European Journal of Operational Research 269 (2018) 353–362

Contents lists available at ScienceDirect

European Journal of Operational Research

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

Innovative Applications of O.R.

The multi-pickup and delivery problem with time windows

Salma Naccache[a], Jean-François Côté[a,*], Leandro C. Coelho[a,b]

[a] CIRRELT and Université Laval, Québec, Canada
[b] Canada Research Chair in Integrated Logistics, Canada

ARTICLE INFO

Article history:
Received 25 May 2017
Accepted 17 January 2018
Available online 15 March 2018

Keywords:
Vehicle routing problem
Multi-pickup and delivery problem
Sequential ordering problem

ABSTRACT

This paper investigates the multi-pickup and delivery problem with time windows in which a set of vehicles is used to collect and deliver a set of items defined within client requests. A request is composed of several pickups of different items, followed by a single delivery at the client location. We formally describe, model and solve this rich and new problem in the field of pickup and delivery vehicle routing. We solve the problem exactly via branch-and-bound and heuristically developing a hybrid adaptive large neighborhood search with improvement operations. Several new removal and insertion operators are developed to tackle the special precedence constraints, which can be used in other pickup and delivery problems. Computational results are reported on different types of instances to study the performance of the developed algorithms, highlighting the performance of our heuristic compared to the exact method, and assessing its sensibility to different parameter settings.

1. Introduction

In many applications, vehicles must perform several sequential pickups of one or different commodities, and once all pickups are performed, the vehicle must deliver all of them to a given location. This type of problem arises, for example, in the collection of cash from parking tolls: an employee leaves the depot with a key that only allows access to the cash of some tolls to be dropped in a given delivery location. He can then visit the tolls in any order, but must visit them all before delivering all the cash, which must happen before he can have access to another key. This multi-pickup and delivery problem also appears for companies that allow a client to order food from different restaurants; the company must then perform all pickups at different places, before delivering all meals to the client location. Examples of companies operating under this setting are *JUST EAT*, *Uber eats* and *SkipTheDishes*. These applications impose not only a partial ordering of the visits (all pickups prior to the delivery), but also that all stops associated to a single request must be performed by the same vehicle.

In this paper we consider a multi-pickup and delivery problem with time windows (MPDPTW), in which a set of requests is satisfied by a fleet of vehicles. In each request, items are required to be picked up from different locations to be shipped and unloaded at one common delivery location. In addition, a time window (TW) is associated with each node, such that pickups and deliveries can only be performed within the node's start and end times. The depot at which the vehicles are also housed contain TWs representing its opening hours. The goal is to obtain feasible vehicle tours fulfilling the requests for pickups and deliveries, while minimizing the overall costs associated with the routing of a set of requests.

In the MPDPTW, a request must be fulfilled by a single vehicle. This means that all pickups and the corresponding delivery must be performed by a single tour, possibly combined with other requests. Moreover, vehicle tours have to be developed with respect to precedence constraints, while reducing the overall routing cost. The precedence constraints are related to the order in which nodes of a given request are visited. These constraints do not incur a direct precedence between the last visited pickup and delivery nodes. It is rather required for a vehicle fulfilling a given request to visit all its pickup locations before reaching the corresponding delivery node.

The MPDPTW shares some characteristics with problems previously studied in the literature, namely the pickup and delivery problem with time windows (PDPTW) and the sequential ordering problem (SOP). These are briefly reviewed next.

According to the review and classification of Berbeglia, Cordeau, Gribkovskaia, and Laporte (2007), our problem lies in the family of many-to-many pickup and delivery, in which a request is associated with a pickup node and a delivery node. This differs from the one-to-many in which the origin of all commodities is the depot, and the many-to-one in which all deliveries are headed to the depot. However, in classical many-to-many problems, a request consists of a pair origin-destination. In the MPDPTW, many compulsory origins are associated with a single delivery, and all

* Corresponding author.
E-mail address: jean-francois.cote@fsa.ulaval.ca (J.-F. Côté).

https://doi.org/10.1016/j.ejor.2018.01.035
0377-2217/

origins must be visited (in no particular order) prior to the delivery. We note, however, that existing algorithms for pickup and delivery problems cannot solve an instance of the MPDPTW due to the added complexity of handling multiple pickups for a single request.

Approximate algorithms for the pickup and delivery problem include the adaptive large neighborhood search of Pisinger and Ropke (2007), the parallel neighborhood descent of Subramanian, Drummond, Bentes, Ochi, and Farias (2010), and the particle swarm optimization of Ai and Kachitvichyanukul (2009); Goksal, Karaoglan, and Altiparmak (2013). Exact algorithms for the pickup and delivery with TWs include the branch-and-cut of Ropke, Cordeau, and Laporte (2007), the branch-cut-and-price of Ropke and Cordeau (2009) and the set partitioning-based algorithm of Baldacci, Bartolini, and Mingozzi (2011). A number of variants of the problem exists, as it is used to represent many real-life distribution problems (Coelho, Renaud, & Laporte, 2016).

A similar problem is the SOP, which consists of building a Hamiltonian path in order to solve the asymmetric traveling salesman problem (ATSP) with precedence constraints: the visit of a given node has to be done after visiting a required set of direct and/or indirect predecessors. The SOP differs from the pickup and delivery problem as a node can have multiple direct predecessors (Alonso-Ayuso, Detti, Escudero, & Ortuño, 2003; Guerriero & Mancini, 2003). Moreover, a single node can be the direct predecessor of several other nodes, resulting into a tree-like route structure. This problem was introduced in Escudero (1988) to design heuristics for production planning systems. It has been extended into the constrained SOP (CSOP) (Escudero & Sciomachen, 1993) to include additional precedence relationships between nodes, such as TWs, where a release date and a deadline are associated with each visited node. The SOP is used to model real-world problems within production planning in flexible manufacturing systems and for vehicle routing and transportation problems (Ezzat, Abdelbar, & Wunsch, 2014). It has been applied to helicopter routing, job sequencing in flexible manufacturing, stacker cranes in automatic storage systems (Ascheuer, Jünger, & Reinelt, 2000), single vehicle routing with pickup and delivery constraints (Fiala Timlin & Pulleyblank, 1992; Savelsbergh, 1990), multicommodity one-to-one pickup and delivery TSP problems (Gouveia & Ruthmair, 2015; Hernández-Pérez & Salazar-González, 2009) and dial-a-ride problems in which items or people are picked up at some points and delivered to others (Balas, Fischetti, & Pulleyblank, 1995). According to Desaulniers, Desrosiers, Erdmann, Solomon, and Soumis (2001) a variety of techniques based on restrictions, e.g., precedence constraints, are used in order to reduce the network size. Several approaches have been adopted to solve the SOP. Savelsbergh (1990) developed local search algorithms based on the k-exchange concept. Balas et al. (1995) used time separation algorithms for solving problems arising in both scheduling and delivery routing problems. Ascheuer et al. (2000) used an integer program solved by a branch-and-cut. Guerriero and Mancini (2003) proposed a sequential solution approach through a parallel version of the heuristic rollout algorithm, while Seo and Moon (2003) adopted a hybrid genetic algorithm. Alonso-Ayuso et al. (2003) used a lagrangian relaxation-based scheme for obtaining lower bounds on the optimal solution. Finally, Letchford and Salazar-González (2016) provided a multi-commodity flow formulation for the SOP and the CSOP.

