



Real-World Vehicle Routing Using Adaptive Large Neighborhood Search

Vojtěch Sassmann¹, Hana Rudová¹  , Michal Gabonnay²,
and Václav Sobotka¹ 

¹ Faculty of Informatics, Masaryk University, Brno, Czech Republic
hanka@fi.muni.cz

² Wereldo.com, Brno, Czech Republic

Abstract. Our work addresses a real-world freight transportation problem with a broad set of characteristics. We build upon the classical work of Ropke and Pisinger [10] and propose an effective realization of the adaptive large neighborhood search (ALNS) with constant time complexity for a large portion of frequent steps in insertion and removal heuristics at the cost of additional pre-calculations. Our minimization process handles different objectives with cost models of heterogeneous vehicles. We demonstrate the generic applicability of the proposed solver on various vehicle routing problems. With the help of the standard Li & Lim benchmarks [6] for pickup and delivery with time windows, we show its capabilities compared to the best-found solutions and the original ALNS. Experiments on real-world delivery routing problems provide a comparison with the original implementation by the company Wereldo in OR-Tools [8], where we achieve significant cost savings, faster runtime, and memory savings by order of magnitude. Performance on large-scale real-world instances with more than 300 vehicles and 1,200 pickup and delivery requests is also presented, achieving less than an hour runtimes.

Keywords: Vehicle routing problem · Pickup and delivery with time windows · Adaptive large neighborhood search · Freight transportation

1 Introduction

With today's scale, freight transportation presents a broad field of study for research [13]. The vehicle routing problem (VRP) and its numerous variants can be used to formalize real-world transportation problems. State-of-the-art classification and taxonomic review of VRP can be found in [1, 3, 11]. The specific reviews [4, 9] classifies VRP and its variants based on the used metaheuristics. Among them, the adaptive large neighborhood search (ALNS) [10] belongs to common metaheuristics applied to routing problems. Even though its adaptiveness was not found to be crucial for the improvement [12], we will demonstrate its generic application on complex and large-scale routing problems, which can have very different characteristics.

The problems we consider have heterogeneous fleets as it is the case for many other works reviewed in [5]. We will work with pickup and delivery with

time windows (PDPTW), where each request includes the service of pickup and delivery locations, which may have multiple time windows. As usual, requests have their capacities, and we need to consider both volume in the number of pallets and weight. Time constraints are also represented by the necessary service time at each physical location and the maximal route duration constraint. The maximal number of physical locations per route is constrained as well. Finally, we have multiple depots available [7]. Altogether, our problems combine multiple constraints, as is common in rich VRPs [2].

Let us now summarize the specific contributions of this work.

- We describe the formal model of our problem as an extension of [10] with a unique combination of real-world constraints and objectives.
- We extend ALNS to handle minimization with real-world objectives and propose an efficient realization of its insertion and removal heuristics. It allows for a constant time complexity for many frequent operations instead of the linear one wrt. to the length of the route. To achieve that, we have identified crucial information to store and pre-calculate.
- We verify our implementation on the standard Li & Lim [6] benchmarks to compare it with the best so-found results and with the original ALNS.
- We compare our solver with implementation in OR-Tools [8] provided by the company Wereldo and demonstrate significant improvement in the solution’s costs, faster computation, and memory savings by order of magnitude.
- We demonstrate the efficiency of our solver on a large scale real-world problem with more than 300 vehicles and 1,200 pickup and delivery requests, where results were computed within less than an hour.

Overall, we have proposed and implemented a generic algorithm capable of solving a broad family of routing problems. We are glad to see that the Wereldo company nowadays uses the described solver for their everyday operation and specific case studies.

2 Mathematical Model

In this section, we describe the formal mathematical model of the PDPTW variant as an extension of the model described by Ropke and Pisinger [10].

We consider n requests and m vehicles. We denote the pickup nodes by P , where $P = \{1, \dots, n\}$ and the delivery nodes by D , where $D = \{n + 1, \dots, 2n\}$. Request i is represented by a pickup node i and a delivery node $i+n$, i.e., there are $2n$ tasks to be serviced. We denote the set of all vehicles by K , where $|K| = m$. We also denote the set of all pickup and delivery nodes by N ($N = P \cup D$). This set represents all locations that have to be visited. For each vehicle k ($k \in K$) we consider its starting terminal τ_k ($\tau_k = 2n + k$) and its ending terminal τ'_k ($\tau'_k = 2n + m + k$). In practice, the ending and starting terminals can represent the same location. In our model, we define them separately to distinguish between the start and end of vehicles. For each pickup or delivery node i ($i \in N$), we consider a service time s_i needed at each location to load or unload the goods.

Each vehicle can have different pricing. For each vehicle k , we define its minimum price C_k^{\min} , the price per kilometer when the vehicle is empty C_k^{empty} , and the price per kilometer when the vehicle is fully loaded C_k^{full} . If the vehicle is partially loaded, the price per kilometer is calculated from the C_k^{empty} and C_k^{full} values based on the vehicle's current load. For each vehicle k , we also define its maximum route duration F_k and the maximum number of trips W_k (different physical locations on the route), where $W_k \geq 3$, so each vehicle can carry at least one of the requests. There can be at most three trips for a vehicle with a single request: start-pickup, pickup-delivery, and delivery-end.

Each pickup and delivery can have a different number of possible time windows. We define ρ_i ($\rho_i > 0$) for each node i as the number of its time windows. Then we define the time windows TW_i for each location where $TW_i = \{[a_{ir}, b_{ir}] \mid r \in \{1, \dots, \rho_i\}\}$. a_{ir} and b_{ir} are the earliest and latest possible times, respectively.

