (Ls × VNS)

The main algorithm consists of two main parts (see Algorithm 8). The first one initialises the tabu list TL to the empty set (see Section 4.2), performs the reduction tests (see Section 4.5) using a function $\texttt{reduction}(\beta)$ then attempts to build quickly a good initial routing using the starter algorithm (see Section 4.6). The second part consists of an inner-loop, which is a classical VNS, controlled by an outer loop. The outer-loop starts by concatenating the best routing x_{best}, initially found in the first part of the algorithm, into a giant tour T and setting k to 1. The inner loop copies T into a giant tour T' which is perturbed using the shaking procedure (see Section 4.2). T' is then added to TL before being examined using the split search (see Section 4.4). If the obtained routing improves the current best one, T is set to T', the best routing is updated and k is reset to 1. Otherwise, k is incremented. The inner-loop terminates when k exceeds $k_{\max}$. The stopping criterion for the outer-loop is reached whenever ni non-improving iterations passed or when a time limit $t_{\max}$ has been exceeded.

Table 1
Results for various combinations of Lagrangian split parameters.

Tuning			Deviation (%)				CPU (secs)				# of contacts
#	λ_i^0	p	Min.	Avg.	Max.	Med.	Min.	Avg.	Max.	Med.	
1	0	30	0.00	0.32	8.31	0.00	0.02	0.10	0.40	0.08	79
2	$10 \times \max_{j \in [n]} \delta_j$	20	0.00	0.03	0.85	0.00	0.01	0.10	0.71	0.07	90
3	$\lvert C(i) \rvert \max_{j \in C(i)} \delta_j$	15	0.00	0.03	1.23	0.00	0.01	0.09	1.24	0.06	91
4	$\lvert C(i) \rvert \max_{j \in [n]} \delta_j$	15	0.00	0.02	0.53	0.00	0.00	0.10	3.32	0.05	92

Algorithm 8. Integration of Ls into VNS (Ls × VNS).

Input: integers k_{max}, k'_{max}, p, r, l, ns, rn, ni, t_{max}; reals β, ρ
Output solution x_{best}.
TL:=∅;
`reduction` (β);
x_{best}:=`starter`($k'_{max}, p, l, ns, rn, \rho$);
repeat
  T:=`concat`(x_{best});
  k:=1;
  repeat
    T':=T;
    T':=`shake`(T', k, r, TL);
    TL:=$TL \cup T'$;
    x:=`splitSearch`($T', k'_{max}, p, l, \rho, TL$);
    if $c(x) < c(x_{best})$ **then**
      T:=T';
      x_{best}:=x;
      k:=1;
    else
      k:=$k+1$;
    end if
  until $k > k_{max}$;
until *stopping criterion*;
return x_{best}

In Ls × VNS (see Algorithm 8), if k_{max} is set to 1 then the algorithm becomes an iterated local search (see Algorithm 2 in Prins [26]). Also, Ls × VNS resembles an evolutionary local search (ELS) but it is slightly different as our shaking procedure depends on the counter k which is not the case in ELS (see Algorithm 3 in Prins [26]).

5. Computational experiments

The experiments were conducted on an Intel Core i3 with 2.20 GHz × 4 speed and 3.9 GB RAM running Linux. The algorithms are coded in C++. Two sets of experiments are performed. The first set assesses the ability of the Lagrangian split (see Algorithm 3) in producing good split solutions quickly (see Section 5.1), whereas, the second assesses the performance of our integration of Ls into a VNS in comparison with state-of-the-art methods (see Section 5.2).

5.1. Results for the Lagrangian split

The Lagrangian split (Ls) is a key algorithm in our approach for solving the CVRP. It is called several times and impacts highly on the final routing solution. In this set of experiments, we assess the performance of Ls both in terms of split quality and CPU time. To this end, we propose to conduct the experiments as follows: Given a CVRP instance for which we know a best routing x_{best}, a giant tour T is built by concatenating x_{best} using the `concat`(x_{best}) procedure (see Section 4.3). Using Ls, the giant tour T is then split into the same number of routes present in x_{best}. The cost of the obtained routing is then compared with $c(x_{best})$ and the computing time is recorded.

The dataset proposed recently by Uchoa et al. [27] consists of 100 instances ranging from 100 to 1000 customers generated by varying the following four attributes: the depot positioning, the customer positioning, the demand distribution and the average route size. Furthermore, the best solutions obtained by Uchoa et al. [27] are also available in [28] making this dataset easily accessible.