While the MPDPTW is associated with many model characteristics of existing distribution problems, to our knowledge, this problem has not received any attention in the literature. In this paper we introduce a formal problem formulation of MPDPTW taking into account its complicating multi-pickup characteristics. Moreover, each vehicle route in a MPDPTW can be interpreted as an ATSP, in which the distance between two nodes is different depending on the sequence in which the nodes are visited. For example given two locations l_1 and l_2, the travel time from l_1 to l_2 may be different from the travel time from l_2 to l_1. Given the precedence constraints on request nodes and on vehicle start and end depots, the vehicle routes are similar to a constrained ATSP. We solve this formulation by branch-and-bound using a commercial solver, capable of proving optimality for small instances and providing bounds for larger ones. Moreover, we exploit the ALNS framework to design and implement an algorithm tailored for the MPDPTW. We propose new removal and insertion operators handling the multi-pickup characteristics of the requests defined within the problem. These operators are also new in the literature and can help solve other types of similar problems, being adapted for the TOP or the ATSP with precedence constraints. Promising solutions are also polished using local search operators within our heuristic framework.

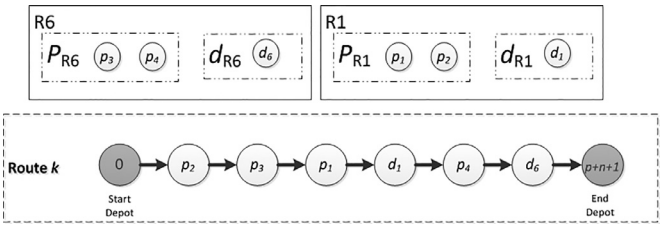

Fig. 1. Requests R1 and R6 inserted into route k.

The remainder of this paper is organized as follows. Section 2 describes MPDPTW and introduces a mathematical programming formulation. The general heuristic framework based on a hybrid ALNS method is presented in Section 3, including the new request insertion procedure. Computational results are reported in Sections 4 and 5 concludes the paper with main findings and future research avenues.

2. Problem description

A problem instance of the MPDPTW contains n requests and m vehicles. Let $P = \{1, \ldots, p\}$ be the set of pickup nodes, and $D = \{p+1, \ldots, p+n\}$ be the set of delivery nodes where $|D| = n$ and $p \geq n$. Let $R = \{r_1, \ldots, r_n\}$ be the set of requests to be routed. Each request $r \in R$ is represented by a set of pickup nodes $P_r \subseteq P$ and a delivery node $d_r \in D$. Each pickup node belongs to exactly one set, and each request always contains at least one pickup node. Let $N = P \cup D$ be the set of customer nodes. Let $r(i)$ be the request associated with node $i \in N$. Let $K = \{1, \ldots, m\}$ be the set of available vehicles.

The graph $G = (V, A)$ consists of the nodes $V = N \cup \{0, p+n+1\}$ where 0 and $p+n+1$ are the starting and ending depot. Each node $i \in V$ has a service time s_i and a TW $[a_i, b_i]$. Given the TW, a vehicle can arrive at node i earlier than the start of its TW a_i, having to wait until a_i to start the service. Moreover, a vehicle must arrive at node i before b_i, such that the service at node i starts within its TW.

The set of arcs is $A = V \times V$ minus arcs that lead to infeasible solutions: we omit arc (i, j) if i is a pickup node and j is its delivery node if $b_j < a_i + s_i + t_{ij}$. A distance $d_{ij} \geq 0$ and a travel time $t_{ij} \geq 0$ are associated with each arc $(i, j) \in A$. Let $A^+(i)$ and $A^-(i)$ be the sets of outgoing and incoming arcs from node $i \in V$.

Fig. 1 illustrates two requests $R1$ and $R6$ and a route associated with vehicle k. For example in $R6$ two pickups nodes p_3 and p_4 have to be visited to collect items to be delivered to d_6. $R1$ and $R6$ are inserted in route k where precedence constraints are respected. Note that node p_1 is not directly visited after p_2 from the same request. Moreover, the delivery node of request R_1 is visited after

all items have been collected from p_1 and p_2. The same applies for R_6.

A solution to the MPDPTW is a set of feasible routes obtained by assigning all requests to the vehicles of the fleet. The problem can be mathematically formulated with the following decision variables:

- x_{ij}^k is 1 if arc (i, j) is traversed by vehicle k, 0 otherwise;
- y_{rk} is 1 if request r is visited by vehicle k, 0 otherwise;
- S_i indicates the beginning of service at node $i \in V$.

Then, the MPDPTW can be formulated as follows:

$$\min \sum_{k\in K} \sum_{(i,j)\in A} c_{ij} x_{ij}^k \tag{1}$$

$$\text{s.t.} \sum_{j\in A^+(i)} x_{ij}^k = y_{r(i)k} \qquad k \in K, i \in N \tag{2}$$

$$\sum_{j\in A^-(i)} x_{ji}^k = y_{r(i)k} \qquad k \in K, i \in N \tag{3}$$

$$\sum_{j\in A^+(0)} x_{0j}^k \le 1 \qquad k \in K \tag{4}$$

$$\sum_{k\in K} y_{rk} = 1 \qquad r \in R \tag{5}$$

$$a_i \le S_i \le b_i \qquad i \in V \tag{6}$$

$$S_j \ge S_i + s_i + t_{ij} - M\left(1 - \sum_{k\in K} x_{ij}^k\right) \qquad (i, j) \in A \tag{7}$$

$$S_{d_r} \ge S_i + s_i + t_{id_r} \qquad i \in P_r, r \in R \tag{8}$$

$$x_{ij}^k \in \{0, 1\} \qquad (i, j) \in A, k \in K \tag{9}$$

$$y_{rk} \in \{0, 1\} \qquad r \in R, k \in K. \tag{10}$$

The objective function (1) minimizes the overall transportation cost. Constraints (2) and (3) are degree constraints. They also ensure that all nodes of a request belong to the same vehicle. Constraints (4) ensure that at most K vehicles are used in the solution. Constraints (5) force a request to be served by only one vehicle. Constraints (6) and (7) guarantee schedule feasibility with respect to TWs. The precedence order is preserved via constraints (8). Constraints (9) and (10) impose the nature and the domain of the variables. M is usually a big enough number and it is equal to $\max\{b_i + s_i + t_{ij} - a_j, 0\}$ for each constraint (7) (Desaulniers, Madsen, & Ropke, 2014).

