University of Groningen

Design of combinatorial algorithms

Final report

ALNS with a Tabu list

Solving a VRP with electrical charging stations and time windows

Author: Rens Haagsman, Lucas Krol

Student Number: S4154878, S4072103

August 24, 2023

Abstract

In this paper a meta-heuristic has been proposed to solve the LRPCC problem. This meta-heuristic had the ALNS algorithm as a basis, with a Tabu component, which disallowed certain chargers to be used. We found that for larger instances in which optimization of the chargers was needed, the Tabu component yielded a considerable improvement in the solution quality, reducing the objective value up to 10% for the largest instance size. When comparing the solutions of the meta-heuristic to best-known solutions on the same instances, it was found that the meta-heuristic outperformed all but one and often with a considerable margin.

Contents

Introduction

With the rise of online retailers, such as Amazon, the need for an efficient distribution schedule has been ever-growing. Next-day delivery has become standard in most business to customer companies and without a proper routing schedule the corresponding transportation costs can become too large a burden for the profitability of a company.

At the same time, the usage of petrol and gasoline vehicles is being restricted by governments, aiming to reduce emissions as much as possible. For instance, recently the European Union expressed their goal to only allow production of zero-emission vehicles after 2035, which with the current technologies would imply that the electrical vehicle (EV) becomes the standard. However, these vehicles come with their limitations, as they usually have a limited driving range and therefore will need to be recharged on a regular basis. Since this recharging also takes a considerable amount of time, including this into a route planning model will be a key component for a distributor working with EVs. Additionally, charging can only happen at locations which have a charger, so finding the optimal location for these chargers is just as important.

Furthermore, to improve customer satisfaction, many companies allow customers to specify delivery times. Thanks to this feature, customers are able to better plan their day. For instance, for a grocery shop, it allows their deliveries to be at the start of the working day such that it is on stock for their own customers as well. This has consequences for the route planning, as arriving early might simply mean that there is no one to receive the goods, whereas arriving too late might mean that the customer lost business himself, which might result in them choosing a different supplier or asking for a compensation.

The problem of route planning for electrical vehicles with time windows is labelled a Location Routing Problem with Charging at Customers (LRPCC). This problem has, even for small instances, a very large solution space and therefore is considered a NP-hard problem. Therefore, exact solutions are nearly impossible to achieve and one has to depend on meta-heuristics to find good solutions. For example in the paper by Schiffer and Walther (2018), A Large Neighbourhood Search (ALNS), with a dynamic programming and local search element, is employed which works especially well for larger instances.

In this paper, we will aim to devise an algorithm to find solutions to LRPCC problems which performs better than the existing algorithms, with respect to either solution quality or computation time, or both. This algorithm will use ALNS as a basis, but will extend it with a Tabu component: certain locations for a charger will be disallowed. This will reduce the solution space and therefore allow for finding better solutions quicker.

Our algorithm differs significantly from most algorithms in the existing literature, since we work with a situation in which we can have partial recharging, whereas the existing literature mostly focuses on either fully charging) or even battery swapping Yang and Sun (2015). Hence, our problem is a relaxation of those problems and our algorithm should therefore be able to achieve lower costs than implementing those existing algorithms. In Schiffer and Walther (2018) the exact same problem was studied, but Schiffer and Walther (2018) did not include any operator related to the charger location or charge percentages and instead used penalty functions as an alternative. In the end, they then made the solution feasible again. Our algorithm will, however, make the solution feasible again in every iteration. We also include charge and charger location operators, to optimize these during the algorithm itself already.

In the remainder of this paper, we will first look at the problem formulation. This will be followed by a detailed description of the algorithm. Afterwards, we will look at the solution quality of the algorithm and show a sensitivity analysis. We will end with a conclusion and recommendations for further research.

Problem formulation

The LRPCC problem considers one depot (labelled 0) with n customers (labelled 1 up to n). Travel distances between vertices are given as $d_{i,j}$ $(i, j = 0, ..., n)$. There are an unlimited amount of vehicles available at the depot, but using them comes with a cost c_V. The number of vehicles used is denoted by K. All vehicles have a speed of v and have an unlimited vehicle capacity. The costs of driving are denoted per distance unit and are given as c_d.

At every customer, a charger station can be placed at a cost c_i^c. If a charger station is placed at a location, the variable y_i will equal 1, otherwise it will equal 0. Battery charging happens linearly at a rate r per time unit, whereas discharging will happen at a rate b per distance unit. The total capacity of a battery equals B. At the start of the day, the EV will always have a full battery.

Every customer should be visited exactly once. There is an unloading time, s_i, for each customers. For every customer there is a time window, $[e_i, \ell_i]$, in which you can visit them. Here, the e_i is strict, implying that there can be no charging or unloading before this time. Hence, if you do arrive early, you will have to wait till time e_i. The ℓ_i is a soft constraint however and can be violated at a cost of c_t per time unit you are too late. The total time by which the window is violated at location i is denoted by f_i. These time violations are counted both when charging and when unloading. Charging and unloading can happen at the same time.

The objective function is given as follows:

$$\min Z = c_V K + \sum_{i=0}^{n} \sum_{j=0}^{n} c_d d_{ij} x_{ij} + \sum_{i=1}^{n} c_t f_i + \sum_{i=1}^{n} c_i^c y_i \tag{1}$$

where $x_{i,j}$ is a binary variable, equaling 1 if an arc (i, j) is travelled in the solution and 0 otherwise.

Thus, the problem is to minimize this objective function by determining

- the number K of vehicles to use,
- for each location whether or not to add a charging station (y_i),
- the route R_k and schedule for each vehicle $k, k = 1, \ldots, K$,
- the amount of electricity charged by the visiting vehicle at location i, denoted w_i (or equivalently, the charging time w_i/r).