After preliminary tests, we observed that the two parameters λ^0 and p were the most impacting on the performance of Ls. Therefore, we decided to test several combinations using these two parameters. Table 1 summarizes the results for the most representative tunings each corresponding to a version of Ls which will be denoted by Ls-k for $k = 1, \ldots, 4$. The first three columns correspond to the tuning number, the initial value of λ_i^0 and the value of p respectively. Columns 4, 5, 6 and 7 are the minimum, the average, the maximum and the median deviations of the routing value corresponding to the split produced by Ls in comparison with the best routing of Uchoa et al. [27]. Columns 8, 9, 10 and 11 provide the same statistics for the CPU times. A *contact* means that Ls could produce a routing with a value equal to the value of the best routing found by Uchoa et al. [27]. Column 10 indicates the number of contacts obtained out of 100.

Results for Ls-1 are presented to demonstrate the sensitivity of Ls towards the choice of λ^0 and p. In terms of split quality, the average deviation is found to be always less than 0.32% and the median deviation null for all settings. The best tuning corresponds to Ls-4 with an average and maximum deviations of 0.02% and 0.53% respectively, and 92 contacts made out of 100. Regarding the CPU times, on average, all the settings can split a giant tour within 0.10 secs except in some special cases for Ls-4.

Table 2 presents the characteristics of the instances for which Ls-4 provides the worst CPU times. Columns 1 and 2 are the instance name and the number of routes present in the best routing obtained by Uchoa et al. [27]. Column 3 is the percentage deviation of the routing obtained using Ls with m vehicles and Column 4 is the CPU time spent for that partitioning. The remaining columns are the instance characteristics as described in [27]. The considered dataset does not contain any instance with the same characteristics as X-n573-k30. Thus, we can not conclude that Ls performs slowly on that type of instances. Furthermore, this behaviour may be sporadic given the gap between the largest and the second largest CPU times. Regarding the split quality, a relatively slow splitting does not imply a poor splitting given the null deviation obtained on X-n573-k30.

We observe from these experiments that, on average, Ls is able to provide quickly good split solutions in a wide range of situations. Furthermore, the fact that Ls provides sometimes a suboptimal split solution allows a soft diversification of the search when Ls is integrated into an intensification mechanism. Indeed, we observed

Table 2
Characteristics of the three instances for which Ls-4 provides the worst CPU times (see [27]).

Instance*	m	Dev (%)	Ls CPU (s)	Depot positioning**	Customers positioning[a]	Demands[b]	Q	Max. route size
X-n573-k30	30	0.00	3.32	E	C	SL	210	19.1
X-n957-k87	87	0.00	0.27	R	RC	U	11	11.0
X-n936-k151	159	0.31	0.24	C	R	SL	138	6.2

* X-nA-kB: Instance with A nodes (including the depot) and a minimum of B vehicles.
** R: Random, C: Central, E: Eccentric.
[a] C: Clustered, R: Random, RC: Random-Clustered.
[b] SL (Many small values, few large values): Most demands are generated uniformly in [1,10], the remaining from [50,100], U: Unitary.

Table 3
Results of Ls × VNS on the Christofides et al. (1979) and Golden et al. (1998) datasets.