3. A hybrid ALNS heuristic for the MPDPTW

The ALNS heuristic is an extension of the large neighborhood search (LNS) introduced by Shaw (1998). The ALNS framework presented in Pisinger and Ropke (2007) and Ropke and Pisinger (2006) was applied to five variants of distribution problems, including the PDPTW. The ALNS has been applied to the VRPTW, the CVRP, the multi-depot VRP, the site-dependant VRP, the open VRP and the PDPTW (Pisinger & Ropke, 2007; Ropke & Pisinger, 2006), the two-echelon VRP (2E-VRP) (Grangier, Gendreau, Lehuédé, & Rousseau, 2016; Hemmelmayr, Cordeau, & Crainic, 2012), two-echelon multi-trip VRP with satellite synchronization (Grangier et al., 2016), the location-routing problem, which is modeled as a 2E-VRP (Hemmelmayr et al., 2012), VRPs with stochastic demands and weight-related costs (Luo, Qin, Zhang, & Lim, 2016), VRPs arising in an integrated inventory-transportation synchronized supply chain (Coelho, Cordeau, & Laporte, 2012a; 2012b; Lee, Chan, Langevin, & Lee, 2016), a real-life multi-depot multi-period VRP with a heterogeneous fleet (Mancini, 2016), among others.

In line with the recent literature (Bortfeldt, Hahn, Männel, & Mönch, 2015; Coelho, Cordeau, & Laporte, 2012b; Mancini, 2016; Muller, Spoorendonk, & Pisinger, 2012; Pereira, Coelho, Lorena, & de Souza, 2015), we exploit the ALNS framework and include local search operators to improve the quality of promising solutions, with the aim of improving the quality of the final solution and decreasing computational time. We adapt ALNS for solving the MPDPTW as follows.

An initial solution S is generated through a construction heuristic in which requests are progressively inserted within an available vehicle, at its minimum insertion cost position. After all requests have been inserted, solution S is improved by our local search operators. The main part of the algorithm lies in the iterative removal and insertion of pickup and delivery nodes. An overview of our algorithm is presented Section 3.1, while removal procedures are described in Section 3.2 and insertion procedures in Section 3.3. The local search improvement is presented in Section 3.4.

3.1. Overview of the hybrid ALNS algorithm

Algorithm 1 provides an overview of the framework of our heuristic.

Request removal and insertion operators *ro* and *io* are randomly selected from the sets *RO* and *IO* using independent roulette wheels based on the scores of each operator. The score of an operator is initially set to zero and updated as follows. The score of an operator is incremented by θ_1 if the modified solution is a new best one, by θ_2 if the modified solution is better than the previous one, and by θ_3 if it is not better but still accepted (such that $\theta_1 > \theta_2 > \theta_3$). Let ω_i be a measure of how well operator i has performed in the past; then given h operators with weights ω_i, operator j will be selected with probability $\omega_j / \sum_{i=1}^{h} \omega_i$.

The acceptance criterion is such as that a candidate solution S' is accepted given the current solution S with a probability $e^{-(f(S')-f(S))/T}$, where T is the temperature that decreases at each iteration according to a standard exponential cooling rate. When the temperature reaches a minimum threshold T_{min}, it is set to its initial value, in a procedure called reheating. This procedure allows worse solutions to be more easily accepted, and increases the diversification of the algorithm.

Finally, the algorithm updates the scores and weights of the operators at every φ iteration. Let π_i be the accumulated score of operator i and o_{ij} be the number of times operator i has been used in the last segment j. The updated weights are then

$$\omega_i := \begin{cases} \omega_i & \text{if } o_{ij} = 0 \\ (1-\eta)\omega_i + \eta\pi_i/o_{ij} & \text{if } o_{ij} \neq 0, \end{cases} \tag{11}$$

where $\eta \in [0, 1]$ is called the reaction factor and controls how quickly the weight adjustment reacts to changes in the operator performance. The scores are reset to zero at the end of each segment.

3.2. Removal operators

Removal operators allow a solution to be changed by partially destroying it, removing some requests from the routes of some vehicles. We define a set *RO* of removal operators containing four different heuristics to remove nodes from a solution. The number q of requests to remove is drawn between limits established

Algorithm 1 Hybrid ALNS framework for the MPDPTW.

Require: T_{init}: initial temperature; c: cooling rate
Require: S: initial solution
Require: RO: set of removal operators with score null
Require: IO: set of insertion operators with score null
$S_{best} \leftarrow S$
$T \leftarrow T_{init}$
while stop criterion not met **do**
  $S' \leftarrow S$
  $q \leftarrow$ Generate a random number of requests to remove
  $ro \leftarrow$ Select an operator from RO (Section 3.2) using a roulette wheel based on the weight of the operators
  $io \leftarrow$ Select an operator from IO (Section 3.3) using a roulette wheel based on the weight of the operators
  Remove q requests from S' by applying ro
  Insert removed requests into S' by applying io using a random pickup insertion method (Section 3.3.1)
  if $f(S') < f(S_{best})$ **then**
    Apply *improvement* (Section 3.4) to S'
    $S_{best} \leftarrow S \leftarrow S'$
    Increase the scores of the ro and io by θ_1
  else if $f(S') < f(S)$ **then**
    $S \leftarrow S'$
    Increase the scores of the ro and io by θ_2
  else if $acccept(S', S)$ **then**
    $S \leftarrow S'$
    Increase the scores of the ro and io by θ_3
  end if
  $T \leftarrow T * c$
  if $T < T_{min}$ **then**
    $T \leftarrow T_{init}$
  end if
  if end of a segment of φ iterations **then**
    Update weights and reset scores of operators
    Apply *improvement* (Section 3.4) to S
  end if
end while

Algorithm 2 Randomly remove requests from a solution.

Require: S: a solution
Require: q: number of requests to remove
$L \leftarrow$ set of assigned requests in S
for $i = 1$ to q **do**
  $r \leftarrow$ a randomly chosen request in L
  $L \leftarrow L \setminus \{r\}$
  Unassign all the nodes of request r in solution S
end for

in Section 4, while we set α as the number of requests already assigned to a route.

1. Random removal: here we randomly remove q requests from a solution. The motivation behind this operator is to increase diversity in the search. The operator is described in Algorithm 2. At the beginning, the set L contains all the assigned request in solution S. At each iteration, a request r is randomly selected in the list. All its nodes are removed from S and r is removed from L too.
2. Shaw removal: this operator removes clusters of requests that are related one to each other. A seed request is randomly selected, and then its $q-1$ most related neighbors are removed from the solution. The idea of the Shaw removal is to remove requests that are similar to each other, in the hope that they can be all inserted elsewhere in a more profitable position. The procedure is described in Algorithm 3. First, a function $g(r_1, r_2, S)$ measuring the relatedness between two requests is defined. Several measures were tested, and the distance between the delivery nodes was selected as the best measure of relatedness. Second, an assigned request r is randomly selected from the solution. The nodes of r are removed from the solution and r is stored inside the set of removed requests R. Finally, the removal loop occurs. A request r is randomly selected in R. Assigned requests from S are put inside the array L and sorted according to the function g. A random number between 0 and 1 is chosen. The request located in position y^D is removed from the solution and added to R. D is a parameter set to 6 as in Ropke and Pisinger (2006). The loop is restarted until q requests are removed.
3. Most expensive nodes: here we identify the nodes that lead to the biggest savings when removed from the solution, in the hope that a less expensive insertion position can be found for them later. This operator can be seen as a variant of Shaw removal where the relatedness between two requests $g(r_1, r_2, S)$ is the cost of removing the most expensive arcs connecting a node from request r_1. The final value is multiplied by -1 to have the most expensive request in the first position of the array.
4. Most expensive requests: this operator is similar to the most expensive nodes, but acts on full requests rather than on individual nodes. This operator can be seen as a variant of Shaw removal where the relatedness between two requests $g(r_1, r_2, S)$ is the cost of removing request r_1 from S. The final value is multiplied by -1 to have the most expensive request in the first position of the array.