For the reader's convenience we also included a summary of all variables and parameters used in the model, which can be found in Table 1

Table 1: Nomenclature of the model

Nomenclature	**Description**
Parameters	
0	Index depot
n	Number of customers
$d_{i,j}$	Distance between location i and j, $i, j = 0, 1, ..., n$
c_V	Cost per vehicle
K	Number of vehicles used
v	Speed of the vehicle
c_d	Cost of driving, per distance unit
c_i^c	Charger station placement costs for customer i
r	Battery charging rate per time unit
b	Discharge rate per distance unit
B	Total capacity of a battery
s_i	Unloading time for customer i
e_i	Start of the time window for customer i
ℓ_i	End of the time window for customer i
f_i	Time window violation at customer i
Variables	
$x_{i,j}$	Binary variable indicating if arc (i, j) is travelled
y_i	Binary variable indicating if a charger station is placed at customer i

Structure of the algorithm

In this section we propose an ALNS meta-heuristic, with a Tabu component in it, to solve the LRPCC problem. We will first give the psuedo-code for the algorithm as a whole and will then specify underlying heuristics, operators and the Tabu component in more detail.

The basis of our algorithm is explained in Algorithm 1. As input it requires the total run time (in seconds) and the starting temperature. The cooling rate will be determined in the algorithm itself, based on the expected number of iterations (μ_{its}) which can be run within the allotted time. Specifically, the cooling rate is determined in such a way that for every $0.1 \cdot \mu_{its}$ times the solution is accepted, the temperature decreases by a factor $\frac{4}{3}$. We have found an input temperature of 50% of the objective value of the initial solution to work quite well.

Algorithm 1 ALNS with Tabu

Input: t_{run}, T
Construct initial solution s
Run short time local improvement on s
Give all operators an equal weight
$\mu_{it} = g(t_{run}, n)$ ▷ Expected number of iterations
$cr = (\frac{3}{4})^{(0.1\mu_{it})^{-1}}$
$s_{best} = s$
$it = 1$
while $t < t_{run}$ **do**
  Select operators
  $s' = o(s, \text{Tabu})$ ▷ $o(\cdot)$ is a generic (set of) operator
  if $f(s') < f(s_{best})$ **then** ▷ $f(\cdot)$ calculates the objective value
    Run local improvement on s'
    $s_{best} = s'$
    Add 100 to weight $o(\cdot)$
  end if
  if Accept(s', s, T, cr) **then**
    $s = s'$
    Add 1 to weight $o(\cdot)$
    $T = T * cr$
  end if
  if $it \,\%\, \text{round}(\mu_{it}/1000) = 0$ **then** ▷ After 0.1% of the total iterations
    Update Tabu list
  end if
  if Time since last improvement $\geq$ 30% of t_{run} **then**
    Run a large neighborhood destroy operator on s ▷ Big jump
  end if
  if At 45% of μ_{it} **then**
    Reset weights
  end if
  $it = it + 1$
end while
Run local improvement on s_{best}

Weights and probability

The operators are each chosen by a certain probability depending on their weight. All operators start with a weight of 10. If a solution is an improvement of the best solution, we add 100 to this weight. We add such a big amount then because a direct improvement is something that is naturally desirable for our algorithm, so we want to reward operators that accomplish this. Since it is quite easy to make an improvement in the first few iterations, we do not add such a big weight during the first 1000 iterations. If a solution is not an improvement of the best solution, but is accepted by our acceptance criterion, we add a weight of 1 to the operators used in that iteration. If a combination of a destroy and repair operator leads to an accepted/improved solution, we assign weights to both operators, as we can not distinguish which operator lead to the improvement.

As for calculating the probability, we first use the total weight of the remove operators and the total weight of the independent operators (operators that do not need a separate repair operator, like splitting a route in 2), to determine whether that iteration we will select from the pool of remove or the pool of independent operators. Having selected a pool of operators, we then look within that pool to assign probabilities of distinct operators being used. For every operator we set a minimum probability, as we want to make sure that an operator which performs better in the latter stages of the algorithm will still be chosen after bad performance in the first stages of the algorithm. An additional reason to do this, is because some operators might not directly lead to better results but can be important for exploration of the solution space and therefore yield better results later on.

The probability calculation consequently goes as follows: Let w_i be the weight of operator i, where i is in the set of possible operators I. Add up all weights of the possible operators, $v = \sum_{i \in I} w_i$. The minimal probability per operator will be set equal to 0.3 divided by the size of I which we will call j. Thus, thirty percent of the probability will be fixed and the other seventy percent is free to be distributed among the strongest operators. We then find the lowest value of $w_i \in I, i^*$. If $w_{i^*}/v \leq 0.3/j$, we set the probability i is chosen, p_i equal to $0.3/j$ and we scale the other probabilities to this value according to their weight If this is not the case, we simply set $p_i = w_i/v$ for all operators.

Expected number of iterations

The function $g(\cdot)$ calculates the expected number of iterations μ_{it} based on the total run time t_{run}. This function is mainly needed, because of how we calculate our cooling rate. Determining this cooling rate based on the number of iterations instead of setting it equal to a certain value, say 0.9999, has much added value since it generalized the algorithm to also work well for different input values. Furthermore, using the expected number of iterations reduces the need of dummies, where letting everything depend on the total run time would require them. Storing and checking these dummies every iteration would take a toll on the speed of our algorithm, which is naturally something we want to avoid. The formula we use is:

$$g(t, n) = t \cdot (\beta_0 + \beta_1 \cdot n + \beta_2 \cdot n^2)$$

We derived β_0, β_1 and β_2 by running our algorithm for a fixed time for different instance sizes. The instance sizes we used varied between 10 and 150 customers. We then calculated the number of iterations per time unit and fitted a model on the number of iterations per second depending

on instance size. This returned the following values:

- $\beta_0 = 118.270771222$
- $\beta_1 = 0.291625164$
- $\beta_2 = -0.002487726$

Acceptance criterion

The acceptance criterion used in our ALNS algorithm is simulated annealing, as proposed in (Bertsimas and Tsitsiklis, 1993). This says that we accept a new solution (s') with probability 1 if the new solution is better than the old solution (s) and with probability $e^{-(f(s')-f(s))/T(t)}$ if it is worse than the old solution. In here, $f(\cdot)$ is the criterion where you base the comparison on, in our case the objective value, and $T(t)$ the temperature over time. The formula for $T(t)$ we use is:

$$T(t) = \begin{cases} cr \cdot T(t-1) & \text{if accept}(s, s') \\ T(t-1) & \text{elsewhere} \end{cases}$$

Tabu

As for our Tabu implementation, we create an empty Tabu list at the beginning of our algorithm. This Tabu list has a predetermined memory equal to 30% of the number of possible charging locations, n. The Tabu list is then updated after every 0.1% of the total estimated iterations. We update the Tabu list by selecting the $m \in [0, n]$ most expensive charger locations that are currently being used in our solution. Out of these m locations, we randomly select $k \in [0, m]$ locations to add to our Tabu list. Here m and k are hyper parameters, which we let depend on the instance size. m will be equal to the memory size and k will be 20 percent of m. Thus, it will take 5 iterations in which Tabu is applied before the memory is exceeded. If by adding those k locations to our Tabu list the memory is exceeded, we remove the oldest location from that list in order to add the new ones.

