OPTIMAL ROUTING AND CHARGING OF ELECTRIC RIDE-POOLING VEHICLES IN URBAN NETWORKS

Non confidential

eCAL *energy, controls & applications lab*

Advisor : Scott Moura

August 28, 2016

Léa Nicolas,
Promotion X 2013
MAP 593 : *Automatic and Operations Research*
Organizer : Pr. Stéphane Gaubert

Je soussigné Léa NICOLAS certifie sur l'honneur :

- Que les résultats décrits dans ce rapport sont l'aboutissement de mon travail.

- Que je suis l'auteur de ce rapport.
- Que je n'ai pas utilisé des sources ou résultats tiers sans clairement les citer et les référencer selon les règles bibliographiques préconisées.

Je déclare que ce travail ne peut être suspecté de plagiat.

Date : 28 août 2016

Signature : *Léa Nicolas*

ABSTRACT

In this project, we study an Electric Vehicle Routing Problem with Pick-ups and Deliveries, Time Windows, and Recharging Stations on New York City Taxicab data.
In order to solve this problem, we divide the problems into three phases :

1. grouping similar customer requests by identifying geographic zones and time slots
2. determine groups of passengers to be transported together
3. complete the vehicle itinerary between these groups of passengers

The first phase uses the clustering method *k-means* on the locations of pick-ups and deliveries of New York City taxicabs in january 2013. The second phase uses exact optimization methods, while the third uses meta-heuristics methods.

RÉSUMÉ

Nous étudions ici le problème de tournées de véhicules électriques avec contraintes de temps, stations de rechargement, et passagers à récupérer puis déposer, appliqué aux données des taxis de New York City.
Notre solution se divise en trois étapes :

1. regrouper les requêtes similaires en identifiant des zones géographiques et des créneaux horaires
2. déterminer les groupes de passagers à transporter ensemble
3. compléter le trajet des véhicules entre ces groupes de passagers

La première étape utilise la méthode de partitionnement de données des *k-moyennes*. La seconde étape utilise des méthodes de résolution exactes de problème d'optimisation, tandis que la troisième utilise des méthodes méta-heuristiques.

CONTENTS

1 INTRODUCTION

We live in a world that is increasingly urban-centric: the United Nations estimates that by 2050, 66% of the world's population will live in cities, up from 54% today and 30% in 1950. This emphasis on living in cities poses a series of challenges to civic planners and governments, amongst which we have an "urban mobility" class of problems that need to be addressed. The problem is not only one of time-loss and urban space degradation. It is also an important environmental concern that is central to human health and well-being. According to the World Health Organization (WHO), road transport in cities contributes up to 70% of fine particular matter emissions and up to 90% of carbon monoxide and nitrogen oxide emissions; these being the three most critical air pollutants to human and animal respiratory function.
There are two ways to address the problem. On the one hand we can change the way transport is powered needs to change by shifting from Internal Combustion Engine Vehicles to Plug in Hybrid Electric Vehicles and EVs (Battery Electric Vehicles). This would allow for emissions to be concentrated in power plants outside urban areas – thereby making them easier to control. Important inroads in EV motor and battery technology are making EV adoption increasingly viable. Vehicles like the Nissan Leaf, with more than 80 miles of autonomy and the price points of a family saloon, are opening EV use to broad consumer markets. Moreover, Tesla's pioneering Model S, with its 250 miles of autonomy, is making EVs viable for long-range travel in addition to day-to-day commuting. Another important solution to urban mobility issues, which complements the introduction of EVs, is the development of new mobility channels and finding alternatives to private vehicle ownership. In the past five years, the integration of geo-localization technology into portable communication devices has made it possible to connect transport users and providers in such a way as to create alternatives, among which ride-pooling stands out.
This aims at understanding how electric vehicles can be integrated into the increasingly complex urban mobility framework that these new transport modes have created. Adopting EVs requires an understanding of how they should to be dispatched and routed to ensure that their range and charging requirements do not hinder their use – and exactly this understanding is the focus of this research endeavour.

Schneider et al. (2012) [13] were the first to study electric vehicle routing. Their interest was last-mile delivery, so their problem is an electric vehicle routing problem with time windows and recharging stations.
Here, our interest is the use of electric vehicles for people's transportation in urban areas, and in particular, dial-a-ride transportation. The subsequent problem is the capacitated electric vehicle routing problem with pick-ups and deliveries, time windows and recharging stations (E-VRPTW), which incorporates the possibility of recharging at any of the available stations with recharging times depending on the charge level when arriving at the station, and considers passenger capacity constraints on vehicles as well as customer time windows. E-VRPPDPTW aims at minimizing the number of employed vehicles and total traveled distance. We aim to apply the work to NYC taxicab data.
The other study about E-VRPPDPTW is Barco et al. [2], but their case study, an airport shuttle service scenario including only 6 nodes, is significantly smaller than ours.