For each request, we consider its weight and volume. Therefore, for each location i , we define the differences of weight l_i and the volume h_i after serving the location i . These amounts must be positive numbers for all pickup locations and negative for all delivery locations. For each pickup i and its delivery $i+n$ it has to be true that $l_i = -l_{i+n}$ and $h_i = -h_{i+n}$. This ensures that the amount of loaded goods equals the amount of unloaded goods. Each vehicle $k \in K$ has its weight limit Q_k and a volume limit U_k .

We define a graph with all locations by $G = (V, A)$. Where $V = N \cup \{\tau_1, \dots, \tau_m\} \cup \{\tau'_1, \dots, \tau'_m\}$ and $A = V \times V$. This graph represents all locations from our problem, the pickup and delivery locations, and the starting and ending terminals. For each vehicle k , we define a subgraph $G_k = (V_k, A_k)$ where $V_k = N \cup \{\tau_k\} \cup \{\tau'_k\}$ and $A_k = V_k \times V_k$. Each subgraph G_k contains only the depot of the vehicle k (G contains all depots). For each arc $(i, j) \in A$ we consider its distance d_{ij} ($d_{ij} \geq 0$) and its travel time t_{ij} ($t_{ij} \geq 0$).

We use five decision variables. A binary variable x_{ijk} where $i, j \in V$ and $k \in K$. This variable has a value one if the arc (i, j) is used by a vehicle k and zero otherwise. S_{ik} ($i \in V, k \in K$) is a variable that indicates when the vehicle k arrives at the location i . L_{ik} ($i \in V, k \in K$) is a non-negative integer that indicates the total weight of the goods loaded on the vehicle k after servicing the node i . H_{ik} ($i \in V, k \in K$) is a non-negative integer that indicates the total volume of the goods loaded on the vehicle k after servicing the node i . S_{ik} , L_{ik} , and H_{ik} are only well-defined when the vehicle k visits the location i . z_i ($i \in P$) is a binary variable that indicates if request i is placed in the request bank. The variable is one if the request is placed in the bank and zero otherwise. The request bank is used during the solution process to handle requests not yet assigned to any vehicle.

The total price C_k of vehicle k is defined as

$$C_k = \max(C_k^{\min}, \sum_{(i,j) \in A} x_{ijk} d_{ij} (C_k^{\text{empty}} + (C_k^{\text{full}} - C_k^{\text{empty}}) \frac{L_{ik}}{Q_k}) . \quad (1)$$

The objective function minimizes the weighted sum of the prices of all used vehicles and the number of requests that are not scheduled, i.e., they are kept

in the request bank. Parameters α and β are used to adjust these weights.

$$\alpha \sum_{k \in K} C_k + \beta \sum_{i \in P} z_i . \quad (2)$$

To ensure that each request is either delivered by a single vehicle or that it is placed in the request bank, we define a constraint

$$\sum_{k \in K} \sum_{j \in N} x_{ijk} + z_i = 1 \quad \forall i \in P . \quad (3)$$

If a request is not placed in the bank, its pickup i and delivery $n+i$ have to be performed by the same vehicle k . This constraint is defined as

$$\sum_{j \in V_k} x_{ijk} - \sum_{j \in V_k} x_{j,n+i,k} = 0 \quad \forall k \in K, \forall i \in P . \quad (4)$$

Constraints (5)–(7) together ensures that a correct path from τ_k to τ'_k is constructed for each vehicle k . Each vehicle k has to start its route in its starting location τ_k (5). Also, each vehicle k has to end its route in its ending location τ'_k (6). If a vehicle k visits some location j , except for a depot, the vehicle must also leave this location (7).

$$\sum_{j \in P \cup \{\tau'_k\}} x_{\tau_k, j, k} = 1 \quad \forall k \in K , \quad (5)$$

$$\sum_{i \in D \cup \{\tau_k\}} x_{i, \tau'_k, k} = 1 \quad \forall k \in K , \quad (6)$$

$$\sum_{i \in V_k} x_{ijk} - \sum_{i \in V_k} x_{jik} = 0 \quad \forall k \in K, \forall j \in N . \quad (7)$$

If a vehicle k arrives at a location i at S_{ik} , it has to have enough time to load or unload the goods and travel to the next location j before S_{jk} . However, we want to consider the service time only if the vehicle had to take a trip from i to j . The physical locations of i and j might be the same. In this case, the service time is not needed between i and j . This is ensured by

$$x_{ijk} = 1 \implies S_{ik} + s_i \cdot \text{sgn}(d_{ij}) + t_{ij} \leq S_{jk} \quad \forall k \in K, \forall (i, j) \in A_k \quad (8)$$

where sgn function is used to include/exclude values from the sum when the distance is (non)zero.