Now that we have charger locations that we consider Tabu for our algorithm to use in a solution, we temporarily assign a cost to those locations that is significantly higher than the maximum cost of a charger location in that instance. This will discourage the algorithm from using that location when selecting locations to charge, but if the charger location is necessary to get a feasible solution, it can still be chosen. For example, it could be the case that a location is a large distance away from the locations in the route before and after it, in which case it is necessary to charge at that location in order to not run out of battery mid route.

Big jump

In order to avoid getting stuck in a local optimum, we force the algorithm to make a big jump in the solution space if no improvement has been made for a significant time. This time is set at 30% of the total runtime of the algorithm. In this big jump, we use a random removal operator that removes between 30 percent and 50 percent of the total customers. The repair used consequently is the perturbed greedy insertion, which is our most effective insert operator. The details of perturbed greedy insertion will be explained in the operators section.

Constructive heuristic

To find an initial solution, which can be improved in the ALNS algorithm, a modified Clarke-wright savings algorithm is employed. In this algorithm, we first construct n routes, which go from the depot to the customer and then back. We will then iteratively merge routes, based on the routes for which the most distance can be reduced. However, since we do not only have costs with regards to distances to take into account but also time window violations, we furthermore specify that any route can be at most as long as $dist_{max}$. Here $dist_{max} = 1.2 \cdot \max\{\ell_i : i = 1, ..., n\} \cdot v$, which implies you can't construct routes which will give very large time window violations. A more detailed description can be found in Algorithm 2

Algorithm 2 Modified Clarke-wright savings algorithm

```
Construct n routes which go from the depot to the customer and back
Calculate the savings list {s : s_{i,j} = d_{i,0} + d_{0,j} − d_{i,j} ∀i, j}
Order s decreasingly
while length(s) > 0 do
    Take the first element of s
    if (x_{0,j} == 1 and x_{i,0} == 1) or (x_{0,i} == 1 and x_{j,0} == 1) then
        dist_new = distance of merged route
        if dist_new < dist_max then
            Merge routes
        end if
    end if
    Destroy the first element of s
end while
```

Operators

Destruction operators

- Random removal: Randomly remove m customers from their routes. The operator randomly picks a number between 1 and n, with n the number of customers. It then removes the corresponding customers from their routes. There are three versions of this operator in our algorithm, small, medium and large. Small removes between 1 and 3 customers, medium removes at most 10 percent of the number of customers and large removes between 30 and 50 percent.
- Single point removal: Randomly selects a route. In this route, it selects a random customer. Every customer between the depot and that randomly selected customer is removed from the route.
- Two point removal: Randomly selects a route. In this route, it selects two random customers. Every customer between these customers, including the selected customers, is removed from the route.
- Binary removal: Randomly selects a route and removes all customers in the first half of the route.

- Early removal: Looks at all customers and the time when the truck arrives at the customer. It then finds the three customers that spend the most time waiting before unloading, i.e. the customers where the truck arrives most early. It removes a random customer from these three customers out of its respective route.
- Latest removal: Looks at all customers and the time when the truck leaves the customer. It then calculates the time window violation costs for each customer. It randomly removes one of the three customers that have the highest time window violation costs.
- Perturbed latest removal: Similar to latest removal, but now the time window violation costs are perturbed with random factor between 0.7 and 1.3. It then still finds the three highest perturbed costs and removes one of them.

Repair operators

- Random insert: Randomly reinsert the removed customers into routes. The operator randomly picks a customer from the routes and places a removed customer after the picked customer in their route. If the algorithm picks the depot, it creates a new route that goes from the depot to the removed customer and back. It will keep doing this until all customers are inserted.
- Perturbed greedy insertion: For each missing customer, the algorithm calculates a random perturbation of the distance from the node to the possible insertion position. The node is then inserted at the position that minimizes the criterion, and the solution is updated accordingly. If the node is inserted after the depot, a new route is created. Otherwise, the node is inserted into an existing route.
- Perturbed early insertion: Looks at all nodes and randomly perturbs the time spent waiting before the truck can start unloading, i.e. the amount of earliness. It then places the missing customer before the customer that has the highest perturbed waiting time.

Independent operators

- Random split: Randomly selects a route. Within this route it selects a random customer. The route is then split into two routes at this location.
- Distance split: Randomly selects a route. Within this route, it finds the longest distance between two customers and splits the route into two routes between these customers.
- Exchange charge: Selects a random route that has at least two chargers. For one of the chargers, it then removes either 10 percent of the total battery capacity or the total amount charged, whichever is smaller. Next, the operator tries adding the charge at other chargers in the route. If a charger has too much charge after adding the requested amount, the excess charge is distributed to another charger location in the route. The solution which leads to the lowest objective value is then chosen as the best solution.
- Exchange charge uniform: Works the same as exchange charge, except removes a random amount of charge from the charging location in question. This amount is uniformly sampled between zero and the total amount charged at that location.

