The vehicle routing problem simultaneous pickups and deliveries, handling costs, and multiple stacks

Emiel Krol s2054248 Niels Wouda s2533561

Team 3

OR Analysis of Complex Systems

31st March 2020

Abstract

In this paper we introduce and solve the vehicle routing problem with simultaneous delivery and pickups, handling costs, and multiple stacks (VRPSPDMS-H). This problem extends the VRPSPD-H to the setting of vehicles with multiple, linear stacks. In the VRPSPDMS-H, a fleet of vehicles operates from a single depot to service all customers. Each customer has a delivery and pickup demand. All delivery items originate from and all pickup items go to the depot. The items on the vehicles are organised into multiple, linear stacks where only the last loaded item is accessible. Accessing other items requires handling operations. We propose an adaptive large neighbourhood search (ALNS) metaheuristic to solve our problem, in which we embed handling policies, and operators specifically aimed to solve the handling subproblem involving multiple stacks. The performance of our method is assessed on a standard benchmark set of several hundred instances, and compared against optimal solutions for small instances. We find good results for the benchmark instances, and show the handling operators effectively solve the handling problem across multiple stacks. We find optimal or near optimal solutions in 12 out of 16 small instances.

1 Introduction

When purchasing major household appliances, the store often home-delivers the new appliance, and takes back the old device. Most such machinery is bulky, and not easily moved around in the vehicle. Thus, if a pickup item is placed in front of delivery items, it may cause obstruction issues at subsequent customers. Handling these pickup items is a time consuming task and should not be ignored when designing routes to service customers. Generally, such appliances are large, but multiple fit side-by-side in the same vehicle. This suggests it is appropriate to model the vehicle loading plan as one with multiple stacks.

A restricted version of the problem above, with a single stack, is introduced in Hornstra et al. (2020) and described as the vehicle routing problem (VRP) with simultaneous pickup and delivery, and handling costs (VRPSPD-H). They model items in a vehicle as a single linear stack which obeys the last-in-first-out (LIFO) constraint, with items accessible only from the rear. The authors propose an adaptive large neighbourhood search (ALNS) procedure to solve the problem. Veenstra et al. (2017) consider the traveling salesman problem with pickups, deliveries and handling costs (TSPPD-H), a restricted setting of the problem in Hornstra et al. (2020). They also model the vehicle loading plan as a single rear-loaded linear stack obeying a LIFO constraint, and apply a large neighbourhood search heuristic (LNS) to solve the problem. Multiple stacks are studied by Côté et al. (2012) in the context of the TSP, but they do not account for handling costs. Conversely, Battarra et al. (2010); Erdoğan et al. (2012) include handling costs for the TSP, but restrict attention to a single stack.

In this paper, we take the setting of Hornstra et al. (2020), but allow for multiple stacks. We name this problem the VRP with simultaneous pickup and delivery, multiple stacks, and handling costs (VRPSPDMS-H). We propose a mathematical formulation to solve small problem instances optimally, and an ALNS metaheuristic as an alternative to solve realistically sized instances. The quality of our metaheuristic is shown by benchmarking on a large number of problem instances, and compared against optimal solutions on a set of small instances.

Section 2 gives a formal problem definition. Section 3 establishes the VRPSPD-H and the TSPPD-H with multiple stacks as special cases of our problem. Section 4 explains the ALNS me-

Table 1: Definition of input parameters.

Notation	Definition
$N \in \mathbb{N}$	Number of customers.
$V = \{0, 1, \ldots, N\}$	Set of nodes. W.l.o.g., node 0 represents the depot.
$A = V^2$	Set of arcs between every pair of nodes, $i, j \in V$.
$V_c = V \backslash \{0\}$	Set of customer nodes.
$c : A \to \mathbb{R}_+$	Arc weights.
$d \in \mathbb{R}_+^N$	Delivery item amounts, with $d_i \geqslant 0$ the amount to be delivered to customer $i \in V_c$.
$p \in \mathbb{R}_+^N$	Pickup item amounts, with $p_i \geqslant 0$ the amount to be picked up at customer $i \in V_c$.
$Q \in \mathbb{R}_+$	Vehicle aggregate capacity.
$\sigma \in \mathbb{N}$	Number of stacks.
$h \in \mathbb{R}_+$	Unit handling costs.

taheuristic proposed to solve the problem, including handling policies for the handling subproblem. Numerical results are discussed in Section 5. Finally, Section 6 concludes the paper.

2 Problem formulation

This section presents the VRPSPDMS-H, and is structured as follows. In Section 2.1, we formally describe the problem and introduce some notation. The important symbols are summarised in Table 1. Here, and in other places, we use the convenient shorthand $[k] = \{1, \ldots, k\}$ to denote the set of natural numbers up to and including k. Finally, we propose a MIP formulation in Section 2.2.

2.1 Notation

Since our problem is a generalisation of the setting studied in Hornstra et al. (2020) we will, where possible, re-use their notation. The VRPSPDMS-H is defined on a weighted, complete directed graph $G = (V, A)$, with $V = \{0, 1, \ldots, N\}$ the set of vertices and $A = V^2$ the set of arcs between every pair of nodes $i, j \in V$. W.l.o.g., we let node 0 represent the depot, such that $V_c = V \backslash \{0\}$ is the set of consumer nodes. Let $c(i, j) \geqslant 0$ denote the weight of arc $(i, j) \in A$, satisfying the triangle inequality. Customer $i \in V_c$ requests a delivery amount $d_i \geqslant 0$, and supplies a pickup amount $p_i \geqslant 0$. All deliveries originate at the depot and must be delivered to the customer, whereas all pickups must return from the customer to the depot. A homogeneous fleet of vehicles is available at the depot, each of aggregate capacity Q. Since we consider vehicles with $\sigma \geqslant 1$ stacks, this works out to a maximum capacity of Q/σ per stack. We consider heterogeneous and atomic pickup and delivery items: an item associated with customer $i \in V_c$ cannot be used to service customer $j \in V_c$, $i \neq j$, and items cannot be split across multiple vehicles or stacks.

For the handling subproblem, we consider the unloading of the delivery item at its destination, and the loading of the pickup item as unavoidable: these are not penalised. We adopt the notion

of an additional operation from Battarra et al. (2010), as the unloading and reloading of an item not related to the servicing of the particular customer at this stop, incurring handling costs h per unit moved. The objective of the VRPSPDMS-H is then to minimise routing and handling costs, while servicing every customer in the problem instance.