The time windows must be kept. For each vehicle k and each of its visited locations i , the time of arrival S_{ik} has to be from one of its intervals $[a_{ir}, b_{ir}]$.

$$a_{ir} \leq S_{ik} \leq b_{ir} \quad \forall k \in K, \forall i \in V_k, \exists r \in \{1, \dots, \rho_i\} . \quad (9)$$

For each pickup i , its corresponding delivery $n+i$ has to be performed after the pickup. This is ensured by

$$S_{ik} \leq S_{n+i,k} \quad \forall k \in K, \forall i \in P . \quad (10)$$

The volume and weight load variables L_{ik} and H_{ik} are set by constraints (11) and (12). The vehicle's weight and volume limits Q_k and U_k are obeyed using the constraints (13) and (14).

$$x_{ijk} = 1 \implies H_{ik} + h_j \leq H_{jk} \quad \forall k \in K, \forall (i, j) \in A_k, \quad (11)$$

$$x_{ijk} = 1 \implies L_{ik} + l_j \leq L_{jk} \quad \forall k \in K, \forall (i, j) \in A_k, \quad (12)$$

$$L_{ik} \leq Q_k \quad \forall k \in K, \forall i \in V_k, \quad (13)$$

$$H_{ik} \leq U_k \quad \forall k \in K, \forall i \in V_k. \quad (14)$$

We have to ensure that each vehicle k starts and ends empty.

$$L_{\tau_k k} = L_{\tau'_k k} = 0 \quad \forall k \in K. \quad (15)$$

We must ensure that all of the maximum route duration constraints F_k and the maximum number of trips W_k are obeyed (sgn function again allows to include/exclude values when the distance is (non)zero).

$$S_{\tau'_k, k} - S_{\tau_k, k} \leq F_k \quad \forall k \in K, \quad (16)$$

$$\sum_{i \in N \cup \{\tau_k\}} \sum_{j \in N \cup \{\tau'_k\}} x_{ij} \cdot \text{sgn}(d_{ij}) \leq W_k \quad \forall k \in K. \quad (17)$$

Finally, we have to set the domains of the used decision variables.

$$x_{ijk} \in \{0, 1\} \quad \forall k \in K, \forall (i, j) \in A_k, \quad (18)$$

$$z_i \in \{0, 1\} \quad \forall i \in P, \quad (19)$$

$$S_{ik} \geq 0 \quad \forall k \in K, \forall i \in V_k, \quad (20)$$

$$Q_k \geq L_{ik} \geq 0 \quad \forall k \in K, \forall i \in V_k, \quad (21)$$

$$U_k \geq H_{ik} \geq 0 \quad \forall k \in K, \forall i \in V_k. \quad (22)$$

3 Our Approach

Our approach uses the adaptive large neighborhood search (ALNS) algorithm based on Ropke and Pisinger [10] (see page 6). In the beginning, we generate an initial solution *formerSolution*, or it can be a solution obtained in an earlier run (Line 2). The current solution s' is iteratively modified by rearranging requests. We decide which neighborhood to search by selecting one removal (Line 6) and one insertion heuristics (Line 7) using a standard adaptive heuristic selection mechanism. The worst removal, Shaw removal, and random removal are complemented by the greedy insertion and regret insertions heuristics [10]. In each iteration, a random number of q requests to remove (Line 8) is generated as in [10]. If it is impossible to insert some of the removed requests (Line 9), these requests are stored in a request bank in the new solution s' . In the reinsertion process, requests from the request bank are also reinserted, if possible. At the end of each iteration, we check if we have found a new best solution (Line 10). Acceptance of the new modified solution (Line 12) is decided by the simulated annealing combined with the heuristics with noisy objective function from [10].

```

1 function ALNS(formerSolution):
2    $s \leftarrow \text{formerSolution}$ 
3    $s_{best} \leftarrow \text{formerSolution}$ 
4   while not reached max iteration do
5      $s' \leftarrow s$ 
6      $h^r \leftarrow$  choose removal heuristic
7      $h^i \leftarrow$  choose insertion heuristic
8     remove  $q$  requests from  $s'$  using  $h^r$ 
9     reinsert unassigned requests to  $s'$  using  $h^i$ 
10    if  $f(s') < f(s_{best})$  then
11       $s_{best} \leftarrow s'$ 
12    if  $\text{accept}(s', s)$  then
13       $s \leftarrow s'$ 
14  end
15  return  $s_{best}$ 

```

3.1 Two-Stage Minimization

To minimize the total vehicle price, we run the ALNS in two stages. In the first stage, we set all of the vehicle's minimum prices C_k^{\min} to zero. This encourages the ALNS to put some of the requests into the larger, more expensive vehicles. For each vehicle, we also set its $C_k^{\text{full}} = C_k^{\text{empty}}$. This is again to achieve greater diversification during the first stage. We take the best solution from the first stage and use it as a former solution in the second stage. In this stage, we use all prices with their original values. Note that for problems without the minimum price C_k^{\min} component, running the ALNS in one stage is sufficient.

This approach was inspired by a similar two-stage approach from [10]. We keep the same approach when solving problems with the standard lexicographic objective aiming to minimize the number of vehicles first and the distances next. In the first stage, the whole fleet is available for the ALNS initially. The search terminates as soon as it finds a feasible solution. In the next ALNS run, one of the used vehicles is removed, and the search is repeated. We repeat the ALNS until a feasible solution with the given number of vehicles is not found (within the iteration limit for the whole first stage). In the second stage of the algorithm, we run the regular ALNS where the former solution is the best feasible solution with the minimum number of vehicles used.

3.2 Insertion Optimization

To build upon the work [10], we propose various methods which allow processing insertions (this section) and removals (next section) effectively. The most time-consuming operation during insertions is finding the best position for a request in a route. Also, this operation is the most time-consuming for the whole solver.