- Violation facility placement: Randomly selects two out of the five locations with the largest charge amounts. It then, for both locations, finds the three locations before the location being handled at which charging would be relatively cheapest. This cheapness is calculated by taking the ratio between the placement costs and the total charge that can be charged for free, i.e. during unloading. It then places a charger at that location and sets the charge amount there equal to the amount of free charge, removes this free charge from the selected expensive location and updates the battery levels accordingly.
- Route facility: Selects a random route that has more than three locations excluding the depot and more than 1 charger. It then removes all chargers from that route. Next we use dynamic programming to find a set of charger locations which minimize the placement costs. It first considers the end of the route and assumes you end up with zero charge there. For every node before the depot, it will then consider adding a charger there or not. If the total discharge between two chargers is larger than the maximal battery capacity, the option is infeasible and removed. Additionally, to reduce memory objective values are compared for solutions which place a charger at the same node and only the one with the lowest objective value is kept. After having found the set of charger locations which minimize the placement costs, charge is added to charger locations such that the battery is always positive and we start with a battery of B and end with a battery of zero.

Local improvement

If we arrive at a new best solution, it might be worth it to run a form of local improvement on that solution before accidentally heading in the wrong direction from that solution. The algorithm used for local improvement is similar to the ALNS algorithm and can be seen in algorithm 3. The main difference is the fact that the operators used for local improvement are merely small neighborhood operators, in order to not steer too far away from the best solution. The temperature is set significantly lower than in the ALNS algorithm, to reduce how much the algorithm explores. The temperature we used for example was equal to 0.5% of the best objective value. We also do not include Tabu in local improvement in order to be able to consider all options.

Algorithm 3 Local Improvement

Input: t_{run}, T
Give all operators an equal weight
$\mu_{it} = g(t_{run}, n)$ ▷ Expected number of iterations
$cr = (\frac{3}{4})^{(0.1\mu_{it})^{-1}}$
$s_{best} = s$
$it = 1$
while $t < t_{run}$ **do**
  Select operators
  $s' = o(s, \text{Tabu})$ ▷ $o(\cdot)$ is a generic (set of) operator
  if $f(s') < f(s_{best})$ **then**
    $s_{best} = s'$
    Add 100 to weight $o(\cdot)$
  end if
  if Accept(s', s, T, cr) **then**
    $s = s'$
    Add 1 to weight $d(\cdot), r(\cdot)$
    $T = T * cr$
  end if
  if Time since last improvement $\geq 0.02 \cdot t_{run}$ **then**
    s = s_{best}
  end if
  $it = it + 1$
end while

It is ran for 0.1% of the total runtime t_{run}. The operators used are:

- Destruction heuristics:
  - Small random removal
  - Early removal
  - Latest removal
  - Perturbed latest removal
- Repair heuristics:
  - Random insert
  - Perturbed greedy insertion

  - Perturbed early insertion

- Independent heuristics:
  - Exchange charge
  - Exchange charge uniform
  - Violation facility placement
  - Route facility

Local improvement is also performed at the end of the ALNS algorithm for 5% of the total runtime, to improve upon the best found solution in the ALNS algorithm.

Charge feasibility

Since most of our operators affect the routing and not the charging, after a majority of the iterations we must make our solution feasible with respect to charging. For our initial solution, we did this quite inefficiently, by going through the routes and simply charging to full battery whenever the charge would otherwise run out on the next segment of the route.

This is of course something that we want to avoid when we are further in the algorithm, as this might completely mess up the charging on a route, increasing the objective value for that iteration and possibly disallowing the algorithm to get close to the best solution in that area of the solution space.

In order to work around this, we designed a function which locally make solutions feasible again. For every route in which changes were made, the function iterates through every set of two charging locations in a route. The depot is included as a charging location, so the first segment it goes through would be from the depot to the first customer with a charger. Then for all of those segments, it will try to end up with the total amount of battery that should be in the EV at the next charger according to the old solution. If nothing changed in the segment, it will not edit anything. However, if something is changed, it will then check if within that segment charge can be removed or added to either the charger at the beginning or the charger at the end in order to end up with that exact amount of battery upon leaving for the next segment. If an extra charger is necessary it will place one at the cheapest possible location. By doing this, the function makes sure that except for the parts of the route that have been changed, the charging scheme for that route stays unchanged, keeping the charging in that route near optimal, possibly even improving upon the previous charging situation, as it will solve the segments in such a manner that if possible the charge with which the truck reaches the depot at the end of the route is equal to zero.

Results

Performance over time

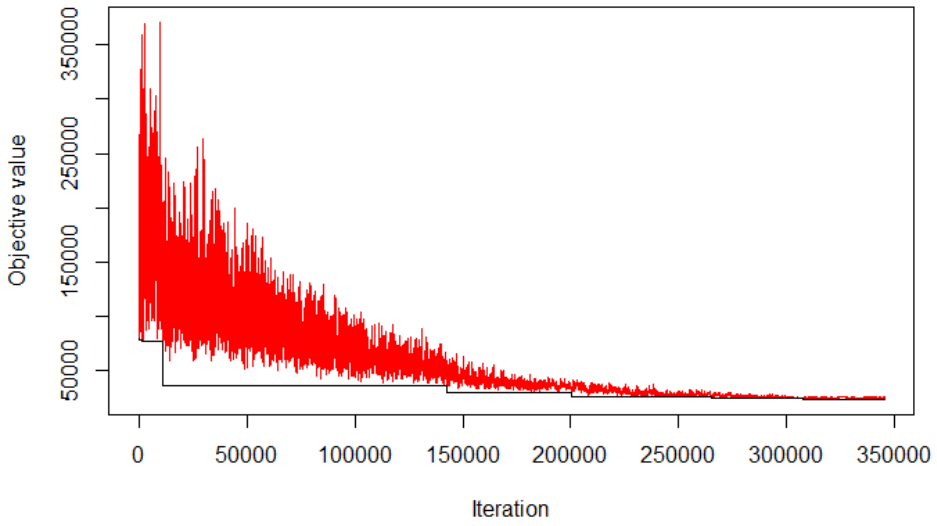

Figure 1: Solution over time

In Figure 1, we see the progress of our algorithm over time. The red line is the current solution the algorithm is working with and the black line is the best found solution so far. This graph was created using a 10 minute run on an instance of size 50. We can clearly see that at the beginning, the algorithm does quite a lot of exploring, some solutions being worse, some being better. However, after a while, this exploration becomes more narrow, and the solution slowly decreases. Towards the end of the run, not many solutions are accepted anymore, yet we still improve our solution a couple of times, i.e. we are finishing up on exploring that solution space and have (nearly) reached the local optimum. Note that this is a short run, so we have not had a drought in improving the solution long enough for a big jump to occur, but this Figure mainly aims to show the performance of our chosen temperature and cooling rate. It is also highly likely that the solution found at this stage is not close to the actual optimal solution, but the graph does nicely show the progress of our algorithm over time.