2.2 Model

For the routing problem, we let the binary variables x_{ij} track whether arc $(i, j) \in A$ is part of the solution. Such arcs must together form valid routes, that is, they be part of a tour including the depot (node 0). Each leg is associated with a loading plan, consisting of σ stacks, each of N slots. This ensures that, in principle, all customer items could be inserted into the same stack—although in practice, such an assignment likely violates capacity constraints. We let $k \in [\sigma]$ denote the stack, and $\gamma \in [N]$ the position in each stack. The binary variable $\alpha_{ij}^{k,\gamma,\delta}$ tracks whether the delivery item of customer $\delta \in V_c$ is in stack k at position γ on arc $(i, j) \in A$; $\beta_{ij}^{k,\gamma,\delta}$ does the same for the pickup item. For book-keeping, we introduce auxiliary binary variables $r_i^{k,\gamma}$ to indicate whether the item in stack k at position γ was moved at customer $i \in V_c$. Similarly, we let $v_i^{k,\gamma}$ denote the volume moved at customer $i \in V_c$ in stack k and position γ. This notation is a straightforward adaptation from Battarra et al. (2010), whose formulation we take as the basis for our problem. The constraints given in Equations (1) to (3), (5), (8), (9) and (12) to (15) are similarly adapted to the setting of multiple vehicles and stacks. The remaining constraints are newly formulated here, including a sub-route elimination constraint due to Miller et al. (1960) using an auxiliary variable $y_i \in [N]$, $i \in V_c$ (Equation (4)). We propose the following mixed-integer program (MIP) formulation of the VRPSPDMS-H:

$$\text{minimise} \quad \sum_{(i,j)\in A} c(i,j)x_{ij} + h \sum_{i\in V_c} \sum_{k=1}^{\sigma} \sum_{\gamma=1}^{N} v_i^{k,\gamma} \tag{1}$$

$$\text{subject to} \quad \sum_{j\in V} x_{ij} = 1, \qquad i \in V_c \tag{2}$$

$$\sum_{j\in V_c} x_{ji} = 1, \qquad i \in V \tag{3}$$

$$y_i - y_j + (N+1)x_{ij} \leqslant N, \qquad i, j \in V_c \tag{4}$$

$$\sum_{\delta\in V_c} \left(\alpha_{ij}^{k,\gamma,\delta} + \beta_{ij}^{k,\gamma,\delta}\right) \leqslant x_{ij}, \qquad (i,j) \in A, k \in [\sigma], \gamma \in [N] \tag{5}$$

$$\alpha_{i0}^{k,\gamma,\delta} = 0, \qquad i \in V, k \in [\sigma], \gamma \in [N], \delta \in V_c \tag{6}$$

$$\beta_{0j}^{k,\gamma,\delta} = 0, \qquad j \in V, k \in [\sigma], \gamma \in [N], \delta \in V_c \tag{7}$$

$$\sum_{j\in V} \sum_{k=1}^{\sigma} \sum_{\gamma=1}^{N} \sum_{\delta\in V_c} \left(\alpha_{ji}^{k,\gamma,\delta} - \alpha_{ij}^{k,\gamma,\delta}\right) = 1, \qquad i \in V_c \tag{8}$$

$$\sum_{j\in V} \sum_{k=1}^{\sigma} \sum_{\gamma=1}^{N} \sum_{\delta\in V_c} \left(\beta_{ij}^{k,\gamma,\delta} - \beta_{ji}^{k,\gamma,\delta}\right) = 1, \qquad i \in V_c \tag{9}$$

$$\sum_{j\in V}\sum_{k=1}^{\sigma}\sum_{\delta\in V_c\setminus\{i\}}\alpha_{j,i}^{k,\gamma,\delta}=\sum_{j\in V}\sum_{k=1}^{\sigma}\sum_{\delta\in V_c\setminus\{i\}}\alpha_{i,j}^{k,\gamma,\delta},\qquad i\in V_c,\gamma\in[N]\tag{10}$$

$$\sum_{j\in V}\sum_{k=1}^{\sigma}\sum_{\delta\in V_c\setminus\{i\}}\beta_{j,i}^{k,\gamma,\delta}=\sum_{j\in V}\sum_{k=1}^{\sigma}\sum_{\delta\in V_c\setminus\{i\}}\beta_{i,j}^{k,\gamma,\delta},\qquad i\in V_c,\gamma\in[N]\tag{11}$$

$$\left|\sum_{j\in V}\alpha_{ij}^{k,\gamma,\delta}-\sum_{j\in V}\alpha_{ji}^{k,\gamma,\delta}\right|\leqslant r_i^{k,\gamma},\qquad i\in V_c,\gamma\in[N],k\in[\sigma],\delta\in V_c\tag{12}$$

$$\left|\sum_{j\in V}\beta_{ij}^{k,\gamma,\delta}-\sum_{j\in V}\beta_{ji}^{k,\gamma,\delta}\right|\leqslant r_i^{k,\gamma},\qquad i\in V_c,\gamma\in[N],k\in[\sigma],\delta\in V_c\tag{13}$$

$$r_i^{k,\gamma+1}\leqslant r_i^{k,\gamma},\qquad i\in V_c,\gamma\in[N-1],k\in[\sigma]\tag{14}$$

$$v_i^{k,\gamma}\geqslant\sum_{j\in V}\sum_{\delta\in V_c\setminus\{i\}}\left(\alpha_{ij}^{k,\gamma,\delta}d_\delta+\beta_{ij}^{k,\gamma,\delta}p_\delta\right)-M(1-r_i^{k,\gamma}),\qquad i\in V_c,\gamma\in[N],k\in[\sigma]\tag{15}$$

$$\sum_{\gamma=1}^{N}\sum_{\delta\in V_c}\left(\alpha_{ij}^{k,\gamma,\delta}d_\delta+\beta_{ij}^{k,\gamma,\delta}p_\delta\right)\leqslant\frac{Q}{\sigma},\qquad (i,j)\in A,k\in[\sigma]\tag{16}$$

$$x_{ij},\alpha_{ij}^{k,\gamma,\delta},\beta_{ij}^{k,\gamma,\delta}\in\{0,1\},\qquad (i,j)\in A,k\in[\sigma],\gamma\in[N],\delta\in V_c\tag{17}$$

$$y_i\in[N],r_i^{k,\gamma}\in\{0,1\},v_i^{k,\gamma}\geqslant 0,\qquad i\in V_c,k\in[\sigma],\gamma\in[N]\tag{18}$$

Equation (1) states the objective function, split into routing and (linearised) handling costs. Equations (2) and (3) are routing constraints, ensuring each customer is visited once. Equation (4) disallows routes not including the depot. Equation (5) guarantees items can only be transported over used arcs, and that each position in a stack can only contain a single item. Equations (6) and (7) ensure that there can be no deliveries going to the depot and no pickups coming from the depot. Equations (8) and (9) ensure each customer receives their delivery item and that their pickup item is collected. Equations (10) and (11) ensure items unrelated to a specific customer are in the vehicle both before and after visiting the customer. Equations (12) and (13) guarantee items for which no actions are performed at a customer do not change positions after leaving the customer. Observe that these constraints are non-linear: linearising them is straightforward, and software packages such as CPLEX have built-in support for working with absolute values. Equation (14) maintains the LIFO constraint on stacks. Equation (15) tracks the volume moved at each customer $i \in V_c$, for each stack and position in that stack. This counts the volume of each item if it is moved, except for items being delivered to, or picked up from, the customer, as those actions are unavoidable. Note that this constraint is non-negative due to the minimisation objective, and the lower bound in Equation (18). It is tight whenever $r_i^{k,\gamma}$ is active. Finally, Equation (16) ensures stack capacities are not violated, and by extension the overall vehicle capacity is respected.