Algorithm 3 Remove requests that are related from a solution.

Require: S: a solution
Require: q: number of requests to remove
Require: $g(r_1, r_2, S)$: a relatedness function between two requests in solution S
$r \leftarrow$ a randomly chosen request in S
$R \leftarrow \{r\}$: set of removed requests
Unassign all the nodes of request r in solution S
while $|R| < q$ **do**
  $r \leftarrow$ a randomly chosen request in R
  $L \leftarrow$ an array of assigned requests in S
  Sort L such that for $i < j \Rightarrow g(r_i, r, S) < g(r_j, r, S)$
  $y \leftarrow$ a random number between 0 and 1
  Unassign all the nodes of request $L[y^D|L|]$ in solution S
  $R \leftarrow R \cup \{L[y^D|L|]\}$
end while

3.3. Insertion operators

Insertion operators aim at building a feasible solution after some requests have been removed from it. We developed a set IO of insertion operators containing five heuristics. Each of these five insertion operators require evaluating the best positions to insert a node. The general procedure for inserting a request r in a route k is described in Algorithm 4. This general procedure requires subroutines described in Section 3.3.1.

1. Greedy insertion: select a remaining request and inserted it based on a greedy criterion, namely in the position yielding the lowest increase in the objective function.
2. Regret-3, no noise: the selection of the next request to be inserted in the solution is based on a regret criterion. This means that one does not want to have a costly insertion at a later point in time if the current request is not selected now. Here,

Algorithm 4 Insert a request in a vehicle route.

Require: r : Request to insert
Require: k: Current route
1: $pickups \leftarrow P_r$
2: **while** $pickups <> \emptyset$ **do**
3: $p_i \leftarrow selectAPickup(pickups, method)$
4: $BestPosition(p_i, k) \leftarrow$ Insert p_i at its best insertion position in k
5: $pickups \leftarrow pickups \backslash \{p\}$
6: **if** $BestPosition(p_i, k) = null$ **then**
7: **return** Request insertion infeasible
8: **end if**
9: **end while**
10: $BestPosition(d_r, k) \leftarrow$ Insert d_r at its best insertion position in k
11: **if** $BestPosition(p_i, k) = null$ **then**
12: **return** Request insertion infeasible
13: **end if**

we set the regret level to 3, and no noise is applied in its computation.
3. Regret-m, no noise: similar to the previous one, the regret level is equal to the number of vehicles m, and no insertion noise is applied.
4. Regret-3, with noise: here, we make use of an insertion noise to allow some extra diversity in the search of the regret-3 computation.
5. Regret-m, with noise: similarly, we use a regret-m criterion to which insertion noise is added.

If the application of a noise is allowed, a *generateRandomNoise*() function generates a random value in the interval $[-addedNoise, +addedNoise]$, in which *addedNoise* is a value proportional to the maximum value in the problem's distance matrix. The insertion noise value is used in Algorithm 5 and is discussed in details next.

Early computational experiments have empirically shown the advantage of these operators when coupled with the insertion order of the pickup nodes (see Section 3.3.1), whereas more traditional regret-1 and regret-2 operators did not perform well for the problem at hand.

In Algorithm 4, while there are nodes in the *pickups* set, the algorithm selects a pickup node $p_i \in P_r$ using one of the pickup selection methods described in Section 3.3.1. Then, the best insertion position of p_i in k ($BestPosition(p_i, k)$) is found by applying Algorithm 5, and then it is removed from the *pickups* set. Once all the pickups are inserted, the delivery node d_r is inserted at its best insertion position in route k. If the insertion of a node $i \in \{P_r \cup \{d_r\}\}$ is not possible, the algorithm is interrupted and $BestPosition(i, k)$ returns *null*.

In order to determine the best insertion position of node i in route k, Algorithm 5 iterates through all the nodes already inserted in the route starting from the depot node 0 until its last customer location. At each step, the algorithm temporarily inserts i after a node j from k, computing the increase in routing costs, Δ_i^k. As an example, assume that Algorithm 5 is processing the pickup node p_6 from request R_3, given the nodes already inserted in route k. The algorithm would test the insertion of p_6 after 0, p_2, p_3, p_1, d_1, p_4 and d_6.

At the first iteration of the algorithm, node i is inserted between the depot node 0 and the first customer of route k. It then computes the arrival time of k at i, verifying that the vehicle would visit i before b_i given the current insertion. Next, t_{next} is set to the vehicle arrival time at *next* assuming that k does not include i. Then, the vehicle arrival time at *next*, namely t'_{next} is computed, assuming that i is inserted in k. The added duration is used to

Algorithm 5 Insert i at its best position in k.

Require: i : Node to insert
Require: k: Current route
1: $\Delta_i^{k^*} \leftarrow \infty$
2: $BestPosition(i, k) \leftarrow null$
3: $prev \leftarrow$ the depot node 0
4: $next \leftarrow$ first customer of route k
5: **while** $prev <> p + n + 1$ **do**
6: $t \leftarrow vehicleArrivalTimeAt(k, i)$
7: **if** $t > b_i$ **then**
8: Exit Algorithm 5
9: **end if**
10: Set t_{next} the actual arrival time at *next*
11: $t'_{next} \leftarrow$ arrival time at *next* if i is inserted before
12: $addedDuration \leftarrow t'_{next} - t_{next}$
13: **if** $t_{next} > b_{next}$ OR $addedDuration > slack_{next}$ **then**
14: $prev \leftarrow next$
15: $next \leftarrow$ next customer after *next*
16: Go to While
17: **end if**
18: $NewCost \leftarrow C_{prev,i} + C_{i,next} - C_{prev,next} + generateRandomNoise()$
19: **if** $NewCost < \Delta_i^{k^*}$ **then**
20: Insert i after $prev$ in current solution to obtain S'
21: **if** S' is feasible **then**
22: $BestPosition(i, k) \leftarrow prev$
23: $\Delta_i^{k^*} \leftarrow NewCost$
24: **end if**
25: **end if**
26: $prev \leftarrow next$
27: $next \leftarrow$ next customer after *next*
28: **end while**
29: **return** $BestPosition(i, k)$

verify that if node i is visited, the vehicle would visit *next* before the end of its TW b_{next}. Also, the added duration is compared with the slack at *next*, that indicates to which extent the vehicle's visit to *next* would be postponed. If one of these conditions is violated, the algorithm moves to the next iteration, to test the next insertion position.