Tabu

In Table 2 we show the results of running our meta-heuristic with and without the Tabu component, once for every instance size. We specifically show the results for one instance per instance size, since we have a total of 300 instances and reporting on them all would be impractical. From the table we can see that the algorithm at least can significantly improve upon the initial solution and often improves upon it by a factor 3-10. Of course, this also shows the initial solution is of a bad quality, which is not very surprising considering it only takes into account routing and does even that heuristically. However, the results of the algorithm itself do not seem to suffer from it.

Instance size	**Initial solution**	**Min no Tabu**	**Mean Tabu**	**Min Tabu**	**Mean no Tabu**	Δ%
10	12141.11407	4053.808665	4053.808665	4053.808665	4073.8949985	0
20	9235.904295	2942.86094	3040.0332114	2769.501486	3080.0967864	-5.89
30	23220.28215	3753.01878	4120.6888946	3663.041607	4037.3858495	-2.4
40	20474.66183	3812.663032	4385.2745467	3886.850039	4215.5899484	1.95
50	46706.88734	9113.828236	10667.6543181	8890.989068	10071.3941442	-2.45
60	72890.02535	5756.405067	6561.047109	6110.932577	6528.735656	6.16
70	78803.60711	3619.472829	4485.7933546	3864.622481	4670.9214314	6.77
80	72840.00576	9590.395562	10789.6209233	9597.529203	10735.2777372	0.07
90	81058.68709	10981.01484	11587.859244	10043.95447	11667.307969	-8.53
100	94218.11347	11520.34149	14219.878103	12438.40508	13620.624111	7.97
125	194211.3	17761.93101	21368.107866	17700.27453	20693.018388	-0.35
150	197838.3555	25755.33806	27570.756116	25716.52471	29143.986991	-0.15

Table 2: For the first instance of the specific instance size, we show the solutions for Tabu and no Tabu and compare the difference in their minima (Δ%). These results are obtained using 10 runs of 1 hour each

When we compare the results of the algorithm with and without Tabu in Table 2, one does not seem to perform better than the other. The reason for this is that for some instances no chargers are needed, or only a very slim amount, and since the Tabu list only works for the chargers, the Tabu component gives a lot of overhead, to no avail. Therefore, it intuitively makes much more sense to only look at the performance of the Tabu list in instances in which charging is needed. For this reason Table 3 was created, which compares the objective value of the algorithm with and without Tabu, but then only for the instances in which at least $0.1 \cdot n$ chargers are used, in the initial solution (after local improvement).

Instance size	Δ%	**Instance size**	Δ%
10	-0.35	70	-1.46
20	-0.13	80	-4.89
30	-0.23	90	-1.61
40	1.23	100	1.62
50	1.17	125	-2.33
60	-1.20	150	-9.45

Table 3: Average percentage difference in minimum objective value of Tabu compared to no Tabu, aggregated over instance sizes

In Table 3 we have aggregated over the instance size, implying that for every instance size, we took the instances with that instance size and compared the minima, when one was run with Tabu implemented and one without Tabu implemented. The average of this is reported in the table. As one can see, the Tabu variant generally performs significantly better than the no Tabu variant, for the largest instance size even yielding an average reduction of 10% in the minimum objective value. The reason why Tabu performs significantly better there, is because those instances have such large solution spaces and therefore punishing sub-optimal charger locations works much better than with smaller instances, where all configurations could more easily be tried. Therefore, we can conclude that especially for larger instance sizes, the Tabu component is really of added value for finding better objective values, as long as the problem requires significant optimization with respect to charging.

Operator processing times

In Table 4 the processing time of every operator compared to the total processing time can be found. These processing times are found by running an ALNS algorithm on an instance of size 50 with an equal probability of choosing every operator. Of course, the destruction operators do not take a lot of the processing time, which is no surprise because they do not recalculate the objective values nor make the solution feasible again. What also can be seen is that the exchange operators on average are relatively more efficient than the insert operators, which is also not very surprising since they can use specific removal structures which yield more efficient calculations whereas for the insertion and destruction operators the removal structure should be kept general such that it can be used by as many insertion operators as possible.

Operator	**% of total proc. time**
Perturbed early insert	21.5%
Random insert	21.24%
Perturbed greedy insertion	15.6%
Violation facility placer	9.49%
Distance split	7.32%
Random split	5.47%
Route facility placement	4.22%
Exchange charge	2.99%
Exchange charge unif	2.94%
Latest remove	0.98%
Early remove	0.97%
Random removal medium	0.87%
perturbed latest remove	0.87%
Two point removal	0.78%
Route facility removal	0.59%
Binary removal	0.34%
Random removal small	0.3%

Table 4: Percentage of the total processing time used per operator

It is a bit surprising that the processing time of random insertion is so large, whilst it actually employs a very easy insertion criterion (random). This might come from the fact that it can construct entirely new routes (whereas for instance perturbed greedy insertion does not) which yield more calculations than putting it in a new route. Furthermore, the insertion criteria used for other operators are also really easy to calculate so the criterion is actually only a small part of the total calculation time of an operator.

Comparison to best known solutions

In Table 5 we show the improvement our algorithm made on the best known solution for the 20 instances for which this is known. As is evident, our algorithm makes a significant improvement to the best known solutions for almost every instance. Especially for instances of larger sizes (and thus for the instance where it is harder to find solutions close to optimal) our algorithm performs significantly better than other algorithms. The only instance the meta-heuristic did not match or improve was instance 61. The reason for this is that in this instance chargers are very

cheap so therefore it would be optimal to place a charger everywhere and charge only a small amount there (during unloading). Unfortunately, our meta-heuristic does only partially take care of such situations with the violation facility placer, but evidently not well enough to actually improve the best known solution. However, because this is something that would not happen very frequently in real life situations, this flaw is of minimum impact on real-life performance.