Instance	$(n, m_{\min})$	BKS	EAMA	CPG	HGSADC	CPMs	Ls × VNS	
							Value	CPU (min)
CMT1	(50,5)	524.61	**524.61**	**524.61**	**524.61**	–	**524.61**	0.55
CMT2	(75,10)	835.26	**835.26**	**835.26**	**835.26**	–	835.77	1.81
CMT3	(100,8)	826.14	**826.14**	**826.14**	**826.14**	–	827.39	1.74
CMT4	(150,12)	1028.42	**1028.42**	**1028.42**	**1028.42**	–	1031.96	6.10
CMT5	(199,16)	1291.29	**1291.29**	1291.45	1291.74	–	1303.79	14.88
CMT11	(120,7)	1042.11	**1042.11**	**1042.11**	**1042.11**	–	1042.12	3.51
CMT12	(100,10)	819.56	**819.56**	**819.56**	**819.56**	–	**819.56**	1.52
G9	(255,14)	579.71	580.60	**579.71**	**579.71**	**579.71**	586.79	23.05
G10	(323,16)	735.66	738.92	737.28	736.26	**735.66**	749.92	33.67
G11	(399,17)	912.03	917.17	913.35	912.84	**912.03**	924.22	74.04
G12	(483,19)	1101.50	1108.48	1102.76	1102.69	**1101.50**	1119.39	170.64
G13	(252,26)	857.19	**857.19**	**857.19**	**857.19**	**857.19**	861.27	25.95
G14	(320,29)	1080.55	**1080.55**	**1080.55**	**1080.55**	**1080.55**	1093.41	37.40
G15	(396,33)	1337.87	1340.24	1338.19	1337.92	**1337.87**	1357.25	67.36
G16	(480,36)	1611.56	1620.56	1613.66	1612.50	**1611.56**	1636.82	159.37
G17	(240,22)	707.76	**707.76**	**707.76**	**707.76**	**707.76**	708.53	23.00
G18	(300,27)	995.13	995.39	**995.13**	**995.13**	997.58	1012.79	48.32
G19	(360,33)	1365.60	1366.14	**1365.60**	**1365.60**	**1365.60**	1376.12	80.34
G20	(420,38)	1817.59	1820.54	1818.32	1818.25	1817.89	1840.45	137.99
Avg. dev (CMT)			0.00	0.00	0.00	–	0.22	
Avg. dev (G)			0.23	0.06	0.03	0.02	1.23	
Avg. dev (overall)			0.15	0.04	0.02	–	0.85	
Computing resource			Xeon 3.2 GHz	Cluster	AMD Opt. 250 2.4 GHz	Cluster	Core i3 2.20 GHz ×4	
Runs per instance			10	5	10	10	10	
Avg. time*			26.86	–	59.59	32.62	47.96	

* Time for HGSADC is scaled to a Pentium IV 3.0 GHz as reported by Vidal et al. [19]. Computing times were not reported for CPG by Groër et al. [32] but a time limit of 5 min was fixed. Time for CPMs is calculated for the G set only.

during the experiments that small deteriorations of the objective induced by Ls during the split search (see Algorithm 6) followed by the VND help the search to escape from local optima.

5.2. Results for the Ls × VNS

In this section, we assess the performance of Ls × VNS in comparison with state-of-the-art CVRP solution methods.

5.2.1. Test datasets

We consider two traditional CVRP datasets: the one of Christofides et al. [29] and the one of Golden et al. [30] referred as CMT and G datasets respectively. Experiments are conducted on instances which do not have distance restrictions. Part of the considered CMT instances have their customers positioned randomly on a grid (CMT1,…,CMT5) whereas, in the other part (CMT11 and CMT12), the customers are clustered. Customers in the G dataset are positioned following geometrical patterns. The sizes range from 50 to 199 customers and from 240 to 483 for the considered CMT and G instances respectively. Our method is compared with the four best existing heuristic methods we are aware of: the edge assembly based memetic algorithm of Nagata and Bräysy [31], the hybrid genetic search with adaptive diversity control of Vidal et al. [19] and the two cooperative parallel algorithms of Groër et al. [32] and Jin et al. [33] which are referred as EAMA, HGSADC, CPG and CPMs respectively. Euclidean distances are calculated following the formula presented in Cordeau et al. [34] that is $c_{ij} = 10^{-d}\lfloor 10^d c_{ij} + 0.5 \rfloor$ with d equal to 7. The cost of the returned solutions are then recalculated with double-precision before being reported.

5.2.2. Parameter calibration

Preliminary experiments were conducted on the instances CMT5, G12 and G20 for calibration. Those instances were selected mainly because of their representativity and their relative difficulty. After calibration, we selected the following parameter setting:

$$(k_{\max}, k'_{\max}, p, r, l, ns, rn, ni, t_{\max}, \beta, \rho) = (5, 5, 20, 20, 8, 5, 2, 300, 10\,800, 1.25, 0.3)$$

with λ_i^0 being set to $|\mathbb{C}(i)| \max_{j \in \mathbb{C}(i)} \delta_j$ and $t_{\max}$ measured in seconds. This parameter setting appeared to provide the best trade-off between solution quality and computing times.

5.2.3. Computational results

The results are presented in Table 3. The first two columns indicate the instance, the number of customers and the minimum number of vehicles used in the relaxed Lagrangian split (see Algorithm 4). The third column indicates the best known solution values (BKS) and proven optimal ones (underlined values) according to [27] and [28]. Columns 4, 5, 6 and 7 indicate the best results presented for EAMA, CPG, HGSADC and CPMs respectively. CPMs has not been tested on the CMT dataset. The two last columns present the best results obtained after 10 runs of Ls x VNS and the average CPU time per run in minutes respectively. In the bottom part of Table 3, the first three rows indicate the average deviation for the CMT dataset, for the G dataset and for all the instances respectively. The three last rows indicate the computing resource on which the methods have been tested, the number of runs per instance and the computing time. Note that CPG and CPMs have been tested on computer clusters involving several nodes each having a number of processors.