3 Special cases

In this section we show the VRPSPDMS-H to generalise the VRPSPD-H, and the TSPPD-H with multiple stacks. We also establish the NP-hardness of our problem.

Theorem 3.1 *The VRPSPDMS-H with $\sigma = 1$ is equivalent to the VRPSPD-H.*

Proof. Let an instance be given with $\sigma = 1$. Equation (16) of the mathematical formulation reduces to the vehicle capacity. Any indices dependent on $k \in [\sigma]$ vanish. Then, any optimal solution for this instance will also be an optimal solution for the VRPSPD-H.

Corrolary 3.1 *The VRPSPDMS-H is NP-hard.*

Proof. Follows directly from generalisation of the VRPSPD-H, and its NP-hardness shown in Hornstra et al. (2020).

Theorem 3.2 *The VRPSPDMS-H with one vehicle is equivalent to the TSPPD-H with multiple stacks.*

Proof. Assume the fleet size for the VRPSPDMS-H is restricted to a single vehicle. Then a solution to the VRPSPDMS-H reduces to a single route, and any optimal solution to this instance is also an optimal solution for the TSPPD-H with multiple stacks.

4 Solution approach

We apply the adaptive large neighbourhood search (ALNS) metaheuristic, as explained in Pisinger and Ropke (2019). The metaheuristic consists of several steps. First a constructive initial solution is found, as explained in Section 4.1. Then the algorithm iterates for a fixed number of iterations. In each iteration, it randomly selects a destroy and repair operator from the operator collection (O_D for destroy and O_R for repair operators, respectively), which transform the current solution s into a candidate solution s^c. These operators are described in Sections 4.2 and 4.3. We use an acceptance criterion to determine if the candidate solution should be accepted, which is explained in Section 4.5. If the candidate solution is a new global best, we apply a local search procedure to further improve the new solution, as detailed in Section 4.4. The *adaptive* part of the ALNS meta-heuristic uses updating weights to select the destroy and repair operators to apply. These weights are tracked by ρ_D and ρ_r for destroy and repair operators, respectively. Updating and operator selection is described in Section 4.6. Initially the weights are equal for all operators, but they are modified based on each operator's performance. An outline of the metaheuristic is given in Algorithm 1.

A solution to the VRPSPDMS-H consists of a set of routes. Formally, a solution $s = (r_1, \ldots, r_m)$, with r_i, $i \in [m]$, a tour from the depot along a number of customers, such that all customers are visited by at least one route. The objective value of a solution is then the sum of the objectives of each route. Using evaluation functions f_S for solutions and f_R for routes, we thus have

$$f_S(s) = \sum_{r \in s} f_R(r).$$

In the explanation below, we use a general evaluation function f to obtain the objective value of a solution s, or a route r in a solution. This allows us to write *e.g.* the improvement strategy of Algorithm 2 in a generic fashion, working on both solutions and routes, at the cost of a slight abuse of notation. The context makes clear where to substitute f_S or f_R for f.

Algorithm 1: Adaptive large neighbourhood search.

Input : Initial feasible solution s.
Output: Best observed solution s^*.

$s^* := s$, $\rho_D := (1, \ldots, 1)$, $\rho_R := (1, \ldots, 1)$.
repeat
  Select destroy and repair methods $d \in O_D$, $r \in O_R$ using ρ_D and ρ_R.
  $s^c := r(d(s))$
  if accept(s^c) **then**
    $s := s^c$
  if $f(s^c) < f(s^*)$ **then**
    $s^* :=$ local search(s^c)
  Update ρ_D and ρ_R
until maximum number of iterations is exceeded
return s^*

4.1 Initial solution

The initial solution assigns each customer to its own route. This ensures a solution with no handling costs, but (potentially) high routing costs due to poor vehicle use.

4.2 Destroy operators

Our destroy operators derive from Hornstra et al. (2020). Each destroy operator removes (at least) q customers from the solution.

1. *Minimum quantity removal.* This operator selects q customers based on the smallest total quantities, which we define the sum of the pickup and delivery amounts for each customer. These customers are selected using a skewed distribution, which favours smaller over larger quantities. Our distribution is (decreasing) triangular for the customers up to and including q, and uniformly flat thereafter, for all customers in $\{q+1, \ldots, N\}$. A customer $y \in [N]$ then has probability

$$P(\text{select } y) = \begin{cases} \frac{q-y+1}{\sum_{i=1}^{q} i+N-q} & \text{if } y \leqslant q, \\ \frac{1}{\sum_{i=1}^{q} i+N-q} & \text{otherwise,} \end{cases}$$

   of being selected.

2. *Random customer removal.* Randomly removes q customers from the solution.

3. *Random nearest removal.* Randomly removes q customers from the solution, based on a distance relationship to the customers already removed. The procedure starts by selecting at random one customer, which is then removed and placed into the unassigned customer pool. A customer is selected at random from this pool, and the closest neighbour in the solution is removed as well and placed into the pool. The process continues until q customers are removed.

4. *Random route removal.* Randomly removes whole routes from the solution, until the number of unassigned customers equals or exceeds q.

5. *Worst routing cost removal.* Removes q customers based on their *additional* routing cost, which for each customer is defined as the difference in routing cost with and without said customer in the solution. These q customers are selected using the randomising procedure explained for *minimum quantity removal*.

6. *Worst handling cost removal.* This operator is similar to *worst routing cost removal*, but only considers handling costs.

7. *Worst cost removal.* Uses both routing and handling costs to remove customers from the solution, but otherwise the procedure is the same as in *worst routing cost removal*.

8. *Cross route removal.* The operator selects a random customer in the solution, as well as the customers immediately preceding and succeeding the selected customer, if present. Next, the nearest customer to the randomly selected customer in a different route is obtained, its neighbours are selected, and all selected customers are removed from the solution. The procedure repeats with another random customer until at least q customers have been removed. The operator aims to reduce overlapping routes by removing chunks of related customers in one iteration.