Instance	**Δ%**	**Instance**	**Δ%**
1	-0.81%	151	-23.13%
16	0%	166	-25.52%
31	-5.12%	181	-17.12%
46	-7.27%	196	-8.38%
61	1.51%	211	-15.01%
76	-1.95%	226	-7.55%
91	-26.85%	241	-13.44%
106	-26.21%	256	-5.25%
121	-17.34%	271	-12.49%
136	-28.52%	286	-9.84%

Table 5: Objective value of our solution compared to the best known solution of the instance (as of 22/03/2023)

Sensitivity analysis

In this section we will perform a sensitivity analysis on the hyperparameters in our algorithm and we will analyse the performance of our operators. The tables shown will display how much the solutions, on average, differ from the best found solution in that run for that instance size. That is, for every instance size we have 25 problem sets, each differing in their cost parameters. For each of these 25 problem sets we will then find the minimum value found for our different hyperparameters. Then, the relative difference in the found objective values to our minimum value is calculated for all hyperparameters (where the one with the minimum value naturally has a deviation). This is then averaged over the 25 problems sets and repeated for every instance size.

Temperature

To find an initial value for the temperature that works well for all instance sizes, we tried different temperature values. Specifically, as we mentioned before, we let temperature depend on the objective value of the initial solution. This ensures that the starting temperature is invariant to the size of the cost parameters: if everything would be 100 times as expensive, our starting temperature still shows the same acceptance pattern whereas a fixed starting temperature would then suddenly be much less likely to accept solutions. After all, the acceptance probability is given by $e^{-(f(s')-f(s))/T(t)}$, so if the values of $f(s)$ and $f(s')$ increase with a factor hundred and the temperature does not, the acceptance probability decreases significantly.

We then tried temperatures equal to 50%, 100%, 200%, 500% and 1000% of the objective value of the initial solution. The results can be seen in Table 6

Table 6: Temperature relative to the objective value of the initial solution

Size	1/2	1	2	5	10
10	0.27%	0.28%	0.47%	0.2%	0.4%
20	1.25%	0.53%	1.03%	1.44%	1.05%
30	2.01%	1.98%	2.94%	2%	2.57%
40	2.46%	1.57%	2.35%	1.11%	3.37%
50	1.93%	3.79%	3.27%	0.8%	2.26%
60	2.18%	2.32%	1.76%	2.26%	2.38%
70	2.74%	4.18%	7.65%	2.67%	3.94%
80	3.37%	4.04%	6.55%	6.82%	2.81%
90	1.91%	1.6%	4.19%	3.66%	3.42%
100	0.65%	2.97%	2.51%	4.27%	4.72%
125	2.35%	2.67%	3.76%	4.5%	6.98%
150	4.6%	5.21%	7.95%	6.64%	3.57%

As can be seen from Table 6, a temperature equal to 50% of the objective value of the initial solution seems to perform on average the best. Therefore, this value is chosen as the temperature for our results section and for the remainder of this sensitivity analysis.

Cooling rate

We performed a similar analysis for the cooling rate of our model. Our cooling rate is defined as: $cr = (\theta)^{(0.1\mu_{it})^{-1}}$, where θ is a value between 0 and 1. Keeping temperature at 50%, we tried

θ values of $\frac{1}{2}$, $\frac{1}{4}$, $\frac{3}{4}$, $\frac{7}{8}$ and $\frac{15}{16}$.

Size	1/2	1/4	3/4	7/8	15/16
10	1.01%	0.79%	0.25%	0.76%	0.13%
20	1.44%	2.06%	1.13%	1.16%	0.65%
30	1.51%	2.76%	1.16%	1.65%	5.7%
40	1.42%	1.91%	1.67%	2.55%	10.61%
50	1.76%	4.16%	2.13%	0.99%	14.63%
60	2.11%	3.38%	1.53%	5.46%	25.09%
70	1.41%	5.74%	4.42%	8.58%	28.69%
80	2.65%	3.86%	0.73%	7.05%	32.09%
90	1.37%	3.55%	2.03%	9.34%	34.18%
100	2.37%	1.41%	2.79%	16.76%	45.18%
125	4.04%	4.5%	2.03%	29.91%	45.25%
150	1.2%	4.53%	2.95%	23.87%	46.98%

Table 7: Cooling rate

We can see in table 7 that a θ value of $\frac{3}{4}$ seems to perform best on average. Choosing θ too high results in big deviations especially for bigger instance sizes, as a θ of $\frac{15}{16}$ means that you keep a high temperature, so there is a lot of exploration, causing the solution to explode and not find its way back to a desirable solution space. Choosing it too low seems to perform too inconsistent, sometimes being better than when we choose $\frac{3}{4}$ and sometimes being worse. Hence, moving forward we set θ equal to $\frac{3}{4}$.

Tabu memory

We also performed an analysis on the memory of our Tabu list. Naturally, this is instance size dependent, as it is undesirable to only put 2 locations on a Tabu list if there are 150 possible locations and vice versa. The memory we choose is a direct percentage of the total amount of customers, where in order to find the best fraction to use, we tried 5%, 15%, 30%, 40% and 50%. This performance of each fraction can be found in table 8.

Size	0.05	0.15	0.3	0.4	0.5
10	0.84%	0.79%	0.59%	0.77%	0.58%
20	1.24%	1.87%	1.08%	2.26%	1.21%
30	2.24%	1.36%	1.35%	1.91%	1.58%
40	1.89%	2.25%	1.11%	0.91%	1.76%
50	1.87%	4.25%	4.2%	1.89%	2.67%
60	4.43%	3.61%	3.05%	1.64%	2.35%
70	6.85%	1.96%	5.24%	4.98%	2.95%
80	6.02%	5.26%	3.78%	3.08%	4.14%
90	4.24%	2.67%	4.5%	5.13%	3.03%
100	4.59%	3.18%	2.16%	1.15%	3.59%
125	7.95%	6%	3.45%	2.89%	12.81%
150	6.31%	3.64%	2.68%	6.26%	7.98%

Table 8: Tabu

Based on these deviations from the best found solution, we opted for a Tabu memory of 30

percent of the total amount of customers, as consistently for most instance sizes, increasing or decreasing this percentage seems to negatively impact the performance of our algorithm. Note that for the lower instance sizes, this is not as clear, but for larger instance sizes 30 percent is definitely most consistent.