When trying to find the best request position for each route, we must find the best position for the pickup and delivery in the route's actions. A series of actions

represent each route. Each action represents one location $i \in N$. No two actions can represent the same location since each location should be served by exactly one vehicle (however, they can represent the same physical location). Suppose we have a route with y actions. After adding the new request, the number of actions will increase by two. This new request is represented by a new pickup action η^p and a new delivery action η^d . The possible positions κ^p for the pickup action η^p can be from the interval $[0, y]$. With such selected pickup position κ^p , the possible delivery action η^d has a position κ^d from interval $[\kappa^p + 1, y + 1]$. An example of such a new route with positions from 0 to 7 is $(\eta_0, \eta^p, \eta_1, \eta_2, \eta^d, \eta_3, \eta_4, \eta_5)$ where $\kappa^p = 1$ and $\kappa^d = 4$. Actions η_0, \dots, η_5 are already present in the route. η^p and η^d are the new pickup and delivery actions.

With these possible combinations of κ^p and κ^d , we have to verify that such route satisfies all of the constraints defined in Sect. 2. We also need to calculate the price of the extended route such that we can compute the insertion cost. This is performed in this given order: (1) validation of capacities (weight + volume), (2) validation of the maximum number of trips, (3) validation of time constraints, and (4) calculation of the cost. If any of these validation fails, the following operations are skipped. This order is essential as the first two validations are the fastest, and the cost should be calculated only for valid routes. The process of validations 1–3 is described next. The cost of the solution is computed by iterating over the whole route since it was not identified as a crucial bottleneck.

Validation of Capacities. In Eqn. 11 and 12, we have defined constraints for vehicles’ weight and volume capacities. Both constraints can be handled using the same validation mechanism which we demonstrate on the weight. When we add a new request at positions κ^p and κ^d , we have to check that the vehicle’s capacity is not overreached. We have to check this at the κ^p position and all other positions between κ^p and κ^d . Actions before κ^p and after κ^d are not affected in terms of capacity and do not need to be checked.

The number of actions that need to be checked is linear in terms of the route’s length. However, with some pre-calculation, it can be done in a constant time. We keep several values for all actions η_e , where $e \in \{0, \dots, y - 1\}$ and y is the route’s length) for all routes. First, we calculate for each action η_e its weight reserve Δ_e^q . This value represents the difference in the vehicle’s weight capacity and the weight of loaded goods after performing the given action. An example of a route with its weight reserves is shown in Table 1 (left).

Then, for each action η_e , we keep a list of values $\nu_{e,j}$ where $j \in \{e, \dots, y - 1\}$. This list represents the minimum weight reserve of actions from interval (e, j) , i.e., $\nu_{e,j} = \min\{\Delta_e^q, \dots, \Delta_j^q\}$. Note that for each action η_e , we keep a different number of the $\nu_{e,j}$ values. The action at the start of the route has the maximum number of such values equal to the route’s length. The last action has only one value for itself. An example of these calculated values is shown in Table 1 (right) for the route on the left. The best way to calculate these values is to start with j set to its maximum value $j = y - 1$ and decrease it. This way, we can reuse

Table 1. Example of a route with weight reserves for a vehicle capacity 200.

	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$
action	η_0	η_1	η_2	η_3	η_4	η_5
weight change	+20	+70	+50	-50	-20	-70
current load	20	90	140	90	70	0
weight reserve Δ_c^q	180	110	60	110	130	200

	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$
$e = 0$	180	110	60	110	130	200
$e = 1$	110	60	60	110	130	
$e = 2$	60	60	60	110		
$e = 3$	60	60	60			
$e = 4$	60	60				
$e = 5$	60					

the previously calculated values. To calculate value $\nu_{e,j}$ we can use

$$\nu_{e,j} = \left\{ \begin{array}{l} \Delta_j^q, \\ \min(\Delta_j^q, \nu_{e-1,j+1}), \end{array} \text{ for } e = 0 \right\}. \quad (23)$$

When we try to assign a new request at positions κ^p and κ^d , we only need to check if the request's weight is lower or equal to the value $\nu_{\kappa^p-1, \kappa^d-2}$. The value $\kappa^p - 1$ represents a position of the action right before the pickup position κ^p . The value $\kappa^d - 2$ represents the position of the action right before the delivery position κ^d . For insertion optimization example on page 7, we get action η_0 for $\kappa^p - 1$ and we get action η_2 for $\kappa^d - 2$.

This mechanism does not work when we try to place the new pickup at the beginning of a route ($\kappa^p = 0$) or when the route is empty. In the first case, we compare the request's weight with the value $\nu_{\kappa^p, \kappa^d-2}$. In the second case, when the route is empty, we compare the request's weight with the vehicle's capacity.

If the capacity check fails, we can also perform one more optimization; we can skip the next possible delivery positions, which are higher than the current κ^d . If the capacity check fails for κ^d , it will always fail for any other higher values.

This pre-calculation has a quadratic time complexity relative to the route length. It seems inefficient to do such pre-calculation. The important thing is that this pre-calculation is run only when a route is truly modified. When we calculate cost increases for all of the unassigned requests, we can reuse these values for all of them. When we finally insert one request, we must recalculate only the route where the request was just inserted.

Validation of Maximum Number of Trips. In Eq. 17, we limit each vehicle's maximum number of trips. A trip can happen between two actions. If the distance between the locations of these actions is non-zero, the vehicle must perform a trip. When assigning a new request to a route, the check of the overall number of trips has linear time complexity relative to the route's length. We again propose a constant time validation.

It is insufficient to compare the number of actions in the route since this does not correspond to the number of trips due to the possibly same physical locations. There can be a sequence of actions representing the same real-world location, for example, pickups of multiple requests at the same location. Between such actions, there are no trips. We keep information about the number of trips on each route. This value is recalculated whenever we remove or reinsert some requests (with a linear time complexity relative to the length of the route). When

we try to assign a new request at positions κ^p and κ^d , we check if we have added some new trips.