4.3 Repair operators

We implement the repair operators from Hornstra et al. (2020). Each operator repairs the destroyed solution by reinserting all unassigned customers, such that the resulting solution is feasible.

1. *Random repair.* Random repair randomly removes an unassigned customer, and finds a feasible random route and insertion location. If no feasible routes and locations exist, a new route is created for just this customer. The procedure continues until no unassigned customers remain.

2. *Greedy insert.* The greedy insert operator randomly removes a customer from the unassigned customers pool. The optimal customer insertion location and corresponding routing cost are computed for each feasible route. If no feasible route exists, or the cost of creating a new route is lower than the insertion cost for the best feasible route, a new route is created for just this customer; otherwise the customer is inserted into the cheapest, feasible route. The procedure continues until no unassigned customers remain.

3. *Perturbed greedy insert.* This operator is a randomised version of the regular *greedy insert* operator, and aims to break out of potential local optima. Instead of inserting a random customer into the best feasible location, the customer is inserted into the y-th best location, with y a random integer between 0 and $\min\{3, \text{number of feasible locations}\}$. If no feasible locations exist, a new route is created.

The above details where a customer is inserted into a route, but does not explain the handling policy used to determine how to insert the customer's delivery and pickup item into the loading

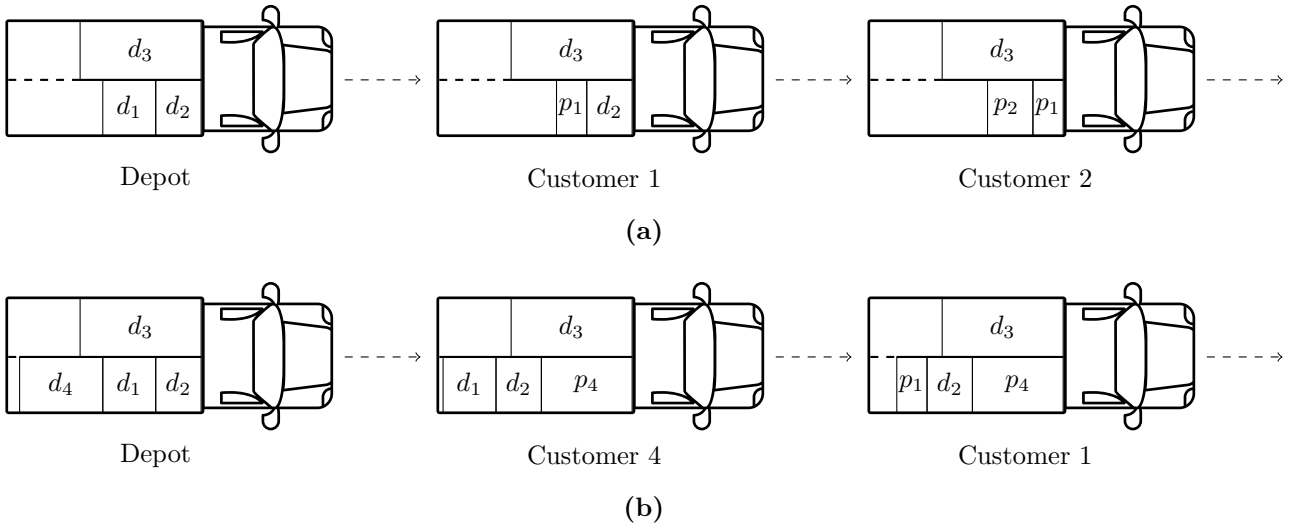

Figure 1: Loading plans before (Figure 1a) and after (Figure 1b) inserting customer 4.

plans along that route. A loading plan describes the item layout in the vehicle after each stop on the route, associated with that leg of the route. Formally, after leaving $i \in V$ for $j \in V$, the loading plan is active on $(i, j) \in A$, and we refer to this as i's loading plan. The first loading plan thus gives the vehicle layout after leaving the depot, and subsequent loading plans the situation just after leaving each customer.

The customer's delivery item must be inserted into all loading plans prior to visiting the customer. We insert the delivery item into each loading plan preceding this location in the rear of the shortest stack of the vehicle, with the stack chosen at the depot. For the pickup item we consider the myopic rear-insertion and front-insertion policies as described in Hornstra et al. (2020). For front-insertion into a stack, all items currently in the stack must be unloaded, the newly inserted pickup item loaded, and all items placed back into the stack. This incurs costs at this stop, but has the benefit that the pickup item never needs to be moved again before returning the depot. Alternatively, rear-insertion is cheap at this stop, but the item will then have to be unloaded to access delivery items at subsequent customers.

We use a mixed insertion policy, which considers a simply rule to assess which of the two policies to apply. At a customer $i \in V$, we select the shortest stack. We then determine the number of delivery items that remain in the stack: if the pickup item is rear-inserted, it will have to be moved (at least) this many times at subsequent stops. We compute $\#\text{deliveries} \cdot p_i$ as the total volume that will have to be moved in this scenario. If the pickup item is front-inserted, we will have to move all items out of the stack, except any pickup items already at the front of the stack (those will not be moved before returning to the depot, thus incurring no additional operations). This amount is compared against the amount that must be moved for rear-insertion, and the pickup item is inserted in the position that results in the lowest amount moved. We have found this approach outperforms the myopic policies, although the local search operators of Section 4.4 construct far better loading plans than any policy considered here.

An example of the described policy is given in Figure 1, for a problem instance with two stacks. Here Figure 1a shows the initial legs of a route, before inserting customer 4, and Figure 1b shows the same route after insertion. At the depot, the delivery item is rear-inserted in the shortest

stack. The vehicle then visits customer 4, where the delivery item is unloaded. The shortest stack is then selected, and a comparison is made: the number of deliveries that remain in the stack (2, for customers 1 and 2) times p_4 is compared against $d_1 + d_2$. In this case, rear-insertion would have resulted in a larger volume to be moved at subsequent customers, and the pickup item is front-inserted instead.

4.4 Local search

When a new best solution is found, a local search procedure is performed to find an even better solution. The nature of our problem invites a dual approach to the local search procedure. First, at the solution level, several operators are applied to improve the routing configurations. These move customers between routes, each operating in a specific neighbourhood. Once this procedure terminates, we apply another local search procedure at the route level. These operators perform limited routing changes, but mostly aim to improve the handling configuration. At each level the operators described below are applied, in order, until no improvement is found. An outline of the search procedure is given in Algorithm 3, using the general improvement strategy of Algorithm 2.

Algorithm 2: Improve, a general strategy of applying a set of operators to improve a given solution or route.