The associated optimization problem is the following :

Graph parameters

- s source node
- t end node
- $P = \{1, ..., n\}$ set of pick-up vertices
- $D = \{n+1, ..., 2n\}$ set of delivery vertices : each pick-up i is associated with the delivery $i+n$
- $F = \{2n+1, ..., N+2n\}$ set of charging vertices

- V total set of vertices
- $t_{i,j}$ travel time for $i \rightarrow j$
- $d_{i,j}$ distance of travel $i \rightarrow j$
- K discharging rate
- $B_{i,min}$ earliest time to begin service at vertex i
- $B_{i,max}$ latest time to begin service at vertex i
- q_i demand/supply at vertex i
- P_i charging rate at vertex i
- Q_{max} maximum occupancy
- SOC_{min} minimum state of charge
- SOC_{max} maximum state of charge

Decision variables

- $$x_{i,j} = \begin{cases} 1 & \text{if } (i,j) \text{ is used} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$
- B_i beginning of service at vertex i
- L_i service time at vertex i
- Q_i number of passengers when arriving at vertex i
- SOC_i State of charge when arriving at vertex i

Constraints

- each pick-up or delivery node must have exactly one trip leaving away from it
$$\forall i \in P \cup D, \sum_{j, i \neq j} x_{i,j} = 1 \tag{2}$$
- charging vertices must not be visited more than once (dummy vertices are created in order to make several visits to charging stations possible)
$$\forall i \in F, \sum_{j \in V \cup \{t\}, i \neq j} x_{i,j} \leq 1 \tag{3}$$
- flow conservation : at each vertex, the number of incoming arcs is equal to the number of outgoing arcs
$$\forall j \in V, \sum_{j} x_{i,j} - \sum_{k} x_{k,i} = 0 \tag{4}$$
- precedence constraint : pick-ups precede the associated delivery
$$i \in P, B_i \leq B_{i+n} \tag{5}$$

- Time window
$$B_{i,\min} \leq B_i \leq B_{i,\max} \tag{6}$$
- Maximum load capacity
$$0 \leq Q_i \leq Q_{max} \tag{7}$$
- state of charge constraints
$$SOC_{\min} \leq SOC_i \leq SOC_{\max} \tag{8}$$
- Dynamics of time
$$\forall i, j, (B_i + L_i + t_{i,j}) - B_j \leq M(1 - x_{i,j}) \tag{9}$$
- Dynamics of passenger load
$$\forall i, j, (Q_i - Q_j) + q_i \leq M(1 - x_{i,j}) \tag{10}$$
- Dynamics of state of charge
$$\forall i, j, SOC_j - (SOC_i + P_i L_i - K d_{i,j}) \leq M(1 - x_{i,j}) \tag{11}$$

Objective
$$\sum_i \sum_j c_{i,j} x_{i,j} \tag{12}$$

With our problem, we are confronted to two difficulties:

1. Complexity of the subsequent problem (Mixed-Integer Linear Program solvable for small instances only)
2. Size of the instances to solve (485 000 taxicabs trips per day in New York City on average)

1.1 Classification of the variations of Vehicle Routing Problems

In order not to get confused in the literature review, let's recap clearly the different routing problems we will mention:

- Vehicle Routing Problem (VRP) is the basic problem from which all of the following are variants: What is the optimal set of routes for a fleet of vehicles to traverse in order to visit a given set of customers? The vehicle routing problem is an extension of the Traveling Salesman Problem, and is, as such, an *NP-hard problem*, as well as all of its variants.
- Vehicle Routing Problem with pick-up and deliveries (VRPPDP): this problem considers passengers or goods being picked up and then dropped off by the vehicle. It adds the time variable, because of the precedence constraint: pick-ups have to be visited before drop-offs.
- Vehicle Routing Problem with time Windows: nodes can only be visited during a certain time window. The time window constraint bounds the time variable at each node between a lower and upper bound.
- Capacitated VRP: the number of goods/passengers that can be transported is limited. This problem adds the load variable.
- Electric Vehicle Routing Problem with recharging stations (E-VRP). It adds the constraint of keeping the state of charge between certain values, and the charging of the vehicle is proportional to the amount of time spent at the charging station. Those constraints are formally identical to the time window constraints (by replacing waiting time by charging time).
- Green VRP: VRP with Alternative Fuel (the stations are rare and the vehicle range is shorter than gas vehicles). The difference with electric vehicles car is the recharging time at a station is fixed, and does not depend on the remaining state of charge.