If all the conditions are verified, the insertion cost of i in k at the position under evaluation (*NewCost*) is computed, possibly adding a random noise. A temporary *NewCost* is compared against $\Delta_i^{k^*}$ to determine whether the insertion of i in k improves the solution. If the condition is verified, the feasibility of the new solution is tested by first inserting i in k to obtain S'. If S' generates a finite cost, the solution is feasible and the best known insertion position of i in k is set to *prev*. Finally, $\Delta_i^{k^*}$ is set to the value of *NewCost* in line 23.

The next insertion position is updated, and after iterating through all customer nodes in k, the algorithm returns the best insertion position of i in k.

3.3.1. Determining the order of pickup nodes

One of the challenges of the MPDPTW is that a single request contains many pickup nodes that can be performed in any order. Hence, when inserting a request on a route, we have the choice to change the order in which pickup nodes are inserted, as long as the delivery node is present after all pickup nodes along the route. Hence, for each of the four insertion operators described in Section 3.3, we must determine the order at which the different nodes will be inserted.

Here we present four methods to determine the insertion order.

1. Simple insertion order: here, the method selects the pickup nodes according to the order in which they are described in the instance.
2. Random insertion order: in the random method, the index of a pickup node from *pickups* is randomly generated in [1..|*pickups*|].
3. Cheapest insertion first: this method inserts first the node with the cheapest insertion cost.
4. Most expensive first: likewise, this method works similarly to the *cheapest* one, but inserts first the most expensive nodes.

3.3.2. Complexity analysis

In this section, we analyze the complexity of inserting a request into a route using the different insertion orders. For this analysis, we suppose the request is composed of k pickup nodes and the route contains n nodes. All our proposed algorithms operate in four steps: the first step selects a pickup node, the second places the selected pickup node in its best position, the third updates the route's arrival times and slacks, and the last step finds the best placement position for the delivery node.

For the first step, the complexity of selecting the pickup is $O(1)$ for the simple and random insertion orders because we simply take the next pickup node in the list. For the cheapest and most expensive insertion orders, we need to go through the whole route to find the best position for each pickup node. Going through the route requires $O(n)$ operations, and this has to be done for each of the remaining k pickup nodes. Thus, the complexity is $O(kn)$ at each first step. Because we repeat this operation $k-1$ times, the total number of operations is $O(k)$ for the simple and random insertion orders and $O(k^2n)$ for the cheapest and most expensive insertion orders.

The second step determines the best insertion position for the selected pickup. For the simple and random insertion orders, we need to evaluate all possible insertion positions in the route. Therefore, the complexity is $O(n)$. For the cheapest and most expensive insertion orders, the best insertion position was already found in the first step. Thus, the complexity is $O(1)$. Because we repeat this step at least $k-1$ times, the overall number of operations is $O(kn)$ and $O(k)$.

The third step requires a forward pass to be done through the route to update the arrival times. Once this step is done, a backward pass is performed to calculate the slack at each node. Thus, this step requires $O(n)$ operations and $O(kn)$ in total.

The last step is to determine the best position for the delivery node which requires $O(n)$ operations.

Overall, including all steps, our *simple* and *random* insertion orders require $O(kn)$ operations, and *cheapest* and *most expensive* insertion orders require $O(k^2n)$ operations.

3.4. Improvement procedure

In order to polish potentially good solutions, we have developed two improvement procedures that we apply sequentially to some solutions. The precise use of this procedure is described in Section 3.1. We now describe each of the two elements of this procedure.

1. Relocate: in the relocate improvement, we select a request and try to insert it into every other route. The accepted move is the one leading the greatest decrease in cost among all routes. If no improvement solution can be found, we test the next request. This is executed for all requests until no improvement can be found.
2. Exchange: here we select two requests from two different routes, and we try to insert them into each other's routes at the smallest cost. If a better solution can be found, the exchange is accepted, otherwise we go back to the previous solution. This is executed for all pairs of requests until no improvement can be found.

These procedures are executed in the order they are presented here and each algorithm is executed while there is an improvement.

4. Computational experiments

In this section we describe our computational experiments. Section 4.1 introduces the characteristics of the MPDPTW instances. The many different parameters of our algorithm and their chosen values are discussed in Section 4.2. The results of detailed and extensive computational experiments are then presented in Section 4.3.

4.1. Test instances

The test instances were built upon existing PDTW instances from (Li & A. Lim, 2001). For each PDTW instance, pickup and delivery nodes are first separated into two distinct lists. To create a request, we first generate a random number k of pickup nodes it will contain. Then, we create an empty request that contains a dummy drop node at the depot. For $k-1$ iterations, we randomly select a pickup node and we add it to the request. Then, we use an insertion operator to insert the request into an empty solution. We use the random insertion order 20 times to insert the request. If it succeeded, the pickup node is removed from the list and we move on to the next iteration. In case of failure the process is restarted with another randomly selected pickup node. Once all $k-1$ pickup nodes are selected, we apply the same process for the drop node. If we cannot find a feasible solution for any of the remaining drop nodes, the whole process is restarted. For an instance with n nodes, the request generation is stopped once our requests contain at least n nodes.

The developed heuristic is tested against a set of instances classified according to the characteristics presented in Table 1. Each instance type is characterized by a TW type, the maximum length of the requests and the number of nodes (instance size). An instance is defined as *Without*, with *Normal* or with *Large* TWs. For each instance, the minimum size of a request is equal to two, as it includes one delivery point. Depending on the instance type, it can include at most 4 (*Short* requests) or 8 (*Long* requests) pickup and delivery nodes. Finally, our instances contain 25, 50, 100 or 400 nodes. For each of the 24 instance types, we have generated five instances, resulting into a total of 120 instances in our test bed. For example, the instances of type *L*_4_50 represent *Large* TW, *short* requests, 50 nodes and are denoted *L*_4_50_1 to *L*_4_50_5.

Then for each window type (*Without* TW, *Normal* TW and *Long* TW) and for each instance size, each request is treated as follows. The TW of the nodes of the request is modified according to the window type:

- *Without* TW: the TW of the original node is deleted
- *Normal* TW: the TW is slightly enlarged by opening it 150 units earlier and closing it 150 units later
- *Large* TW: the TW is enlarged by opening it 300 units earlier and closing it 300 units later

The request is rejected if it is not feasible. Request generation for a given instance is repeated until reaching the desired instance size.

4.2. Parameter tuning

In order to assess the performance of our ALNS heuristic for solving the MPDPTW, a preliminary set of 24 instances was

Table 1
Instance types.

Instance size	Without TW		Normal TW		Large TW	
	Short requests	Long requests	Short requests	Long requests	Short requests	Long requests
25	W_4_25	W_8_25	N_4_25	N_8_25	L_4_25	L_4_25
50	W_4_50	W_8_50	N_4_50	N_8_50	L_4_50	L_8_50
100	W_4_100	W_8_100	N_4_100	N_8_100	L_4_100	L_8_100
400	W_4_400	W_8_400	N_4_400	N_8_400	L_4_400	L_8_400

Table 2
Quality of the solution when changing the number of ALNS iterations.