The results indicate that Ls × VNS has an average deviation of 0.22% for the CMT dataset, 1.23 % for the G dataset and 0.85 % overall. It could find only two best known values among seventeen on these two extensively studied datasets. Our machine was not find in Dongarra [35], however, its performance is estimated to 4.55 MFlop/s in another source [36]. Based on this data, Ls × VNS performs relatively longer than the other serial algorithms (EAMA and HGSADC) as we use a computer which is almost 2.89 times faster than the Pentium IV 3.0 GHz (see EC.3 in Vidal et al. [19]). As EAMA, CPG and HGSADC find better results for the CMT dataset than for the G dataset, the best method in terms of average deviations seems to be CPMs even though it was not tested on the CMT dataset. Nonetheless, given its overall average deviation, HGSADC remains a leading method as it is a serial method and it was not designed particularly for the CVRP as the CPMs. Furthermore, although EAMA is the oldest method, it is not dominated as it provides the best result for CMT5, a value which is not obtained by the other considered methods.

Overall, Ls × VNS is able to find relatively good solutions on the CMT and G datasets which have been extensively studied in the literature. In addition, our approach is novel and flexible enough to cater for related routing problems by redefining suitably the compatibility sets (see Section 2.2).

6. Conclusion

In this paper, a Lagrangian relaxation method named Lagrangian split (Ls) is proposed to solve the giant tour partitioning problem. This procedure consists of a repair and a local search algorithms embedded into a subgradient method. The mathematical validity and the complexity of the repair algorithm are also provided. The Ls procedure is then integrated into a variable neighbourhood search (VNS) alternating between the spaces of giant tours and routing solutions, using innovative shaking and concatenating procedures, variable neighbourhood descent, large neighbourhood search and tabu search mechanisms. Experiments were conducted to assess the performance of the Ls procedure alone and its integration into the VNS. According to our results, Ls is able to find quickly good split solutions in a wide range of situations and its integration to VNS provides relatively good solutions to extensively studied instances of the literature. This study offers interesting perspectives which are the extension of the Lagrangian split to tackle additional aspects such as the presence of time windows, distance restrictions and heterogeneous fleets producing a novel and efficient methodology for the resolution of a variety of vehicle routing problems.

Acknowledgements

The authors would like to thank both referees for their constructive comments that improved the presentation as well as the content of the paper.