For this new request, we define i^p as the pickup location of this request. The i^p is a location from the set of all pickup locations P . We find the location i^{p-1} where the vehicle was previously. If $\kappa^p = 0$ holds, the previous location is the vehicle's starting location τ_k . If $\kappa^p > 0$, we get the previous location from the action on position $\kappa^p - 1$. We also need to find the next location i^{p+1} , where the vehicle goes from i^p . If $\kappa^p = \kappa^d - 1$, the next location is the new request's delivery location. Otherwise, the next location is taken from the action on position $\kappa^p + 1$.

We define $trips^p$ as the number of new trips caused by the new pickup being added; initially, we set it to zero. If the distance between i^{p-1} and i^p is not equal to zero, we increase the $trips^p$ by one. If the distance between i^p and i^{p+1} is not equal to zero, we increase the $trips^p$ by one. Before adding the new pickup, we also need to check if there was already a trip between i^{p-1} and i^{p+1} . If so, we decrease the $trips^p$ by one. This gives us the number of new trips caused by the new pickup. Similarly, we can calculate $trips^d$, which represents the number of new trips caused by the new request's delivery. We also need to check if we did not add the same trip twice in cases where the new delivery is right after the new pickup ($\kappa^p = \kappa^d - 1$). If so, we decrease the $trips^d$ by one.

We add up $trips^p$ and $trips^d$ with the previous number of trips, which has been pre-calculated, to get the total number of trips $trips^{total}$. Finally, we can check that the $trips^{total} \leq W_k$.

Validation of Time Constraints. Constraints specified in Eqs. 8–9 define valid arrival times for all vehicles. They ensure that all vehicles arrive at their locations in corresponding time windows. To check all these constraints, we iterate over the whole route and calculate arrival times. We assume that each vehicle takes the first time window, which is possible to use.

In Eqn. 16, we have also defined a maximum route time. When checking the time windows, we must calculate arrival times for all actions. This allows us to check the maximum duration time without additional calculation.

Same Locations Optimizations. In our model, we have mentioned that some locations can represent the same physical locations. Among a route's actions, there can be a sequence of actions representing the same physical location. The order in which these actions are processed in the location cannot affect the route's price because we consider the service time only for the last of them, and also, in the company instances, the service times are all equal. This allows us to optimize the best positions of η^p and η^d . If we detect such a sequence of actions, and the action η^p represents the same real-world location, we try to assign the action η^p only at the beginning of this sequence. We also do the same when finding the best position of η^d .

This optimization is critical to solving problems with many same physical locations efficiently. Notably, it allows the efficient application to problems where many or even all requests are delivery-only.

3.3 Removal Optimization

The removal heuristics are not as much time-consuming as the insertion heuristics. Still, their efficient implementation is worth consideration.

In the worst removal heuristic, we try to find requests, which increases the most the cost of the solution. Let $\Delta(s, r)$ denotes the difference in costs of solution s when the request r is scheduled and when it is removed from the solution. When we remove a request r , some of the $\Delta(s, r)$ values must be recomputed.

The time complexity of calculating such values has a linear time complexity relative to the route's length. For very long routes, it makes sense to propose constant time recalculation. Suppose we have a route with a length equal to y . This route has actions ζ_e where $e \in \{0, \dots, y - 1\}$. For all these actions, we keep information about the distance the vehicle traveled from its starting node to the location represented by this action. Also, we keep the information about the vehicle's load after performing it for each action. Finally, we keep information about the price of the whole route.

Suppose we have a request for which we want to calculate the value $\Delta(s, r)$. To do so, we only need to calculate the price changes in the route. This request must have a pickup and a delivery action on this route. We denote these actions by ζ_p and ζ_d , where $p, d \in \{0, \dots, y - 1\}$ and $d > p$ hold. Let Δ^p and Δ^d denote the price decreases caused by differences in distances caused by removing the pickup and delivery locations from the route. We have $\Delta^p = price(\zeta_{p-1}, \zeta_p) + price(\zeta_p, \zeta_{p+1}) - price(\zeta_{p-1}, \zeta_{p+1})$. The $price()$ is a function that returns the price of traveling from the first action's location to the second. We have to use the vehicle's current load when calculating these prices. When computing the $price(\zeta_p, \zeta_{p+1})$, we have to consider a higher price per kilometer because the vehicle had a bigger load. ζ_{p-1} represents the action previous to ζ_p and ζ_{p+1} represents the following action. We have $\Delta^d = price(\zeta_{d-1}, \zeta_d) + price(\zeta_d, \zeta_{d+1}) - price(\zeta_{d-1}, \zeta_{d+1})$. In this case, the price per kilometer in $price(\zeta_{d-1}, \zeta_d)$ is higher than in the other two because the vehicle was more loaded.

When we remove the request, we also affect the price of the path from action ζ_{p+1} to ζ_{d-1} since the vehicle has a lower load. We can calculate the price decrease caused by this load change from the precalculated information. We denote the price decrease Δ^{load} . We take the distance traveled from the starting node to the action ζ_{d-1} and subtract it from a distance traveled to action ζ_{p+1} . This way, we can get the distance from ζ_{p+1} to ζ_{d-1} . Using this distance and the weight of the removed request, we can calculate the Δ^{load} . To get the total price, we use $newPrice = oldPrice - \Delta^p - \Delta^d - \Delta^{load}$.