Input : Solution or route to be improved e, and operator set to apply O.
Output: Improved solution or route e', with $f(e') \leqslant f(e)$.

```
Copy e into e'
improved := true
while improved do
    improved := false
    for operator o ∈ O do
        Candidate e^c := o(e')
        if f(e^c) < f(e') then
            e' := e^c
            improved := true
            break
return e'
```

Algorithm 3: Local search procedure.

Input : Feasible solution s.
Output: Feasible solution s', with $f(s') \leqslant f(s)$.

```
s' := improve(s), using the solution level operators.
for route r' ∈ s' do
    r' := improve(r'), using the route level operators.
return s'
```

At the solution level we apply the following operators, based on neighbourhoods described in Savelsbergh (1992), and some operators given in Hornstra et al. (2020).

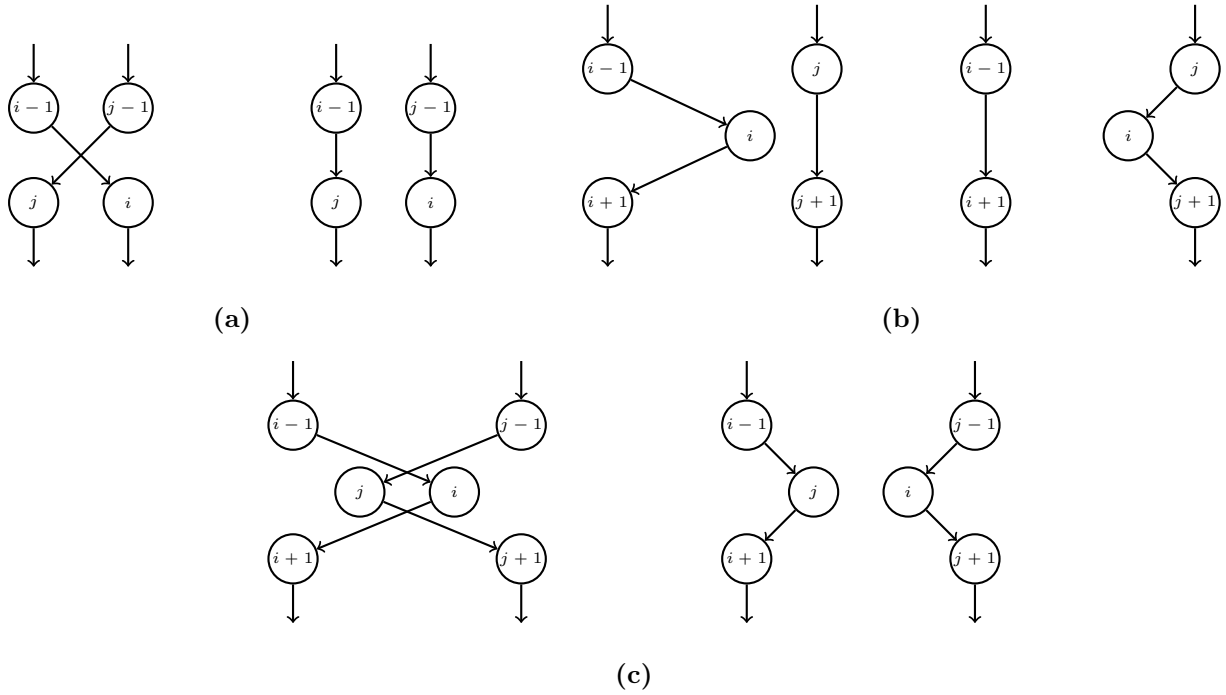

Figure 2: Edge exchange neighbourhoods. Figure 2a details a cross-exchange, Figure 2b a relocate, and Figure 2c an exchange neighbourhood.

1. *Relocate customer.* This operator performs the best feasible customer relocation move, based on routing costs. A relocation is shown in Figure 2b. The explored neighbourhood is in the order of $O(N^2)$. This is similar to the *reinsertion* operators in Hornstra et al. (2020) and Felipe et al. (2009), and commonly seen in local search methods.

2. *Exchange customers.* This operator performs the best feasible exchange move of two customers, based on routing costs. An exchange is shown in Figure 2c. The explored neighbourhood is in the order of $O(N^2)$. This operator is similar to *exchange* in Hornstra et al. (2020).

3. *Cross-exchange customers*

   This operator tries to remove crossing links. If a cross-exchange is performed, the two routes involved swap their last parts (tails). A cross-exchange is shown in Figure 2a. The explored neighbourhood is in the order of $O(N^2)$.

At the route level we apply the following operators. *In-route 2-opt* is based on *intra 2-opt* in Hornstra et al. (2020). *Item re-insert* is a novel operator, and optimally re-inserts the pickup and delivery items into the loading plan. Finally, *pickup push-to-front* is another novel operator, pushing pickup items towards the front of the vehicle as the route progresses. For both *item re-insert* and *pickup push-to-front* we provide pseudo-code in Algorithms 4 and 5, respectively.

We attempted several other strategies at the route level, including optimally solving the TSPPD-H with multiple stacks for the given route, and applying a Markov decision process (MDP) to solve the handling subproblem optimally. These optimal approaches quickly turn computationally intractable, thus limiting their value in practice. Nonetheless, in particular the optimal loading

plans of the MDP provided insight into the handling subproblem, and inspired the two route-level handling operators described below.

1. *In-route 2-opt.* This operator performs the best feasible 2-opt move within a route.

2. *Item re-insert.* This operator finds the best position to re-insert a customer's delivery and pickup item in the loading plan. The stack and insertion index are selected at the depot (for the delivery item) or the customer (for the pickup item), and the items are re-inserted into these stacks at the selected index for all appropriate loading plans. The operator returns once an improving move has been found. The specifics are given in Algorithm 4.

Algorithm 4: *Item re-insert.*

Input : Route r.
Output: Route r', with $f(r') \leqslant f(r)$.

for each customer $i \in r$ **do**
  Copy r into r'.
  Remove the customer's pickup and delivery item from r'.
  Let l'_0 be the depot's, and l'_i the customer's loading plan in r', respectively.
  Find the feasible, minimum-cost stack $k'_d \in l'_0$ and position γ'_d to insert the customer delivery item.
  Insert the delivery item into stack k'_d and position γ'_d for all loading plans from the depot up to the customer.
  Find the feasible, minimum-cost stack $k'_p \in l'_i$ and position γ'_p to insert the customer pickup item.
  Insert the delivery item into stack k'_p and position γ'_p for all loading plans from the customer up to the depot.
  if $f(r') \leqslant f(r)$ **then**
    return r'
return r

3. *Pickup push-to-front.* Since pickup items at the front never cause additional handling operations, it is preferred to have pickup items positioned there, rather than in the middle or rear of each stack. Moving items in that direction naïvely might incur large handling costs, thus offsetting any potential benefit. This operator pushes pickup items towards the front, while avoiding such costs. The algorithm is detailed in Algorithm 5. The operator returns once an improving move has been found.