References

[1] Laporte G. The traveling salesman problem: an overview of exact and approximate algorithms. Eur J Oper Res 1992;59:231–47.
[2] Applegate D, Bixby R, Chvátal V, Cook W. The traveling salesman problem: a computational study Princeton series in applied mathematics. Princeton, New Jersey: Princeton University Press; 2006.
[3] Toth P, Vigo D. Vehicle routing: problems, methods, and applications, 18. Philadelphia, Pennsylvania: SIAM; 2014.
[4] Prins C, Lacomme P, Prodhon C. Order-first split-second methods for vehicle routing problems: a review. Transp Res Part C 2014;40:179–200.
[5] Beasley JE. Route first-cluster second methods for vehicle routing. Omega 1983;11:403–8.
[6] Golden BL, Assad A, Levy L, Gheysens F. The fleet size and mix vehicle routing problem. Comput Oper Res 1984;11:49–66.
[7] Ulusoy G. The fleet size and mix problem for capacitated arc routing. Eur J Oper Res 1985;22:329–37.
[8] Prins C. A simple and effective evolutionary algorithm for the vehicle routing problem. Comput Oper Res 2004;31:1985–2002.
[9] Imran A, Salhi S, Wassan NA. A variable neighborhood-based heuristic for the heterogeneous fleet vehicle routing problem. Eur J Oper Res 2009;197:509–18.
[10] Villegas JG, Prins C, Prodhon C, Medaglia A, Velasco N. A GRASP with evolutionary path relinking for the truck and trailer routing problem. Comput Oper Res 2011;38:1319–34.
[11] Vidal T, Crainic TG, Gendreau M, Prins C. A unified solution framework for multi-attribute vehicle routing problems. Eur J Oper Res 2014;234:658–73.
[12] Bouzid MC, Aït Haddadene H, Salhi S. Splitting a giant tour using integer linear programming. Electron Notes Discrete Math 2015;47:245–52 The third international conference on Variable Neighborhood Search (VNS'14).
[13] Makhorin A. Gnu linear programming kit version 4.52, available at: ⟨http://www.gnu.org/software/glpk/glpk.html⟩.
[14] Fisher ML. The lagrangian relaxation method for solving integer programming problems. Manag Sci 2004;50:1861–71.
[15] Cornuejols G, Fisher ML, Nemhauser GL. Location of bank accounts to optimize float: an analytic study of exact and approximate algorithms. Manag Sci 1977;23:789–810.
[16] Kohl N, Madsen OBG. An optimization algorithm for the vehicle routing problem with time windows based on lagrangian relaxation. Oper Res 1997;45:395–406.
[17] Nezhad AM, Manzour H, Salhi S. Lagrangean relaxation heuristics for the uncapacitated single-source multi-product facility location problem. Int J Prod Econ 2013;145:713–23.
[18] Geoffrion AM. Lagrangian relaxation for integer programming. In: Jünger, et al., editors. 50 years of integer programming 1958–2008. Berlin: Springer; 2010. p. 243–81.
[19] Vidal T, Crainic T, Gendreau M, Lahrichi N, Rei W. A hybrid genetic algorithm for multidepot and periodic vehicle routing problems. Oper Res 2012;60:611–24.
[20] Mladenović N, Hansen P. Variable neighborhood search. Comput Oper Res 1997;24:1097–100.
[21] Hansen P, Mladenović N, Moreno Pérez JA. Variable neighbourhood search: methods and applications. Ann Oper Res 2010;175:367–407.
[22] Croes GA. A method for solving traveling-salesman problems. Oper Res 1958;6:791–812.
[23] Savelsbergh MWP, Goetschalckx M., An efficient approximation algorithm for the fixed routes problem. Research Report. Atlanta: University of Georgia; 1992.
[24] Van Breedam A. Improvement heuristics for the vehicle routing problem based on simulated annealing. Eur J Oper Res 1995;86:480–90.
[25] Salhi S, Sari M. A multi-level composite heuristic for the multi-depot vehicle fleet mix problem. Eur J Oper Res 1997;103:95–112.
[26] Prins C. A GRASP × Evolutionary local search hybrid for the vehicle routing problem. In: Pereira F, Tavares J (Eds.). Bio-inspired algorithms for the vehicle routing problem. Studies in computational intelligence. Vol. 161. Berlin: Springer; 2009. p. 35–53.
[27] Uchoa E, Pecin D, Pessoa A, Poggi M, Subramanian A, Vidal T. New benchmark instances for the capacitated vehicle routing problem, Research Report Engenharia de Produção, Universidade Federal Fluminense; 2014.
[28] CVRPLIB. ⟨http://vrp.galgos.inf.puc-rio.br/⟩ [accessed 16.10.2015].
[29] Chrisofides N, Mingozzi A, Toth P. The vehicle routing problem. In: Chrisofides N, Mingozzi A, Toth P, Sandi C, editors. Combinatorial optimization. Chichester: Wiley; 1979. p. 315–38.
[30] Golden BL, Wasil EA, Kelly JP, Chao I-M. The impact of metaheuristics on solving the vehicle routing problem: algorithms, problem sets, and computational results. In: Crainic T, Laporte G, editors. Fleet management and logistics, centre for research on transportation. Berlin: Springer; 1998. p. 33–56.

[31] Nagata Y, Bräysy O. Edge assembly-based memetic algorithm for the capacitated vehicle routing problem. Networks 2009;54:205–15.
[32] Groër C, Golden BL, Wasil EA. A parallel algorithm for the vehicle routing problem. INFORMS J Comput 2011;23:315–30.
[33] Jin J, Crainic T, Løkketangen A. A cooperative parallel metaheuristic for the capacitated vehicle routing problem. Comput Oper Res 2014;44:33–41.
[34] Cordeau J, Gendreau M, Laporte G, Potvin J, Semet F. A guide to vehicle routing heuristics. J Oper Res Soc 2002;53:512–22.
[35] Dongarra JJ. Performance of various computers using standard linear equations software, (Linpack benchmark report), Computer Science Technical Report CS-89-85. University of Tennessee; 2014.
[36] Milkyway: CPU Performance, ⟨http://milkyway.cs.rpi.edu/milkyway/cpu_list.php⟩ [accessed 05.12.2015].