Big Jump frequency

Now as for the frequency we try to make a big jump to escape a local optimum, we do not want to do this too often, as this will stand in the way of exploring a solution space sufficiently or it will not give the algorithm time to work its way down to a good new solution. We also do not want to do it too little, as when we simply get stuck at local optima we want to escape them and have time left over to explore a new area of the solution space. The values we define is the fraction of total processing time that needs to pass without improving the solution before a big jump is made. As for the possible values, we tried values between 0.1 and 0.5 with increments of 0.1. Thus for a value of 0.1, we would make a big jump at most 10 times whilst with a value of 0.5 we would do it at most 2 times.

Size	0.1	0.2	0.3	0.4	0.5
10	0.21%	0.25%	0.55%	0.58%	0.3%
20	0.55%	0.84%	0.8%	1.68%	1.6%
30	1.67%	0.46%	1.9%	3.7%	3.81%
40	1.02%	3%	1.98%	2.53%	2.06%
50	5.6%	3.57%	1.96%	2.49%	4.06%
60	13.01%	3.22%	1.47%	1.55%	3.37%
70	23.46%	3.92%	4.2%	3.57%	2.95%
80	30.15%	2.98%	2.17%	2.39%	2.17%
90	28.53%	4.91%	5.11%	3.42%	5.51%
100	27.62%	2.87%	4.7%	3.13%	4.53%
125	28.78%	6.04%	2.01%	3.19%	5.83%
150	24.78%	4.79%	5.62%	5.49%	3.18%

Table 9: Big jump

The results of this analysis are found in table 9. We find that for large instance sizes, having a ratio of 0.1 is undesirable. This makes sense as for larger instance sizes, the solution space will be relatively bigger and thus jumping too often gives the algorithm not enough time to improve its solution sufficiently. Our chosen frequency of 0.3 performs well at low instance sizes compared to higher values and more consistent performance at high instances for lower values.

Weight subtraction

A noteworthy part of our algorithm is the fact that we only reward operators when they improve their solution, and do not punish them for not improving. In this section we will show why we chose this method, even though it may seem illogical. If we run our algorithm with subtraction of weight upon failing to improve, we find that for most instance sizes, the algorithm performs worse. This can be seen in table 10, where the percentages indicate how much the objective values from the runs with weight subtraction are higher than without weight subtraction.

Size	Δ%	Size	Δ%
10	-0.27%	70	-0.4%
20	0.77%	80	2.53%
30	0.49%	90	2.23%
40	-1.36%	100	6.26%
50	-0.59%	125	5.61%
60	3.25%	150	5.03%

Table 10: Weight subtraction

Especially for high instance sizes this difference is quite significant. For the few instance sizes that weight subtraction does improve the solution, the difference is quite marginal, and does not make a big enough impact for us to opt for weight subtraction in our algorithm considering the effects on large instances.

Operators

We also performed a sensitivity analysis on our operators. For every single operator, we removed it from the list of operators available to our ALNS function, keeping the other factors and operators constant. The percentages that can be seen in table 11 are the average relative differences for all instances sorted on instance size. Here a positive number indicates a higher objective value, implying that the operator has a positive influence on instances of that size.

In the last column of table 11, the average deviation over all instance sizes is reported. Firstly we see that perturbed greedy insertion is a very important operator for our algorithm. Removing it from our list of insertion operators would mean an average 9.18 percent increase in objective value.

Secondly we can see that for a large amount of operators, the average deviation is negative, i.e. the algorithm performs better without that operator. However one should note that for many of those instances, the difference is especially largely negative for the instance size of 150, thus skewing those averages quite a bit. The reason for this is probably the fact that because the runs are only 30 minutes, the large instances do not get enough time to effectively run the algorithm. Since removing an operator means there is a smaller pool of operators to choose from, this results in perturbed greedy insertion and other effective operators being used more often, which also directly improves the solution quality. Thus even though the average differences are negative, we opt not to remove those operators from our algorithm, as we believe that in longer runs these operators would still perform well.

If we look at the operators we introduced ourselves, and do not take the outlier of the largest instance size into account, we find that for Route Facility, the average difference is 1.43 %, i.e. the objective value without Route Facility is 1.43 % higher, implying it is quite an effective operator as the runs are still quite short. Similarly, for Violation Facility Placement, the average difference would be 0.44 %. This difference is a bit smaller than that of Route Facility, but one should take into account that this operator is more effective in the latter stages of the algorithm, as it mainly aims at shaving off small bits of time violation costs.