1.2 How to solve a Mixed Integer Linear Problem ? Exact algorithms review

Our problem falls into the category of Mixed Integer Linear programming problems, which canonical form is

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \quad & c^T x && (13)\\ \text{subject to} \quad & Ax \leq b && (14)\\ & x \geq 0 && (15)\\ & \exists I \subset \{1,\dots,n\}; \forall i \in I, x_i \in \mathbb{Z} && (16)\\ & \exists J \subset \{1,\dots,n\}; \forall i \in J, x_i \in \{0,1\} && (17)\\ & && (18) \end{aligned}$$

The most straight-forward way to pass through this complexity is relaxing the integer variables to continuous variables, solve the problem, and then round to the nearest collection of integer variables that is a feasible solution of the problem instance. An optimal solution of the LP relaxation of the MILP yields a *lower bound* on the optimal objective function value of the MILP (assuming minimization, of course), and a feasible solution of the MILP yields an *upper bound* on the optimal objective function value of the MILP.
Here we present the basic techniques used by solvers to solve MILP. They all rely on linear relaxation.

1.2.1 • Cutting plane methods

1. Solve the linear relaxation of the given integer program.
   The theory of Linear Programming dictates that under mild assumptions (if the linear program has an optimal solution, and if the feasible region does not contain a line), one can always find an extreme point or a corner point that is optimal.

2. Test the solution for being an integer solution.
   If it is not, there is guaranteed to exist a linear inequality that separates the optimum from the convex hull of the true feasible set.

3. Find such an inequality is the *separation problem*, and such an inequality is a *cut*

4. Add this cut to the relaxed linear program. Then, the current non-integer solution is no longer feasible to the relaxation

5. Repeat this process until an optimal integer solution is found.

1.2.2 • Branch and bound method and variants

1. Solve the LP relaxation P_0

2. Pick some variable x that is restricted to be integer, but whose value in the LP relaxation is fractional $x^* = r$.

3. Exclude this value by, in turn, imposing the restrictions $x \leq \lfloor r \rfloor$ and $x \geq \lceil r \rceil$. We have two new MIPs P_1 and P_2.
   The variable x is then called a *branching variable*, and we are said to have *branched on* x. It is clear that if we can compute optimal solutions for each of P_1 and P_2, then we can take the better of these two solutions and it will be optimal to the original problem, P_0.

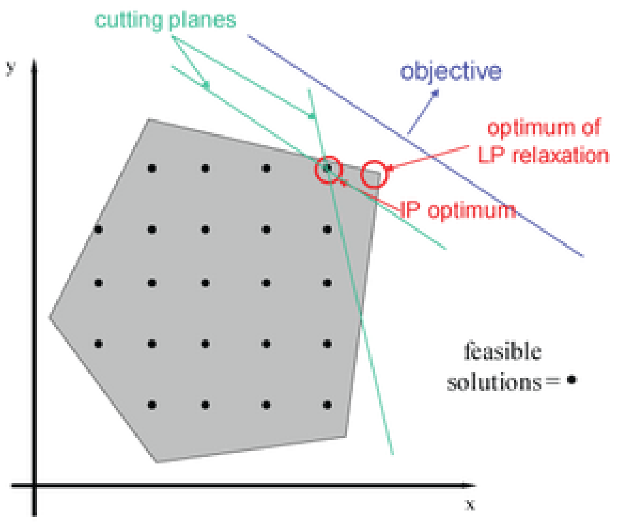

Figure 1: Cutting planes method

4. Apply this process to P_1 and P_2.
   In doing so, we generate what is called a *search tree.* The MIPs generated by the search procedure are called the nodes of the tree, with P_0 designated as the root node. The leaves of the tree are all the nodes from which we have not yet branched.
   If it happens that all of the integrality restrictions in the original MIP are satisfied in the solution at a node, then we know we have found a *feasible* solution to the original MIP. There are two important steps that we then take.

   (a) Designate this node as *fathomed.* It is not anymore necessary to branch on this node. It is a *permanent leaf of the search tree.*

   (b) Denote the best integer solution found at any point in the search as the *incumbent.* At the start of the search, we have no incumbent. If the integer feasible solution that we have just found has a better objective function value than the current incumbent (or if we have no incumbent), then we record this solution as the new incumbent, along with its objective function value.