# of iterations	Gap (%)	Time (seconds)
25,000	0.32	5.1
50,000	0.24	8.9
100,000	0.17	16.6
200,000	0.17	30.9
400,000	0.21	60.8

Table 3
Quality of the solution when changing the number of requests to be removed.

Min # of requests	Max # of requests	Gap (%)	Time (seconds)
min(3, 0.05r)	min(8, 0.2r)	0.59	4.4
min(4, 0.1r)	min(12, 0.3r)	0.43	5.7
min(6, 0.15r)	min(18, 0.4r)	0.37	7.8
min(8, 0.2r)	min(20, 0.5r)	0.40	9.3
min(10, 0.25r)	min(22, 0.6r)	0.38	10.9
min(12, 0.3r)	min(24, 0.7r)	0.42	12.6

chosen in order to tune the parameters of the heuristic. We use the following parameters for the algorithm. We initiated each parameter to the values reported in Ropke and Pisinger (2006). We then solved the selected instances changing one parameter at a time, and computing the average gap of that particular configuration with respect to the best result obtained for each instance. In the sequence presented now, we varied the following parameters and tested the values reported next.

We have first changed the number of ALNS iterations allowed to run before stopping. The initial value was set to 25,000 iterations, and we have tested running it for 50,000, 100,000, 200,000 and 400,000 iterations, obtaining the solutions reported in Table 2. Based on these results, we observe that the running times increase significantly, but the results stop improving at some point. For these reasons, the option 100,000 iterations was selected.

Next we have changed the minimum and maximum number of requests to be removed at each iteration. The values reported on Table 3 were tested. There was a clear trend of improvements in the results when the number of requests removed was initially increased, but this trend reversed and never decreased again, so we chose the configuration consisting of the interval [$min(6, 0.15r)$, $min(18, 0.4r)$].

The temperature was set as in Pisinger and Ropke (2007). The initial temperature T_{init} is set by measuring the distance from the initial solution, and having a solution that is w percent worse being accepted with 50% probability. As in the original paper, w is equal to 0.05. By using this strategy, we have tested an initial temperature that is set to $(1+w)*d$, $(1+w)*d*c^{3000}$ and $(1+w)*d*c^{10000}$ where d is the distance in the initial solution. The final temperature (T_{min}) is set to $(1+w)*d*c^{30000}$, and the cooling rate c is set to 0.9995. The results of these experiments are reported on Table 4. Based on these results, we have then decided to start the algorithm with the temperature at $(1+w)*d$.

We have also evaluated the usefulness of each operator and each insertion method, in order to assess whether to keep all of them in the final version of the algorithm. In that sense, we have first tested the algorithm using only one insertion algorithm, as reported in Table 5. Based on these results, using only a random insertion order yielded by far the best and fastest results, hence we have decided to always use this method.

Regarding the ALNS operators, we start with the removal ones. To this end, we have run the algorithm with all but one operator at a time, obtaining the results shown on Table 6. As no operator seemed to be useless, we have kept all operators in the list.

Table 4
Quality of the solution when changing the initial temperature.

Initial temperature	Gap (%)	Time (seconds)
$(1+w)*d$	0.42	8.1
$(1+w)*d*c^{3000}$	0.71	7.9
$(1+w)*d*c^{10000}$	0.80	7.9

Table 5
Quality of the solution when changing the insertion method.

Used insertion method	Gap (%)	Time (seconds)
Most expensive	0.99	7.3
Random	0.43	6.9
Cheapest	1.21	7.5
Simple	1.38	6.4

Table 6
Quality of the solution when using all but one removal operator.

Used removal operator	Gap (%)	Time (seconds)
All	0.46	6.9
No random	0.47	7.3
No related	0.49	6.7
No worst node	0.52	6.8
No worst req	0.57	6.6

Table 7
Quality of the solution when using all but one insertion operator.

Used removal operator	Gap (%)	Time (seconds)
All	0.39	7.0
No Seq	0.44	7.1
No regret 3	0.39	7.2
No N regret 3	0.44	7.2
No k regret	0.35	6.8
No Nk regret	0.42	6.9

Table 8
Quality of the solution when using local search techniques.

Used removal operator	Gap (%)	Time (seconds)
None	0.3971	7.0
On start	0.4327	7.0
On start, on new best	0.3765	7.4
On start, on update	0.4138	9.7
All	0.4620	9.9

Table 9
Detailed ALNS heuristic results.

Instance	# requests	# nodes	Best solution	Avg solution	Time (seconds)	Deviation (%)
W_4_25	8.6	26.0	3079.90	3087.41	2.73	0.29
W_8_25	6.0	26.8	3047.18	3047.18	3.29	0.00
W_4_50	17.2	50.6	5108.32	5117.50	5.40	0.18
W_8_50	10.8	51.0	4970.50	5043.79	5.97	1.43
W_4_100	34.2	101.2	8432.62	8514.17	22.61	0.99
W_8_100	21.6	103.2	9221.80	9273.91	16.90	0.58
W_4_400	401.0	133.8	24523.29	24850.12	71.62	1.38
W_8_400	402.4	81.6	29462.98	29802.48	92.06	1.19
N_4_25	8.6	26.0	4734.83	4734.83	2.10	0.00
N_8_25	6.0	26.8	4646.16	4646.16	2.37	0.00
N_4_50	17.2	50.6	8923.03	8923.03	4.30	0.00
N_8_50	10.8	51.0	8521.67	8521.67	3.95	0.00
N_4_100	34.2	101.2	15217.64	15252.95	18.47	0.23
N_8_100	21.6	103.2	16894.20	16894.20	11.67	0.00
N_4_400	401.0	133.8	47004.53	47367.64	60.17	0.78
N_8_400	402.4	81.6	58130.75	58197.63	66.17	0.11
L_4_25	8.6	26.0	4117.56	4117.56	2.30	0.00
L_8_25	6.0	26.8	4424.05	4424.05	2.70	0.00
L_4_50	17.2	50.6	7213.16	7213.27	4.64	0.00
L_8_50	10.8	51.0	7368.10	7368.10	4.47	0.00
L_4_100	34.2	101.2	12060.54	12081.52	19.10	0.18
L_8_100	21.6	103.2	13930.92	13982.88	13.02	0.35
L_4_400	401.0	133.8	36971.82	37290.04	62.24	0.88
L_8_400	402.4	81.6	46406.75	46754.20	74.76	0.74

Table 10
Computational results: ALNS heuristic vs CPLEX.