If the route remains empty after removing the request r , we do not need to calculate anything. In this case, the new price would equal zero, and we can return the old price as the cost difference. Furthermore, if the pickup and delivery actions are right behind each other ($\zeta_p = \zeta_{d-1}$), we need to make sure we do not calculate the same price differences twice in Δ^p and Δ^d . If the pickup action is the first in the route, we use the vehicle's starting location to calculate Δ^p . Similarly, if the delivery action is the last in the route, we use the vehicle's ending location to calculate Δ^d .

4 Experimental Evaluation

Our solver was implemented in the programming language Go version go1.15.15 linux/amd64. We used the same values for parameters as described in the original work [10]. For the ALNS running in two stages, we set the number of iterations to 25,000 for each stage. For one-stage ALNS (problems without minimum price only), 25,000 iterations were sufficient.

4.1 Li and Lim Benchmark Instances

For initial experiments, we use Li & Lim [6] benchmark instances¹ generated for the capacitated pickup and delivery with time windows. These data sets are for 50 to 500 requests. Instances are divided into three categories based on the placement of locations, and they are clustered (LC), random (LR), and combined random-clustered (LRC). Instances LC1, LR1, and LRC1 have short scheduling horizons, while LC2, LR2, and LRC2 have longer horizons.

The first set of our experiments aimed to demonstrate the influence of the algorithmic optimizations from Sects. 3.2 and 3.3. We benchmarked six instances (one of each category) for data sets with 100, 400, and 1,000 tasks. The average improvement in runtime was 34.4%, 42.6%, and 51.1%, respectively. We have obtained even more significant improvement for the real-world data (see their description in Sect. 4.2) with 84.9%, 78.3%, and 84.2% runtime improvement for 40, 90, and 140 tasks, respectively, since the existence of the same physical locations allows for additional improvement.

We have selected three instances from each category for further experiments in this section. Experiments on all instances were run 100 times. We have executed our experiments using a grid service provided by MetaCentrum². Each run was limited to 1 CPU core and 1 GB of RAM. In this grid environment, our processes were sharing CPUs with other processes running at the same time. Because of that, the runtimes could vary depending on the current workload.

Comparison to Best Found Solutions. Table 2 (left) shows an overview of how many best solutions we could find depending on the instance size. We can see that for the smallest instances, we were able to find the best solutions for 16 of 18 instances. With the increasing instance size, we could find fewer best solutions. However, it is worth noting that the best solutions published on the Sintef web page were achieved by many runs of various solvers. Therefore, the quality of these solutions is very high. We expect that with more runs of our solver, we will find some of more best solutions. Furthermore, for the instances in which we did not find the best solution, our quality is very close to the best solutions.

Figure 1 (left) shows the difference in the number of vehicles used in our best-found solutions and the best so far found solutions. We have vertically

¹ <https://www.sintef.no/projectweb/top/pdptw/li-lim-benchmark/>.

² <https://metavo.metacentrum.cz/en/>.

Table 2. The number of instances where we were able to find the best so far found solutions (left) and comparison of our best solutions to the original ALNS’s best solutions (right) for 18 instances.

instances size	100	200	400	600	800	1,000
better best	0	9	13	14	15	13
equal best	17	9	4	3	3	4
worse best	1	0	1	1	0	1

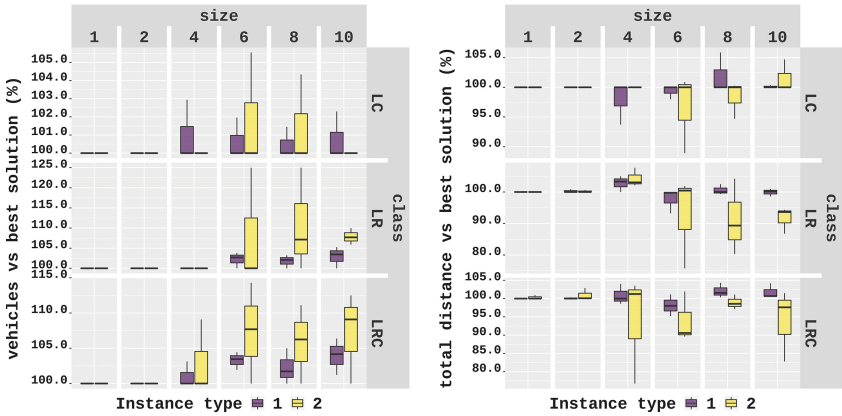


Fig. 1. The differences for vehicles (left) and distances (right) between our best solutions and the best so far found solutions.

categorized the results by the instance category and horizontally by the sizes of the instances (1 to 10 corresponds to 100 to 1,000 tasks). We have also separated them by the length of the schedule horizon (1 represents the shorter horizon, and 2 represents the longer horizon).

We can see in the results that our solver has performed better in the clustered instances than in the random and random clustered. In the clustered instances, the search space is more constrained by the time windows. Also, the number of vehicles used usually corresponds to the number of clusters in these data instances. Therefore, it is easier for the solver to find a solution with the minimum number of vehicles. Not surprisingly, we can see that our solution has performed worse with the increasing size of instances. Our solver has also performed worse for instances with longer schedule horizons (type 2). Because of the longer schedule horizon, it is possible to construct longer routes with fewer vehicles. These solutions are for our solver more challenging to find because they usually require a longer distance.