There are two other possibilities that can lead to a node being fathomed :

- The branch that led to the current node added a restriction that made the LP relaxation infeasible. Obviously if this node contains no feasible solution to the LP relaxation, then it contains no integer feasible solution

- An optimal relaxation solution is found, but its objective value is bigger than that of the current incumbent. Clearly this node cannot yield a better integral solution and again can be fathomed.

Best bound and gap Once we have an incumbent, the objective value for this incumbent, assuming the original MIP is a minimization problem, is a valid upper bound on the optimal solution of the given MIP. That is, we know that we will never have to accept an integer solution of value higher than this value. Somewhat less obvious is that, at any time during the branch-and-bound search we also have a valid lower bound, sometimes call the *best bound.* This bound is obtained by taking the minimum of the optimal objective values of all of the current leaf nodes. Finally, the difference between the current upper and lower bounds is known as the *gap.* When the gap is zero we have demonstrated optimality.

Solvers used In order to solve the problem, I used the Gurobi solver, which one of the best implementations (with CPLEX) of branch-and-bound and branch-and-cut algorithms (we call branch-and-cut algorithms the methods who combine the two previous methods). An analysis of commercial and free and open source solvers for linear optimization problems, justifying this choice, can be found in [11].
The largest instances solved with this method contain 12 requests.

1.3 Literature review

We noticed earlier that time window constraints are formally the same as state of charge constraints, if we consider every point as a recharging station. Thus, my interest was particularly focused on capacitated vehicle routing problems with time windows and pick-ups and deliveries solutions. Indeed there is very few existing literature concerning E-VRP, whereas there has been intensive research on CVRPPDPTW, and papers which recapitulate the different methods.

We can note, however, that state of charge constraints are less coercive than time constraints, and thus make the problem less easy to solve. Indeed, Solomon has shown in [15] that wide time windows are significantly less easy to solve than short time windows (which yield fewer feasible solutions). Thus, we can expect that the algorithms that have been showed to work on very large instances may not work for same-size instances when we consider state of charge constraints.

Algorithms fall into one of the three following categories :

1. Exact solution
2. Heuristic solution : after a construction phase, we use some local searches in a descent way (i.e. the value of the objective function at current iteration step is better than those at previous steps)
   The final result is usually observed as local optimum and is sensitive to the initial solution.
3. Meta-heuristic solutions, on the contrary, can temporarily accept worse solutions during the optimization procedure. Thus, it is possible to drive the search out of local optima.

For exact methods the objective is to minimize the total distance traveled. For heuristics the primary objective is to minimize the number of vehicles used and the secondary to minimize the total distance traveled. This is why we should be more interested in the *heuristics methods.*

1.3.1 • Heuristics methods

Column-generation The general idea is to consider only a subset of variables: only those that have the potential to improve the objective function, which are found through a sub-problem.

Clustering methods The cluster-first, routing-second consists in first creating clusters that will be served by the same vehicle, and then computing a Traveling Salesman Problem on the clusters.
It was first introduced by Bodin et al. in [3]. Bodin and Sexton in [4] construct clusters before applying to each cluster a single-vehicle algorithm and making swaps between the clusters. Results are presented on two instances extracted from a Baltimore data base and containing approximately 85 users each.
Borndorfer et al. in [5] construct in the first phase, a large set of good clusters (complete enumeration) and then, a set partitioning problem is then solved to select a subset of clusters serving each user exactly once. In the second phase, feasible routes are enumerated by combining clusters and a second set partitioning problem is solved to select the best set of routes covering each cluster exactly once. Both set partitioning problems are solved by a branch-and-cut algorithm. On real-life instances, the algorithm cannot always be run to completion so that it must stop prematurely with the best known solution. It was applied to instances including between 859 and 1771 transportation requests per day in Berlin.
Cluster-first, route-second, has also been used for green vehicle routing by Erdogan and Miller-Hooks in [9]

where customers in a single cluster are served with a single vehicle and clusters are formed such that vehicle capacity limitations are not exceeded.