Instance	ALNS solution	CPLEX solution	Optimality gap (%)$^{(\#solved)}$	Deviation to CPLEX (%)	Avg time (seconds)
W_4_25	3079.90	3079.90	$25.76^{(0)}$	0.00	3483.54
W_8_25	3047.18	3047.18	$24.87^{(0)}$	0.00	3538.86
W_4_50	5108.32	5108.32	$44.70^{(0)}$	0.00	3537.00
W_8_50	4970.50	4970.50	$46.10^{(0)}$	0.00	3553.83
W_4_100	8432.62	8432.62	$54.92^{(0)}$	0.00	3600.00
W_8_100	9221.80	9221.80	$58.68^{(0)}$	0.00	3600.00
W_4_400	24523.29	–	–	–	–
W_8_400	29462.98	–	–	–	–
N_4_25	4734.83	4734.83	$0.00^{(5)}$	0.00	32.55
N_8_25	4646.16	4646.16	$0.00^{(0)}$	0.00	314.91
N_4_50	8923.03	8923.03	$40.11^{(5)}$	0.00	3495.59
N_8_50	8521.67	8521.67	$35.21^{(1)}$	0.00	3097.24
N_4_100	15217.64	15217.64	$59.22^{(0)}$	0.00	3600.00
N_8_100	16894.20	16894.20	$66.51^{(0)}$	0.00	3530.45
N_4_400	47004.53	–	–	–	–
N_8_400	58130.75	–	–	–	–
L_4_25	4117.56	4117.56	$13.61^{(2)}$	0.00	2124.43
L_8_25	4424.05	4424.05	$11.98^{(3)}$	0.00	2251.28
L_4_50	7213.16	7213.16	$48.36^{(0)}$	0.00	3588.43
L_8_50	7368.10	7368.10	$51.54^{(0)}$	0.00	3557.98
L_4_100	12060.54	12060.54	$61.03^{(0)}$	0.00	3600.00
L_8_100	13930.92	13930.92	$67.29^{(0)}$	0.00	3600.00
L_4_400	36971.82	–	–	–	–
L_8_400	46406.75	–	–	–	–

We proceeded the same analysis for insertion operators, with the results reported on Table 7. These results indicate that the k regret operator was deteriorating the results, hence it was not used.

We have also evaluated how often we should use the local search techniques. Hence, we have tested not using it, only on the initial solution, only on initial solution and on new best found solutions, only on the initial solution and on the incumbent solution when updating ALNS parameters, and on all these combinations. The results are reported on Table 8, which led us to choose using the local search techniques on the initial solution and on each new best found solution.

Finally, with this new set of parameters on hand, we have restarted the process of changing one parameter at a time. The best chosen ones have not changed this time.

4.3. Computational results

The experiments reported here have been performed on a desktop computer equipped with a 2.67 gigahertz Intel processor operating under the Scientific Linux 6.3. Each test was allowed to run for a maximum of 1 hour and to use up to 8 gigabytes of RAM.

We start our analysis by running our heuristic algorithm 10 times for each instance. This is to allow us to evaluate the variability of the algorithm with respect to its random choices. The results are presented in Table 9. We report the name of the instance (average over its five replications), the average number of requests and number of nodes in the instances, followed by the best solution value obtained among the 10 runs of the ALNS, the average solution value, the runtime of the best execution, and the average deviation of the average solution with respect to the best one.

Specifically, looking at the instances without TW at the table, it shows that our algorithm is very robust and fast, as even for the largest instances the runtime is only one and a half minutes, and that the deviation is never higher than 1.5% even for instances with 400 nodes. The average running time is only 27 seconds and the average deviation is 0.75% for these instances. We have not identified a difference in terms of difficulty (running time and gap) when solving instances with few (4) or many (8) pickups per request.

The results for instances with normal TW follow, which indicate that these instances are relatively easier to solve than those without TW. This is corroborated by the lower running time, which averaged at 21 seconds, and the very low deviation, with maximum values at about half of those obtained for the cases without TW, and average deviation of only 0.14%. Here we could observe that instances with more pickups per request are easier to solve, as the running time is a bit smaller and the average deviation is always significantly tighter than for smaller requests.

Finally, the results for the instances with large TW are comparable to those of the normal TW instances. The average running time is 22 seconds and the average deviation is 0.27%, with maximum values on the same range as for the case with normal TW.

The previous results indicate that our algorithm is indeed robust and fast, but it does not provide an insight with respect to the quality of the solution against a lower bound. To this end, we have solved the model presented in Section 2 by branch-and-bound using CPLEX 12.7 using the best known solution as a warm start. We have then compared the results in terms of running time and quality of the lower bound obtained to assess the performance of our algorithm.

From the results presented in Table 10, two remarkable information is that CPLEX was never able to improve any solution after running for one hour, and that none of the instances with 400 nodes could even be handled by CPLEX as it lacked memory before the optimization process could even start. Next to that, CPLEX was able to prove optimality for only 16 out of the 120 instances. Obviously, all 16 optimal solutions were obtained by our heuristic. Finally, we note that the optimality gap is generally very large, showing that the current model might not be the best one to tackle this kind of problems. We note that some instances used all the memory available before reaching the time limit, which explains why an average run time of less than one hour is shown while no instance was solved to optimality.

5. Conclusion

This paper described a new heuristic applied for the MPDPTW problem. Our new operators are specifically designed to tackle this problem, and can help other local search algorithms for similar problems, such as the PDP and the SOP, among others. We have also modeled the problem via an integer programming formulation, and designed a complete benchmark set of instances inspired from the PDTW ones. We have extensively tested the many parameters of the heuristic, and after selecting the best ones, we have described the results of our computational experiments. Our results indicate that the heuristic is very robust and fast, and that it can solve instances with up to 400 nodes very efficiently. Moreover, CPLEX applied to the mathematical model described in the paper could not improve any of the solutions obtained by our heuristic. CPLEX could prove optimality for 16 of our solutions, but for the remaining instances CPLEX still shows a very high optimality gap after 1 hour of computing time. Finally, we note that the problem is very difficult and complex, as instances with 400 nodes could not be handled by the mathematical programming solver.

We have provided the first solutions for this problem, and we hope that it motivates other researchers in developing competing algorithms and on developing methods to provide lower bounds for this problem. Despite being tested on randomly generated instances, our method has outperformed a state-of-the-art solver, which could justify applying it to real cases. We have shown that more work is required on the exact solution of this problem, as the initial lower bounds provided by our model are generally weak. Moreover, this work could be extended by considering a heterogeneous fleet with different capacities or with special loading condition features.

Future work could focus on extensions of this problem, such as the case in which some pickup nodes must be visited with a given order as well.

Acknowledgments

This research was partly supported by grants 2014–05764 and 2015–04893 from the Canadian Natural Sciences and Engineering Research Council. This support is gratefully acknowledged. We thank the editor and two anonymous referees for their valuable comments on an earlier version of this paper.

References

Ai, T. J., & Kachitvichyanukul, V. (2009). A particle swarm optimization for the vehicle routing problem with simultaneous pickup and delivery. *Computers & Operations Research, 36*(5), 1693–1702.

Alonso-Ayuso, A., Detti, P., Escudero, L. F., & Ortuño, M. T. (2003). On dual based lower bounds for the sequential ordering problem with precedences and due dates. *Annals of Operations Research, 124*(1–4), 111–131.

Ascheuer, N., Jünger, M., & Reinelt, G. (2000). A branch & cut algorithm for the asymmetric traveling salesman problem with precedence constraints. *Computational Optimization and Applications, 17*(1), 61–84.