Figure 1 (right) shows the difference in distances in best solutions obtained by our solver against the best so far found solutions. Our solutions had a lower total distance traveled for the instances with a longer schedule horizon (type 2). This is because our solutions used a higher number of vehicles. Therefore, the solutions could have a lower total distance. It is worth noting that despite the

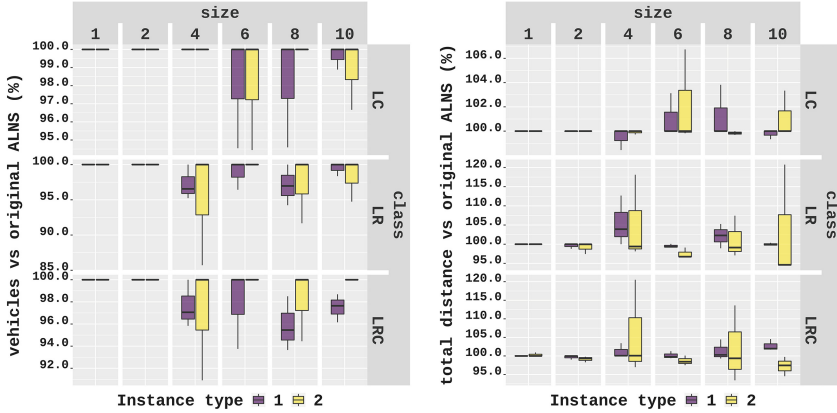


Fig. 2. The differences for vehicles (left) and distances (right) between our best solutions and the original ALNS.

lower distances, our solutions were not better than the best solutions since the goal is to minimize the number of used vehicles, not the total distance.

Comparison to the Original ALNS. Ropke and Pisinger [10] have also performed experiments on the Li & Lim [6] datasets. For instances with 400 tasks and lower, they performed 10 runs. For the other instances, they have performed only 5 runs. They have presented their data about their best solutions, average solutions, and average runtimes.

Table 2 (right) compares our best-found solutions and the best solutions found by the original ALNS implementation. We can see that we have managed to find a better solution in most of the instances. However, this is also because we have performed much more runs than Ropke and Pisinger [10]. Figure 2 compares the quality of our best solutions and the best solutions presented in the original work. We can see that we have very similar results. In many instances, we have managed to find a solution with a lower number of vehicles. Of course, in most cases, this results in a higher total distance.

4.2 Real-World Instances

Comparison with OR-Tools. We have received 12 problem instances with 40–140 requests from company Wereldo. These requests have one or two time windows for pickup and delivery. In each of these instances, all requests have the same real-world pickup location. These instances usually have up to 10 different vehicle models. Each model had different prices per kilometer and capacity. For each of these instances, we have also received results of 10 runs performed by the Wereldo’s original solver, which was implemented using OR-Tools [8]. They have a two-stage approach as well. In the first stage, they also ignore the minimum vehicle prices and use the minimum prices per kilometer, ignoring the vehicle’s

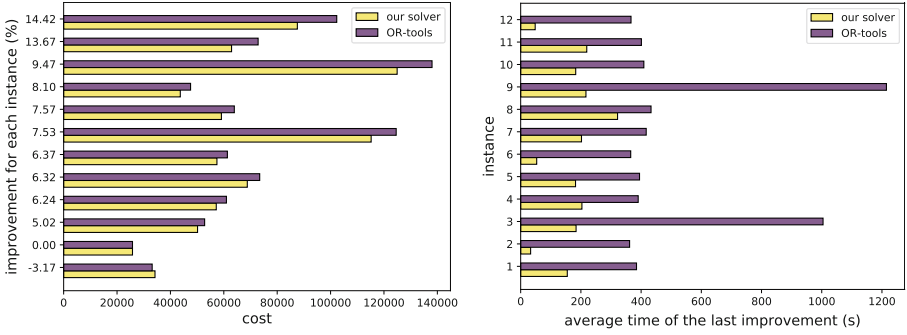


Fig. 3. Comparison of costs (left) and average times (right) of best solutions achieved by our solver and company’s solver in OR-tools.

current load. In these results, we received the costs of the best solutions found in both stages and the times when they were found. However, the OR-Tools did not improve the previous solution in the second stage in any of the runs. Therefore, we will use only the results from the first stage for comparison. We have also received information about the peak memory usage obtained during both stages. The company runs these experiments on the Amazon Web Services³ (AWS). With our solver, we also performed 10 runs for each of these instances on a regular laptop with the Intel Core i7-8550U CPU (1.80GHz).

Figure 3 (left) compares the costs of our best and the company’s solutions. We have found a better solution in 10 out of 12 instances with 5–14% improvement. In one instance, the best solutions were equal, and in the last one, the best company’s solution was slightly better. In this instance, we have noticed that the company’s solver had managed to find a solution with one less vehicle, which led to a solution with a lower cost. Our solver could not find a solution with this lower number of vehicles in any of its runs. In Fig. 3 (right), a comparison of averages of times of the last improvements is provided. The experiments were not executed in the same environment. Still, it is notable that our solver running on a regular laptop was faster. The average memory peak usage provided by the company was 480–570 MB for the first stage and 2.9–4.1 GB in the second stage. Our overall peak memory usage was 100–120 MB in both stages, which is a significant improvement. Note that the high memory usage in the second stage is caused by the fact that the relative vehicle’s price per kilometer is not well optimized in the OR-Tools. It is not even recommended to be used, and the developers have practically abandoned this feature.

Large-Scale Problems. These experiments ran on CentOS8, Intel Xeon Skylake 2.3 GHz, 8 CPUs and 16 GB RAM, and each run was limited to 1 CPU. The Wereldo company provided data with characteristics listed in Fig. 4 (left). Data were separated into 5 single-day instances. While the earlier real-world data sets were delivery-only, now we have requests with pickup and deliveries

³ <https://aws.amazon.com/>.