   This operator immediately suggests another: *delivery push-to-rear.* Although we implemented this operator, it did not yield a significant improvement. We explain this by noting that pushing pickups to the front already implies deliveries are pushed to the rear, and thus that one operator implies the other.

4.5 Acceptance criterion

Motivated by the good results obtained by Santini et al. (2018) for the capacitated VRP, we compare the simulated annealing (SA) and record-to-record travel (RRT) acceptance criteria for our problem.

Algorithm 5: *Pickup push-to-front.*

Input : Route r.
Output: Route r', with $f(r') \leqslant f(r)$.

```
for each customer i ∈ r do
    for each customer j ∈ r not visited before customer i do
        Copy r into r'.
        for each loading plan l' from j onwards in r' do
            Find the stack k' ∈ l' containing the customer's pickup item.
            Let γ' be the position of the pickup item in stack k'.
            Swap the pickup item with the item at γ' + 1.
        if f(r') ⩽ f(r) then
            return r'
return r
```

SA accepts a candidate solution s^c with probability

$$P(\text{accept } s^c) = \exp\left\{\frac{f(s^c) - f(s)}{T}\right\},$$

with T the temperature in the current iteration, and s the current solution. The initial (T^{start}) and final (T^{end}) temperatures are set at the start of the metaheuristic procedure. The temperature T is initially set to T^{start}, and then decreased in every iteration by multiplying the temperature of the previous iteration by a cooling rate $\tau \in (0, 1)$.

The RRT structure is very similar, but instead uses an updating threshold to accept candidate solutions. A candidate solution s^c is accepted when

$$f(s^c) - f(s^*) \leqslant T,$$

with T the threshold in the current iteration, and s^* the best solution observed so far. The initial (T^{start}) and final threshold (T^{end}) levels are set at the start of the metaheuristic procedure. The threshold T is initially set to T^{start}, and then decreased linearly in each iteration as

$$T := \max\{T^{\text{end}}, T - \tau\},$$

using a parameter $\tau > 0$.

4.6 Updating and operator selection

We apply the same roulette wheel mechanism when determining which operator to select as outlined in Ropke and Pisinger (2006). If we have $k \in \mathbb{N}$ operators, each with weights $w_i \geqslant 0$, $i \in [k]$, the probability of selecting operator j is given by

$$P(\text{select } j) = \frac{w_j}{\sum_{i=1}^{k} w_i}.$$

We maintain two lists of such weights: one for the destroy operators (ρ_D), and one for the repair operators (ρ_R).

To update the weights of the destroy and repair operators, we determine their performance based on four measures: (i) a new best solution is found, (ii) the current solution is improved, but the global best solution remains unchanged, (iii) the solution is accepted as the new one, without improving its objective, and (iv) the solution is rejected. Each outcome is assigned a factor ω_i, $i \in [4]$. For a given operator j and observed outcome i, the weights are updated as a convex combination of the original weight and the observed outcome, as

$$w_j := \theta w_j + (1 - \theta)\omega_i,$$

using a decay parameter $\theta \in [0, 1]$, which controls how quickly the metaheuristic responds to changes in the effectiveness of its operators. As we cannot differentiate the effect of the destroy and repair operators in a single iteration, both are updated by the same factor.

5 Computational results

The ALNS metaheuristic is implemented in Python using the `ALNS` package, and the mathematical model of Section 2.2 is implemented using the Python bindings for CPLEX 12.9.0. All our experiments involving the metaheuristic run on commodity hardware, in particular, for each instance we used a single Intel Xeon E5 2680v3 2.5 GHz core, 1GB of memory, and an hour of runtime on the Peregrine computer cluster. The mathematical formulation was granted considerably more resources: eight processor cores of the same type, 48GB of memory, and some 48 hours of runtime per instance.

In Section 5.1 we tune the metaheuristic's parameter settings. In Section 5.2 we compare the metaheuristic's performance with the optimal results found by the mathematical formulation of Section 2.2 for several small instances. In Section 5.3 the metaheuristic's performance on larger instances is briefly discussed. Finally, in Section 5.4 operator effectiveness is investigated.

5.1 Tuning

In the description of our solution approach in Section 4 we proposed several parameters for the ALNS procedure. Here, we provide details about the parameter settings. For the purpose of parameter tuning, we perform a grid-based search, evaluating several levels for each parameter in question. For the weight factors $\omega = (\omega_1, \ldots, \omega_4)$ we consider $(25, 10, 1, 0.8)$ and $(40, 8, 2, 1)$, the latter rewarding globally best solutions by a larger weight increase than the former. For the decay parameter θ, we consider the values 0.5, 0.7, and 0.9. The lower values suggest a rather flexible metaheuristic, quickly adapting to newly performant operators, whereas a value of 0.9 indicates a more rigid operator evaluation. For the number of customers to remove in each iteration, q, we consider a degree of destruction of $0.1|V_c|$ and $0.2|V_c|$, that is, we alter either ten or twenty percent of the solution in each iteration. To determine the maximum number of iterations for the ALNS procedure, we sequentially changed the number of iterations, indeed finding that the quality of our solutions increases with the number of iterations. Analogously to Hornstra et al. (2020) and earlier Ropke and Pisinger (2006), we find that a maximum of 25000 iterations yields good solutions at reasonable computational cost.

We evaluate the two acceptance criteria of Section 4.5 as follows. For the SA criterion, we use a starting temperature of 2000, and select a step size such that after 25000 iterations an end

Table 2: Tuning results. The significance levels are obtained through a one-way analysis of variance, testing whether the outcomes for each parameter level suggest a statistically different result.

Parameter	Levels	Mean objective value
ω	$(25, 10, 1, 0.8)$	1239.6
	$(40, 8, 2, 1)$	1240.4
θ	0.5	1239.6
	0.7	1240.7
	0.9	1239.8
q	$0.1\|V_c\|$	1225.1^{**}
	$0.2\|V_c\|$	1254.9^{**}
Acceptance criterion	SA	1232.8^{*}
	RRT	1247.3^{*}

* $\alpha < 0.05$, ** $\alpha < 0.01$

temperature of 1 is reached. For the RRT criterion we select a starting threshold of 200 and a zero end threshold, reached after 25000 iterations with a step size of 200/25000.