Balas, E., Fischetti, M., & Pulleyblank, W. R. (1995). The precedence-constrained asymmetric traveling salesman polytope. *Mathematical Programming, 68*(1–3), 241–265.

Baldacci, R., Bartolini, E., & Mingozzi, A. (2011). An exact algorithm for the pickup and delivery problem with time windows. *Operations Research, 59*(2), 414–426.

Berbeglia, G., Cordeau, J.-F., Gribkovskaia, I., & Laporte, G. (2007). Static pickup and delivery problems: A classification scheme and survey. *Top, 15*(1), 1–31.

Bortfeldt, A., Hahn, T., Männel, D., & Mönch, L. (2015). Hybrid algorithms for the vehicle routing problem with clustered backhauls and 3d loading constraints. *European Journal of Operational Research, 243*(1), 82–96.

Coelho, L., Renaud, J., & Laporte, G. (2016). Road-based goods transportation: a survey of real-world logistics applications from 2000 to 2015. *INFOR: Information Systems and Operational Research, 54*(2), 79–96.

Coelho, L. C., Cordeau, J.-F., & Laporte, G. (2012a). Consistency in multi-vehicle inventory-routing. *Transportation Research Part C: Emerging Technologies, 24*(1), 270–287.

Coelho, L. C., Cordeau, J.-F., & Laporte, G. (2012b). The inventory-routing problem with transshipment. *Computers & Operations Research, 39*(11), 2537–2548.

Desaulniers, G., Desrosiers, J., Erdmann, A., Solomon, M. M., & Soumis, F. (2001). VRP with pickup and delivery. In P. Toth, & D. Vigo (Eds.), *The vehicle routing problem* (pp. 225–242). Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

Desaulniers, G., Madsen, O. B. G., & Ropke, S. (2014). The vehicle routing problem with time windows. In P. Toth, & D. Vigo (Eds.), *Vehicle routing: Problems, methods, and applications*. In *Monographs on Discrete Mathematics and Applications: 18* (pp. 119–159). Philadelphia: MOS-SIAM Series on Optimization.

Escudero, L. F. (1988). An inexact algorithm for the sequential ordering problem. *European Journal of Operational Research, 37*(2), 236–249.

Escudero, L. F., & Sciomachen, A. (1993). Local search procedures for improving feasible solutions to the sequential ordering problem. *Annals of Operations Research, 43*(7), 397–416.

Ezzat, A., Abdelbar, A. M., & Wunsch, D. C. (2014). An extended EigenAnt colony system applied to the sequential ordering problem. In *Proceedings of the IEEE symposium on swarm intelligence, Orlando, US* (pp. 1–7).

Fiala Timlin, M. T., & Pulleyblank, W. R. (1992). Precedence constrained routing and helicopter scheduling: heuristic design. *Interfaces, 22*(3), 100–111.

Goksal, F. P., Karaoglan, I., & Altiparmak, F. (2013). A hybrid discrete particle swarm optimization for vehicle routing problem with simultaneous pickup and delivery. *Computers & Industrial Engineering, 65*(1), 39–53.

Gouveia, L., & Ruthmair, M. (2015). Load-dependent and precedence-based models for pickup and delivery problems. *Computers & Operations Research, 63*, 56–71.

Grangier, P., Gendreau, M., Lehuédé, F., & Rousseau, L.-M. (2016). An adaptive large neighborhood search for the two-echelon multiple-trip vehicle routing problem with satellite synchronization. *European Journal of Operational Research, 254*(1), 80–91.

Guerriero, F., & Mancini, M. (2003). A cooperative parallel rollout algorithm for the sequential ordering problem. *Parallel Computing, 29*(5), 663–677.

Hemmelmayr, V. C., Cordeau, J.-F., & Crainic, T. G. (2012). An adaptive large neighborhood search heuristic for two-echelon vehicle routing problems arising in city logistics. *Computers & Operations Research, 39*(12), 3215–3228.

Hernández-Pérez, H., & Salazar-González, J.-J. (2009). The multi-commodity one–to-one pickup-and-delivery traveling salesman problem. *European Journal of Operational Research, 196*(3), 987–995.

Lee, Y. C. E., Chan, C. K., Langevin, A., & Lee, H. W. J. (2016). Integrated inventory–transportation model by synchronizing delivery and production cycles. *Transportation Research Part E: Logistics and Transportation Review, 91*, 68–89.

Letchford, A. N., & Salazar-González, J.-J. (2016). Stronger multi-commodity flow formulations of the (capacitated) sequential ordering problem. *European Journal of Operational Research, 251*(1), 74–84.

Li, H., & A. Lim, A. (2001). A metaheuristic for the pickup and delivery problem with time windows. *International Journal of Artificial Intelligence Tools, 12*(2), 160–170.

Luo, Z., Qin, H., Zhang, D., & Lim, A. (2016). Adaptive large neighborhood search heuristics for the vehicle routing problem with stochastic demands and weight-related cost. *Transportation Research Part E: Logistics and Transportation Review, 85*, 69–89.

Mancini, S. (2016). A real-life multi depot multi period vehicle routing problem with a heterogeneous fleet: Formulation and adaptive large neighborhood search based matheuristic. *Transportation Research Part C: Emerging Technologies, 70*, 100–112.

Muller, L. F., Spoorendonk, S., & Pisinger, D. (2012). A hybrid adaptive large neighborhood search heuristic for lot-sizing with setup times. *European Journal of Operational Research, 218*(3), 614–623.

Pereira, M. A., Coelho, L. C., Lorena, L. A. N., & de Souza, L. C. (2015). A hybrid method for the probabilistic maximal covering location-allocation problem. *Computers & Operations Research, 57*, 51–59.

Pisinger, D., & Ropke, S. (2007). A general heuristic for vehicle routing problems. *Computers & Operations Research, 34*(8), 2403–2435.

Ropke, S., & Cordeau, J.-F. (2009). Branch and cut and price for the pickup and delivery problem with time windows. *Transportation Science, 43*(3), 267–286.

Ropke, S., Cordeau, J.-F., & Laporte, G. (2007). Models and branch-and-cut algorithms for pickup and delivery problems with time windows. *Networks, 49*(4), 258–272.

Ropke, S., & Pisinger, D. (2006). An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science, 40*(4), 455–472.

Savelsbergh, M. W. P. (1990). An efficient implementation of local search algorithms for constrained routing problems. *European Journal of Operational Research, 47*(1), 75–85.

Seo, D.-I., & Moon, B.-R. (2003). A hybrid genetic algorithm based on complete graph representation for the sequential ordering problem. In *Proceedings of the genetic and evolutionary computation conference* (pp. 669–680).

Shaw, P. (1998). Using constraint programming and local search methods to solve vehicle routing problems. In *Proceedings of the international conference on principles and practice of constraint programming* (pp. 417–431).

Subramanian, A., Drummond, L. M. d. A., Bentes, C., Ochi, L. S., & Farias, R. (2010). A parallel heuristic for the vehicle routing problem with simultaneous pickup and delivery. *Computers & Operations Research, 37*(11), 1899–1911.