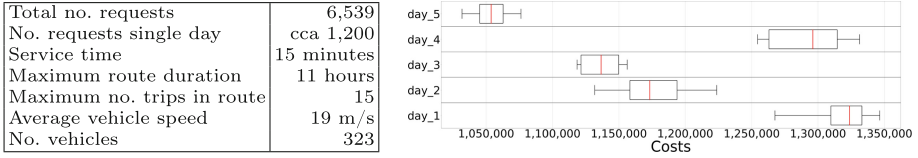


Fig. 4. Characteristics of the dataset and costs of solutions for single day instances.

(85 % of requests being delivery-only). In addition, the vehicle’s minimum prices are not considered, and one-stage ALNS is run only. The vehicle fleet is heterogeneous both in capacities as well as in operational costs. Regarding capacities, a significant portion of the fleet (68.1 %) is composed of large trucks capable of accommodating 66 or 72 pallets and up to 24 tons of load. The remaining 6 vehicle types are very different, with capacities from 4 to 25 pallets. There are 8 cost models of vehicles where 71.8 % of them have cost X per kilometer with others ranging between $0.4X$ and $3.1X$. In Fig. 4, we can see the resulting costs for 10 runs of single-day instances. We achieved average runtimes 52.7 ± 1.7 min.

5 Conclusion

This work considers real-world freight transportation problems with many characteristics. We provided a formal model for a unique combination of constraints and objectives which were necessary for the representation of our problems. We proposed a solver based on the adaptive large neighborhood search, which allows us to solve a wide variety of routing problems efficiently. To achieve that, we have concentrated on the complexity of insertion and removal heuristics which constitutes the heart of the search procedure. We have identified frequent validation steps for insertion heuristics, which can be processed in constant instead of linear time relative to the route length. To achieve that, we have identified crucial information to store and precompute. Similarly, constant time recalculation is also proposed to compute differences between the costs of solutions for removal heuristics. The minimization process is enhanced to handle real-world objectives for heterogeneous vehicles where the minimum vehicle’s prices necessitate two stages of the ALNS.

In the paper, we demonstrate the generic application of the solver on problems with different characteristics. First, we verify results on the standard Li & Lim benchmarks by comparison with the best-found solutions and the original ALNS solver. These benchmarks represent base pickup and delivery problems with time windows. Further real-world problems from the company with 40–140 requests have the same pickup location, time windows for both pickup and location, and contain all features described in our formal model, including the minimum price. Experimental results show better solution quality, faster computation, and significant memory saving compared with the company’s solver in OR-Tools. The last type of problem contains about 85 % delivery-only requests, and its vehicles do not need to consider minimum vehicle prices. Including this

data set allows us to demonstrate less than one-hour runs on large-scale problems with 1,200 requests and more than 300 heterogeneous vehicles. To conclude, the solver is used by company Wereldo in everyday operations, and they have also applied it to various case studies for different customers.

Acknowledgements. Computational resources were supplied by the project “e-Infrastruktura CZ” (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures.

References

1. Braekers, K., Ramaekers, K., Van Nieuwenhuysse, I.: The vehicle routing problem: state of the art classification and review. *Comput. Indus. Eng.* **99**, 300–313 (2016)
2. Caceres-Cruz, J., Arias, P., Guimarans, D., Riera, D., Juan, A.A.: Rich vehicle routing problem: survey. *ACM Comput. Surv.* **47**(2), 1–28 (2014)
3. Eksioğlu, B., Vural, A.V., Reisman, A.: The vehicle routing problem: a taxonomic review. *Comput. Indus. Eng.* **57**(4), 1472–1483 (2009)
4. Elshaer, R., Awad, H.: A taxonomic review of metaheuristic algorithms for solving the vehicle routing problem and its variants. *Comput. Indus. Eng.* **140**, 106242 (2020)
5. Koç, Ç., Bektaş, T., Jabali, O., Laporte, G.: Thirty years of heterogeneous vehicle routing. *Eur. J. Oper. Res.* **249**(1), 1–21 (2016)
6. Li, H., Lim, A.: A metaheuristic for the pickup and delivery problem with time windows. In: *Proceedings 13th IEEE International Conference on Tools with Artificial Intelligence. ICTAI 2001*, pp. 160–167 (2001)
7. Montoya-Torres, J.R., Franco, J.L., Isaza, S.N., Jiménez, H.F., Herazo-Padilla, N.: A literature review on the vehicle routing problem with multiple depots. *Comput. Indus. Eng.* **79**, 115–129 (2015)
8. Perron, L., Furnon, V.: OR-Tools. <https://developers.google.com/optimization/>
9. Potvin, J.Y.: State-of-the art review – evolutionary algorithms for vehicle routing. *INFORMS J. Comput.* **21**(4), 518–548 (2009)
10. Ropke, S., Pisinger, D.: An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transp. Sci.* **40**, 455–472 (2006)
11. Toth, P., Vigo, D.: *Vehicle routing: Problems, methods, and applications*. Society for Industrial and Applied Mathematics (2014)
12. Turkeš, R., Sörensen, K., Hvattum, L.M.: Meta-analysis of metaheuristics: quantifying the effect of adaptiveness in adaptive large neighborhood search. *Eur. J. Oper. Res.* **292**(2), 423–442 (2021)
13. Vidal, T., Laporte, G., Matl, P.: A concise guide to existing and emerging vehicle routing problem variants. *Eur. J. Oper. Res.* **286**(2), 401–416 (2020)