Mini-cluster methods To overcome the difficulty arising from the dispersion of the two locations (pick-up and drop-off) represented by each customer, mini-clusters have been introduced ([8]).
"Mini-clusters" of users are groups of users to be served within the same area at approximately the same time. These mini-clusters are then optimally combined to form feasible vehicle routes, using a column generation technique.
The authors have successfully solved instances derived from real-life data from three Canadian cities: Montreal, Sherbrooke and Toronto. Instances with up to 2545 users are easily solved, while *larger instances require the use of a spatial and temporal decomposition technique.*

1.3.2 • Meta-heuristics methods

Concerning meta-heuristics methods, a central concept in most successful heuristics for the VRPTW is that of *local search.* Local search algorithms are based on *neighborhoods.*
Let $\mathcal{S}$ be the set of feasible solutions to a given VRPTW instance, and let $c : \mathcal{S} \rightarrow \mathbb{R}$ be a function that maps from a solution to the cost to this solution. The set $\mathcal{S}$ is finite, but it is often extremely large. Since the VRPTW is a minimization problem, our goal is to find a solution $s^* \in \mathcal{S}$ for which $c(s^*) \leq c(s), \forall s \in \mathcal{S}$. However, with heuristics, we are willing to settle for a solution that might be slightly inferior to $c(s^*)$.
Let $\mathcal{P}(\mathcal{S})$ be the set of subsets of solutions in $\mathcal{S}$. We define a *neighborhood function* as a function $N : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{S})$ that maps from a solution s to a subset of solutions $N(s)$. This subset is called the *neighborhood of s.* A solution s is said to be *locally optimal* with respect to a neighborhood $N(s)$ if $c(s) \leq c(s'), \forall s' \in N(s)$. With these definitions, we can describe a steepest descent algorithm. The algorithm takes an initial solution s as input. At each iteration, it finds the best solution s' in the neighborhood $N(s)$ of the current solution s. If s' is better than s, then s' replaces s as the current solution. The algorithm is called a steepest descent algorithm because it always chooses the best solution in the current neighborhood.

```
**Result:** $s$
input: initial solution $s \in \mathcal{S}$ ;
$s' = s$ **repeat**
    $s = s'$;
    $s' \in \mathrm{argmin}_{s'' \in N(s)}$ ;
**until** $c(s') \geq c(s)$;
```

Algorithm 1: Steepest descent algorithm

In general, larger neighborhoods lead to solutions of better quality when the neighborhoods are used, for example, in a steepest descent algorithm. The drawback of larger neighborhoods is that the evaluation of all solutions is more time consuming in a large neighborhood compared to a smaller one unless clever algorithmic ideas can be used to speed up the search.

Traditional neighborhoods Traditional neighborhoods can be divided into two categories: *intra-route* and *inter-route.* The neighborhoods in the former category contain solutions in which a single route is changed with respect to the reference solution, whereas, in the second category, they contain solutions obtained by moving customers between two or more routes.

Large neighborhoods Methods for searching large neighborhoods can be divided in two categories: exact and heuristic evaluation methods. An exact evaluation should always identify the best solution within the neighborhood, while a heuristic evaluation may well miss it. In the VRPTW literature, the most common large neighborhoods are encountered in the *Large Neighborhood Search* (LNS) framework put forward by Shaw [14]. The key elements in LNS are the *destroy* and *repair* operators. A destroy operator partially disintegrates a solution, while a repair operator reconstructs a complete solution starting from a partial solution. The two operators are used repeatedly, as illustrated in the pseudo-code of Algorithm 2. In this pseudo-code x is the current solution, x^* is the best solution observed during the search, and x' is a temporary solution. In line 6, a test determines whether the new solution x' should be accepted as the current solution. In the most simple case only improving solutions are accepted.

```
Result: $x^*$
input: initial solution $x$ ;
$x^* = x$ ;
while stop criterion not met do
    $x' = repair(destroy(x))$ ;
    if $accept(x, x')$ then
        $x = x'$
    end
    if $c(x') < c(x^*)$ then
        $x^* = x'$
    end
end
```

Algorithm 2: Large Neighborhood Search

2 PROBLEM RESOLUTION

2.1 Methodoly used

1. Division of New York City into zones
2. Division of time into time slots
3. Association of requests with pick-up zone, drop-off zone and time slot
4. Creating mini-clusters from this grouping
5. Solve an electric vehicle routing problem with recharging stations and time windows formulation between the mini-clusters

Mini-clusters are objects with the following characteristics:

1. a mini-cluster is an ordered set of nodes
2. the way the set of nodes is ordered is the shortest path between the nodes with respect to precedence and time window constraints
3. the vehicle is empty when it enters and when it leaves the mini-cluster, but is never empty in between
4. the passenger capacity is never exceeded