Size	10	20	30	40	50	60	70	80	90	100	125	150	Average
Small Random Removal	0.59%	1%	-0.17%	0.75%	-0.53%	1.96%	2.53%	1.05%	1.1%	2.21%	7.06%	4.64%	1.85%
Medium Random Removal	-0.14%	0.43%	0.07%	1.25%	-1.97%	0.24%	2.09%	-0.82%	-4.25%	-1.83%	2.87%	-7.12%	-0.77%
Large Random Removal	0.16%	0.22%	0.1%	3.67%	-3.28%	-0.05%	1.92%	-1.23%	-4.9%	-1.45%	0.86%	-0.37%	-0.36%
Random Insert	0%	0.03%	1.09%	1.21%	0.46%	1.72%	3.05%	0.13%	-3.16%	1.25%	3.94%	-6.61%	0.26%
Perturbed Greedy Insertion	0.03%	0.11%	2.92%	3.54%	5.52%	11.75%	9.86%	8.2%	10.83%	14.59%	17.04%	25.67%	9.18%
Single Point Removal	-0.68%	0.98%	0.14%	1.23%	-0.69%	2.19%	0.76%	-1.26%	-3.79%	-0.05%	3.16%	-7.31%	-0.44%
Two Point Removal	-0.05%	1.69%	1.24%	1.39%	0.01%	3.66%	4.61%	2.33%	0.36%	2.6%	4.01%	-3.25%	1.55%
Binary Removal	0.12%	-0.17%	0.24%	-0.67%	-0.78%	1.19%	1.38%	0.31%	-0.77%	7.47%	2.31%	-7.07%	0.30%
Random Split	-0.23%	0.45%	1.07%	0.19%	-0.56%	0.34%	2.57%	-0.19%	-2.54%	-1.75%	1.53%	-6.75%	-0.49%
Distance Split	0.02%	1.4%	-0.45%	-0.75%	-2.83%	0.94%	0.93%	-0.02%	-2.13%	1.23%	0.87%	-0.2%	-0.08%
Early Remove	-0.17%	0.64%	0.74%	-0.21%	0.29%	0.73%	2.03%	-1.12%	-1.09%	-3.73%	1.65%	-2.48%	-0.23%
Latest Remove	0.1%	0.61%	3.31%	1.75%	-2.11%	1.45%	2.96%	-0.59%	-2.51%	-0.21%	0.33%	-5.43%	-0.03%
Perturbed Latest Remove	0.17%	0.75%	-1.01%	-2.52%	-2.66%	-1.16%	2.26%	-0.32%	-3.41%	-1.12%	-3.12%	-4.53%	-1.39%
Perturbed Early Insert	0.24%	1.09%	0.3%	0.57%	-2.88%	-1.52%	3.67%	-0.93%	-2.78%	-0.44%	3.66%	-4.49%	-0.30%
Route Facility	0.28%	2.74%	1.94%	1.42%	-1.1%	1.61%	1.09%	-0.12%	2.77%	1.57%	3.51%	-4.73%	0.92%
Exchange Charge	0.11%	-0.14%	-0.34%	-0.82%	-0.24%	0.27%	1.18%	0.78%	-2.62%	0.31%	4.13%	-3.4%	-0.07%
Exchange Charge Uniform	0.79%	0.12%	0.34%	-0.18%	-1.25%	-0.33%	0.68%	1.02%	-1.07%	-0.16%	6.57%	-5.04%	0.12%
Violation Facility Placement	0.67%	1.61%	0.11%	0.73%	-0.36%	0.65%	0.95%	0.3%	-1.34%	-0.68%	2.2%	-5.94%	-0.09%

Table 11: Operators

Conclusion

This paper has focused on solving the LRPCC problem by the means of a meta-heuristic. The LRPCC problem is the problem of finding routes and charging schemes for an electrical vehicle routing problem. Additionally, the vehicles have a certain time-window in which they can arrive and leave, where arriving too early means they will have to wait and leaving too late comes at a cost.

To solve this LRPCC problem a meta-heuristic has been proposed. This meta-heuristic the ALNS algorithm as a basis, but within this a Tabu list was implemented. This Tabu list disallowed certain charger locations to appear in a solution, unless it was specifically needed to make the solution feasible. Furthermore, the meta-heuristic contains a "Big jump" component, which removes a large part of the current solution when it is stuck in a local optimum.

A multitude of new operators has been implemented. The route facility placement operator removed all the chargers in a route and then reinserted them on the cheapest location in the route, given that a suitable charging scheme could be found for this. To do so, dynamic programming is used, with a time-efficiency that is competitive to other operators. Moreover, the violation facility placement operator was also proposed and used, which puts new chargers at locations in such a way that the most use is made out of battery charge that is charged when unloading at a location.

Then, the added value of the Tabu component to the solution quality of the meta-heuristic was investigated. We found that the Tabu component was of no added value when a solution barely required any chargers or when the solution space was small. However, for larger instance in which optimization of the chargers was needed, the Tabu component yielded a considerable improvement in the solution quality, reducing the objective value up to 10% for the largest instance size. Furthermore, we have seen that our new operators on average give a considerable improvement to the solution quality and are relatively efficient.

When comparing the solutions of the meta-heuristic to best-known solutions on the same instances, it was found that the meta-heuristic outperformed all but one and often with a considerable margin.

Discussion

It should be noted that the hyper parameters were determined in the sensitivity analysis by running every instance for 30 minutes, whereas for the actual results we used longer runs. This means that hyperparameters which worked well in the sensitivity analysis might not actually work as well in the actual results. However, most of the hyperparameters where initialized in such a way that they worked well independent of the run time, parameters and instance size. Furthermore, the hyperparameters for the local improvement algorithm were not optimized separately, so for future research this would definitely be something to take into account as well.

Furthermore, the current algorithm lacks an advanced termination criterion, with the current criterion only specifying that we stop after the total time has elapsed. This could be improved by saying that if for a certain time no improvement was made, the algorithm should stop. However, this parameter should be chosen carefully, as the algorithm also make big jumps to escape local optima and this should likely be done a few times before terminating the algorithm.

To obtain better results, we would urge other researchers to focus on including more operators to algorithm, specifically a random charger placement operator. This was something the current algorithm lacked, leading to relatively bad performance on instances where buying many chargers would be optimal.

Lastly, to get an idea of how well a solution performs, it would also be interesting to get a lower bound for the objective value. This could for instance be done quite easily by solving the underlying VRP without time windows and charging exactly, for instance sizes where this is possible. Otherwise, a Lagrangian relaxation of other constraints could also be used.

References

Bertsimas, Dimitris and John Tsitsiklis (1993). Simulated annealing. *Statistical Science 8*(1).

Schiffer, Maximilian and Grit Walther (2018). An adaptive large neighborhood search for the location-routing problem with intra-route facilities. *Transportation Science 52*(2), 331–352.

Yang, Jun and Hao Sun (2015). Battery swap station location-routing problem with capacitated electric vehicles. *Computers & Operations Research 55*, 217–232.

Contribution

	Lucas	Rens
Software coding: operators	50 %	50 %
Verification and validation of the software code	50 %	50 %
Design and execution of experiments	50 %	50 %
Paper writing: description of the heuristic	50%	50 %
Paper writing: all other chapters	50 %	50 %

Table 12: Contributions per person