We compare the resulting objective values in Table 2, for each parameter and level. The different weight factors ω do not result in significantly different performances. Similarly, the decay parameter θ does not appear to influence solution quality either. The degree of destruction done in each destroy step, q, is a relevant control: a smaller destruction rate results in better solutions, perhaps because this grants the metaheuristic procedure more iterations to explore any one neighbourhood. This comes at the cost of exploring fewer neighbourhoods, but that appears worthwhile. Finally, the SA acceptance criterion outperforms the RRT, such that we will use SA instead of RRT. Observe that for all parameter levels, the outcomes are mostly similar, suggesting the algorithm is robust under any one particular parameter choice. Summarising, for the remainder of our computational study we use $\omega = (25, 10, 1, 0.8)$, $\theta = 0.9$, $q = 0.1|V_c|$, and the SA acceptance criterion.

5.2 Small instances

We compare our metaheuristic on small instances against the optimal solution given by the formulation of Section 2.2, the results of which are given in Table 3. Using only a single run, the metaheuristic finds the optimal solution in 7 out of 16 instances, and a near optimal (within 5%) solution in another 5 out of 16 instances. We suspect in the case of the 4 instances where there is some deviation, the ALNS procedure does not adequately explore solutions involving fewer vehicles. This is not problematic for problem instances of realistic size, but when the number of customers is very small it overlooks good solutions with few vehicles.

The instances with a single stack are presented also in Table 7 of Hornstra et al. (2020), although they restrict the number of vehicles that may be used to 4 for the problem to remain tractable. Of these instances, we find the optimal solution in 7 out of 8 cases, and a near optimal solution for the

Table 3: Small instances, with $N = 5$ customers.

#	N	σ	h	MIP		ALNS	
				Objective	Time (s)	Objective	Time (s)
1	5	1	2	317.74	20.88	317.74	26.73
2	5	2	2	238.26	15.69	248.32	27.47
3	5	1	4	322.08	18.36	322.08	26.87
4	5	2	4	238.26	4.20	246.18	27.30
13	5	1	2	384.12	28.59	384.12	29.55
14	5	2	2	270.37	11.40	300.67	27.35
15	5	1	4	406.79	30.77	406.79	27.72
16	5	2	4	270.37	50.16	313.51	27.45
25	5	1	2	291.80	33.23	292.63	27.54
26	5	2	2	181.37	13.91	234.46	28.93
27	5	1	4	312.24	16.05	312.24	28.67
28	5	2	4	181.37	5.65	234.46	28.82
37	5	1	2	199.93	19.62	199.93	30.80
38	5	2	2	174.27	574.46	180.00	28.43
39	5	1	4	199.93	18.80	199.93	30.39
40	5	2	4	174.27	491.67	181.72	29.23
Average				260.20	84.59	285.45	28.33

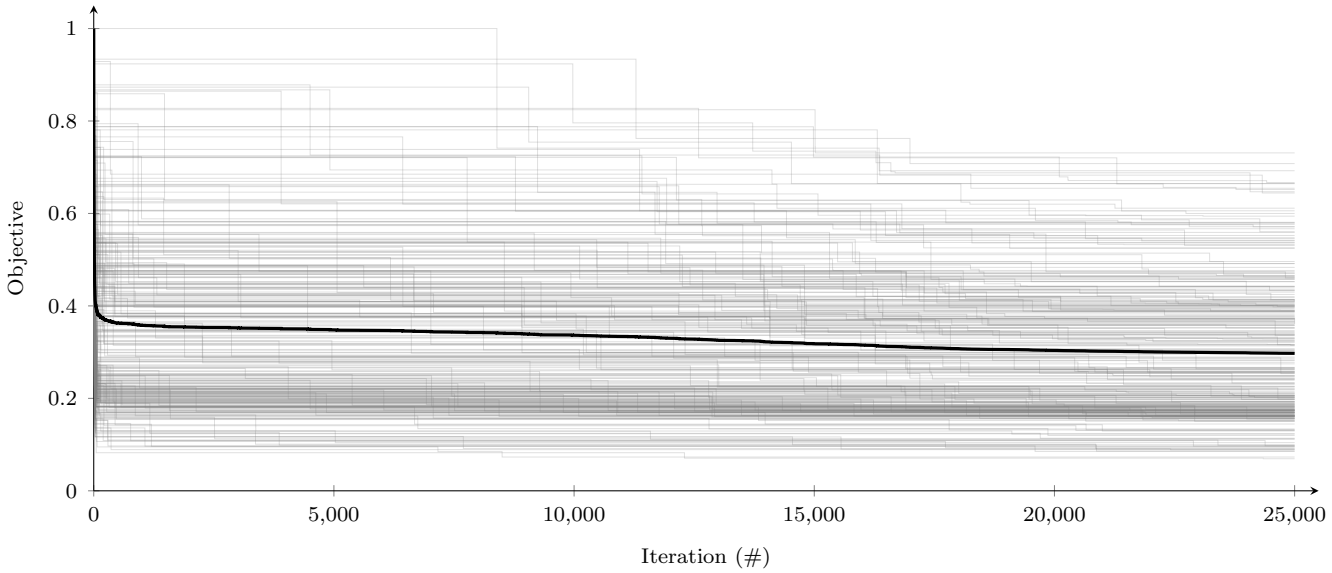

Figure 3: Trajectories for a single run of the ALNS metaheuristic across all large instances. A trajectory displays the best solution observed so far, with the worst (initial) solution normalised to 1. The thick black line presents the mean trajectory, which indicates the metaheuristic quickly leaves the neighbourhood of the initial solution, and continues to find improving moves in subsequent iterations.

final instance. This suggests our metaheuristic can adequately solve the more restricted VRPSPD-H. Finally, the computing times for the metaheuristic are fairly low and consistent, whereas for the MIP these vary considerably. Furthermore, of course, the MIP does not scale beyond such small instances.

5.3 Large instances

We have tested our method on a standard benchmark set of several hundred instances, the results of which can be found in a separate file. In Figure 3 we present a single trajectory of the metaheuristic for each benchmark instance, which shows our method quickly breaks away from the neighbourhood of the initial solution, and then steadily finds improving solutions. We conclude the search procedure is effective.

5.4 Operator quality

Here we investigate our metaheuristic's operator effectiveness. We do this by taking out one operator (*e.g.*, a destroy or repair operator, or one of the local search strategies), and re-solving the problem instances. We perform a single run for each operator taken out, as our aim is not to establish statistical significance, but rather obtain a sense of the relative merit of each operator. As our operators work on either the routing or handling aspect of the solution, we provide a breakdown by both cost components, along with average compute times in Table 4.

Observe first that leaving out any one operator does not appear to impact the overall neighbourhood search too much, with most mean objective values around $1200 - 1250$. An exception

is the *greedy insert* operator, whose removal results in a cost increase, on average, of some 100. This makes some sense: *greedy insert* directs the search towards locally optimal choices, frequently finding better solutions. Since the effects of removing *random repair* and *perturbed greedy insert* are much less pronounced, this suggest a sole greedy insert repair operator, combined with many random destroy operators might well simplify the metaheuristic without sacrificing solution quality.