The mini-cluster generation phase takes into account all the local constraints (time window constraints, capacity constraints, coupling and precedence constraints). The second phase is the itinerary creation. It considers the global constraints (state of charge constraints), it assembles the set of mini-clusters into itineraries, taking into consideration the remaining time window constraints in between the mini-clusters. This phase becomes a multiple traveling salesman problem with time window and electric battery. In the mini-clustering phase, we relax the state of charge constraints. Thus, we assume that the mini-cluster trip is too short to exceed the state of charge constraints, and that the car is not allowed to charge when transporting passengers. This is realistic, as the range of today's electric vehicle (at least 70 miles) is such that an electric taxicab should charge only

once or twice during a regular taxi day.
We also make a relaxation on time windows, in the E-VRPTW phase, we define the cluster's time windows as:

$$[\min_{l\in\{1,\ldots,L\}} a_{i_l} - \sum_{l'=0}^{l} t_{i_{l'},i_{l'+1}}, \max_{l\in\{1,\ldots,L\}} b_{i_l} - \sum_{l'=0}^{l} t_{i_{l'},i_{l'+1}}] \tag{19}$$

$[a_i, b_i]$ being the time window for node i which may lead to individual time window violations, but we consider that time window constraints are not "hard", in the sense that in real life, violation of a time window results in customer's inconvenience, but not in impossibility of route completion.
The choice of the methodology is motivated by

1. the high efficiency of mini-clusters methods (especially in the case where one passenger represents a significant amount of the vehicle capacity, which is the case for ride-pooling in urban areas, with a typical capacity of 4)
2. Our previous local proximity measures are used to cut computing time for mini-cluster construction
3. the fact that it allows to forbid the vehicle from charging when it is not empty (the vehicle does not charge when it traverses a mini-cluster)
4. the separation of local and global constraints

3 HOW TO WORK WITH NYC TAXI CAB DATA ?

3.1 Data Overview

The data that we use is extracted from a data set that has been published online by a blogger [12] and originally hails from the New York City Taxi and Limousine Commission. The blogger obtained the information via a Freedom of Information Law (FOIL) request, and expresses it in a series of large data files separated into two categories: trip data and fare data. In this project we are interested in trip data, and specifically we want to analyse a representative trip carried out by a taxicab.
The data represents a year-worth of taxi cab data, separated into 12 files, each corresponding to a month, in the shape of an array, which entries of interest are :

- pickup datetime
- drop-off datetime
- passenger count
- trip time in seconds
- trip distance (in miles)
- pickup longitude
- pickup latitude
- drop-off latitude
- drop-off longitude

I chose to work on the file corresponding to January 2013: with 14 millions of lines, it does not exceed the limits of computer calculations. I managed the data with the Python library *Pandas*.
The trip data is expressed in latitude and longitude coordinates. Converting from (latitude, longitude) coordinates to a plan and using Cartesian coordinates is very complex and always approximate. Thus the use of the library *folium* is used to visualize the data. I used the library *geopy* to sort the data, using the distance function, which gives the distance between two $(lat, long)$ coordinates. I erased the data containing in which pickup or delivery were more than 100km away from the origin, taken to be Madison Square Garden, which erases most of the errors in the pickup and dropoff locations coordinates. Other errors were erased thanks to the clustering I carried out afterwards.

```
In [9]: map = folium.Map(location=(40.757977,-73.978165))

        #add a marker for every record in the filtered data, use a clustered view
        for index, row in df[0:500].iterrows():
                folium.Marker([row['dropoff_latitude'],row['dropoff_longitude']],icon=folium.Icon(color ='green')).add_to(map)

        map
```

Out[9]:

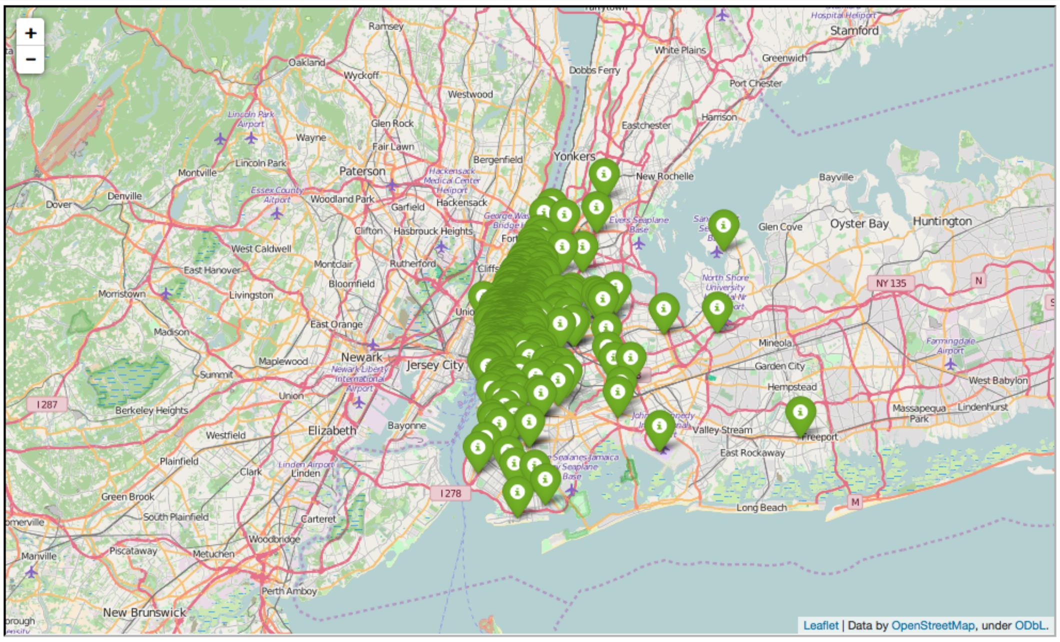

Figure 2: Folium visualization of the first 10 000 drop-off locations

3.2 Clustering

3.2.1 • Pick-up locations

By applying the *k-means method*, with 12 clusters, to the pickup $(lat, long)$ coordinates, I got the following clusters:

1. Upper West side (orange)
2. Greenwich Village (darkgreen)
3. East Village (pink)

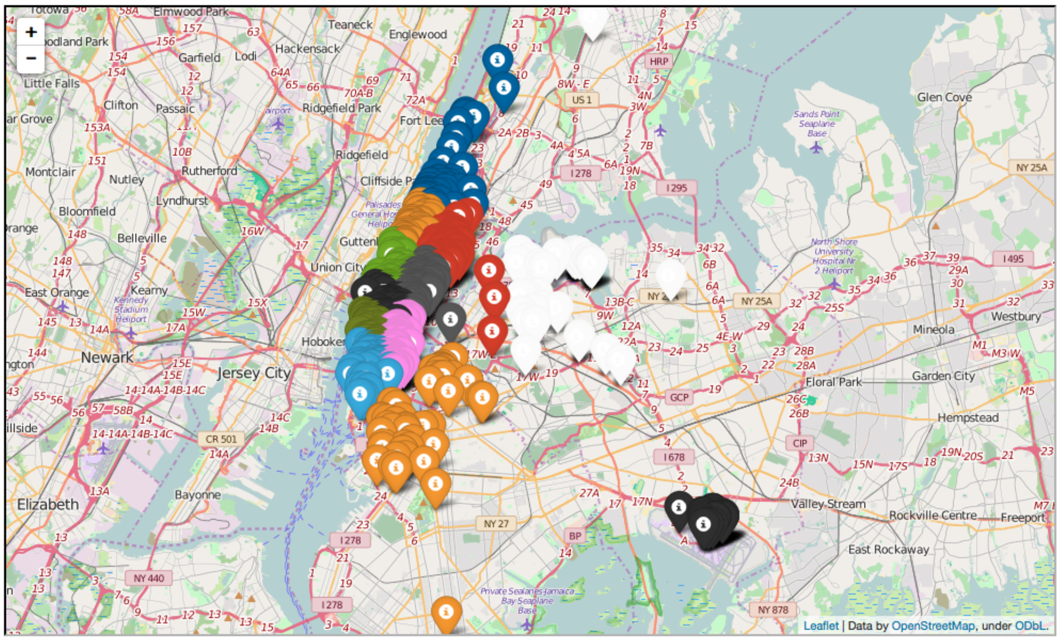
Figure 3: pickup clusters

4. JFK (black)
5. Queens (white)
6. Financial District (sky blue)
7. Upper East Side (red)
8. Hell's Kitchen (green)
9. Harlem (dark blue)
10. East Midtown (gray)
11. Brooklyn (orange)
12. Around Madison Square Garden (black)

Further investigation should be made concerning the time evolution of the drop-off repartition of the clusters. Notably, we notice that the high proportion of trips going from the Financial District to Greenwich Village and East Village + Gramercy may be due to people working in the Financial District, coming home to those neighborhoods, or going out. Thus, more trips towards those areas should be made around 8p.m., and a few during the rest of the day.