The local search procedure is split into two parts, solution and route level search. At the solution level, no operator appears indispensable. The effects on computation times, however, are significant: removing *relocate customer* from the search procedure reduces the compute times by 33%. Similarly, removing *cross-exchange customers* achieves a 22% reduction. The local search procedure at the solution level appears ineffective and computationally costly. At the route level, in particular, the two handling operators bring down the handling cost component of the objective value: without *item re-insert*, this is 207.70, its highest value across all operators. Similarly, *pickup push-to-front* reduces handling costs a little further, at only modest computational expense. We conclude these two operators effectively solve the handling subproblem.

We perform another run, dispensing with the solution level search. Here we find a mean objective value of 1224.33, with routing component 1069.64, handling 154.69, in 504.12 seconds on average. Since these costs are largely similar to the reference run having all operators, at significantly reduced computational expense, we recommend local search at the solution level be skipped, and instead only route level search is applied.

6 Conclusions

We introduced the vehicle routing problem with simultaneous delivery and pickups, handling costs, and multiple stacks (VRPSPDMS-H). We showed this problem generalises the vehicle routing problem with simultaneous pickup and delivery, handling costs and a single stack (VRPSPD-H), and is therefore NP-hard as well. We formulated an exact solution approach capable of solving small instances optimally. We proposed an adaptive large neighbourhood search (ALNS) metaheuristic for this problem, and introduce novel search operators for the handling subproblem. We found good results on a large set of standard benchmark instances, and have shown the handling operators effectively solve the handling problem across multiple stacks, and optimal or near optimal solutions in 12 out of 16 small instances. We investigated several parameters settings, concluding that our metaheuristic is robust.

In our instances handling costs were fairly low compared to the routing costs, amounting to some 10% to 20% of average objective cost. The metaheuristic's performance should be investigated further on instances with larger unit handling cost. In addition, more tuning could be done by expanding the parameter search, although this is not likely to result in significantly better performance. Finally, computing times could be reduced by re-implementing the metaheuristic in a language enabling more control over performance trade-offs, rather than rapid prototyping - such as C/C++.

Table 4: Operator quality is assessed by taking out each operator, and running the metaheuristic procedure again across the instances. The resulting solutions are compared on objective values and computational cost. The result of having all operators included is presented in the final row, for reference.

Operator taken out	Objective	Routing	Handling	Time (s)
Destroy operators				
Cross route	1218.26	1063.49	154.77	717.79
Minimum quantity	1214.09	1063.06	151.03	735.64
Random customer	1213.13	1061.48	151.66	723.43
Random nearest	1224.69	1072.07	152.63	735.85
Random route	1228.85	1085.38	143.47	724.31
Worst cost	1216.52	1061.63	154.90	729.77
Worst routing cost	1220.16	1067.88	152.28	746.30
Worst handling cost	1215.13	1062.90	152.22	727.62
Repair operators				
Random repair	1210.42	1061.61	148.82	711.33
Greedy insert	1311.68	1182.40	129.28	719.28
Perturbed greedy insert	1220.57	1041.93	178.65	706.31
Local search				
Relocate customer	1219.39	1069.56	149.83	489.63
Exchange customers	1213.88	1066.58	147.30	725.76
Cross-exchange customers	1221.12	1074.47	146.64	571.11
In-route 2-opt	1226.77	1078.71	148.07	751.21
Item re-insert	1249.59	1041.89	207.70	699.58
Pickup push-to-front	1224.20	1062.22	161.99	713.77
All operators	1218.95	1070.79	148.17	736.34

References

Battarra, M., G. Erdoğan, G. Laporte, and D. Vigo (2010). The traveling salesman problem with pickups, deliveries, and handling costs. *Transportation Science 44*(3), 383–399.

Côté, J.-F., M. Gendreau, and J.-Y. Potvin (2012). Large neighborhood search for the pickup and delivery traveling salesman problem with multiple stacks. *Networks 60*(1), 19–30.

Erdoğan, G., M. Battarra, G. Laporte, and D. Vigo (2012). Metaheuristics for the traveling salesman problem with pickups, deliveries and handling costs. *Computers & Operations Research 39*(5), 1074 – 1086.

Felipe, A., M. T. Ortuño, and G. Tirado (2009). The double traveling salesman problem with multiple stacks: A variable neighborhood search approach. *Computers & Operations Research 36*(11), 2983 – 2993.

Hornstra, R. P., A. Silva, K. J. Roodbergen, and L. C. Coelho (2020). The vehicle routing problem with simultaneous pickup and delivery and handling costs. *Computers & Operations Research 115*, 104858.

Miller, C. E., A. W. Tucker, and R. A. Zemlin (1960, October). Integer programming formulation of traveling salesman problems. *Journal of the ACM 7*(4), 326–329.

Pisinger, D. and S. Ropke (2019). Large neighborhood search. In M. Gendreau and J.-Y. Potvin (Eds.), *Handbook of Metaheuristics* (3rd ed.)., pp. 99–127. Springer International Publishing.

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

Santini, A., S. Ropke, and L. Hvattum (2018). A comparison of acceptance criteria for the adaptive large neighbourhood search metaheuristic. *Journal of Heuristics 24*(5), 783–815.

Savelsbergh, M. W. P. (1992). The vehicle routing problem with time windows: Minimizing route duration. *ORSA Journal on Computing 4*(2), 146.

Veenstra, M., K. J. Roodbergen, I. F. Vis, and L. C. Coelho (2017). The pickup and delivery traveling salesman problem with handling costs. *European Journal of Operational Research 257*(1), 118 – 132.

A Author contributions

The ALNS structure follows directly from Niels' package, written in 2019. The operators were adapted from Hornstra et al. (2020), by both of us: Emiel did several destroy operators, and greedy insert; Niels completed the rest and made sure they fit into a format the package expects. Niels then designed and implemented the local search procedure, while Emiel worked on and implemented the mathematical formulation. A simple validation tool was written early on by Niels, and used to validate the correctness of all output files. Emiel oversaw the parameter tuning, Niels the assessment of operator quality. A rough breakdown along various categories is given in Table 5.

Table 5: Author contributions.

Task	Emiel	Niels
Design		
Metaheuristic structure	50%	50%
Handling policies	60%	40%
Local search	10%	90%
Mathematical model	100%	0%
Programming		
Program structure, I/O	0%	100%
Policies and operators	35%	65%
Local search	0%	100%
Mathematical model	100%	0%
Misc.		
Verification and validation of code[1]	20%	80%
Design and execution of experiments	50%	50%
Writing		
Mathematical model	80%	20%
Solution approach	25%	75%
Other sections	50%	50%

[1] With important contributions made by TravisCI, for running our tests and validating new implementations.