3.2.2 • Drop-off locations

By applying the *k-means method*, with 22 clusters, to the dropoff $(lat, long)$ coordinates, I got the following clusters:

1. Upper West Side (orange)
2. East Village + Gramercy (between Williamsburg Bridge and Queens-Midtown Tunnel)

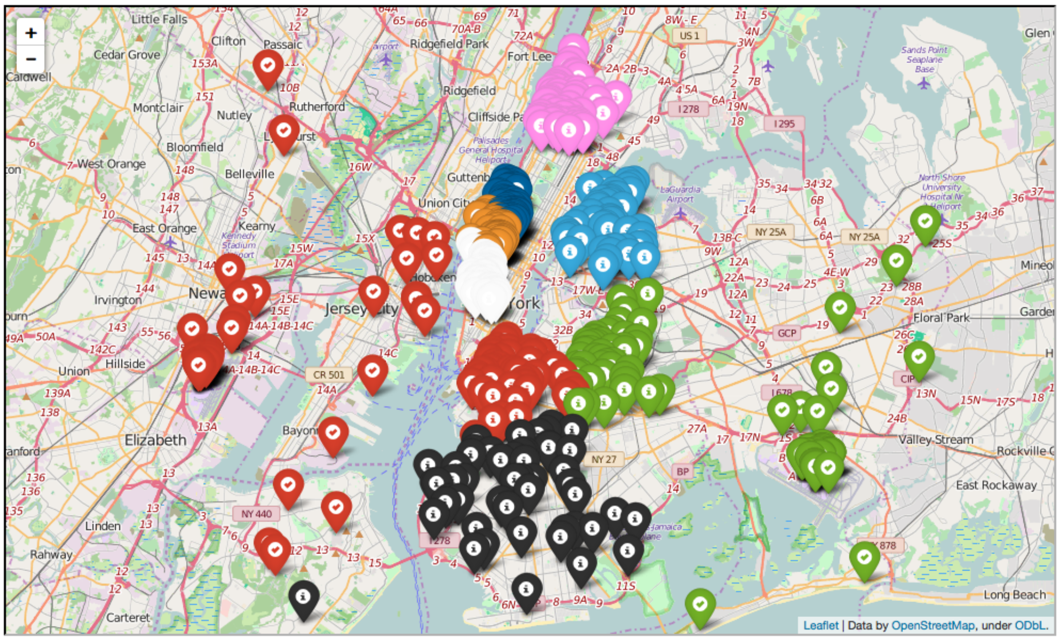
Figure 4: dropoff clusters

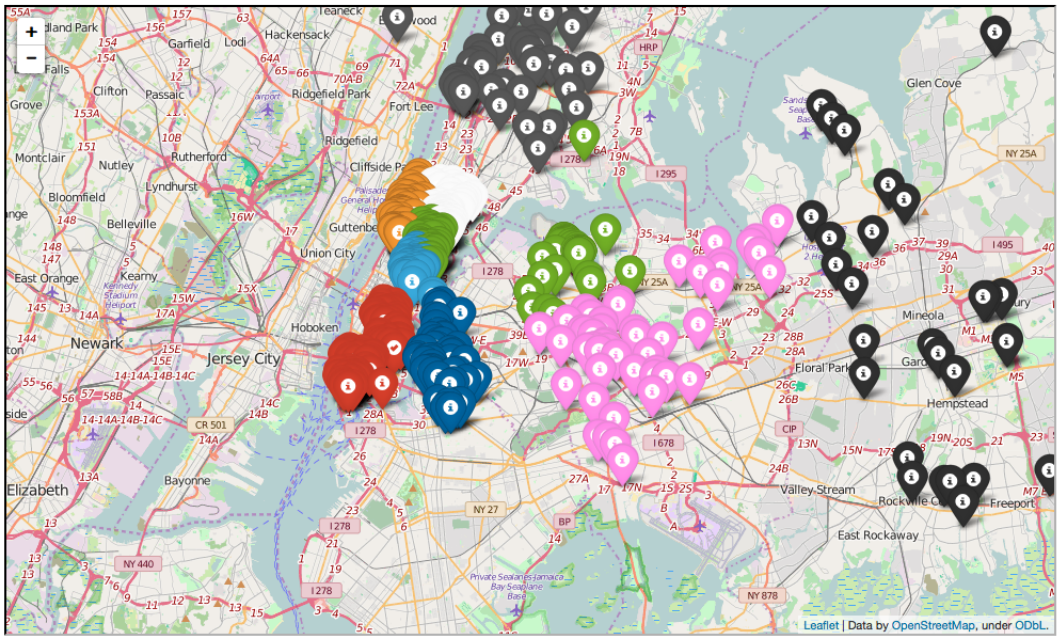
Figure 5: dropoff clusters

3. North of Upper East Side + Spanish Harlem
4. South of Upper East Side + North of Turtle Bay
5. Around Madison Square Garden and Lincoln Tunnel (23rd Street to 50th Street)
6. Greenwich Village and around (Canal Street to 23rd Street)
7. Upper East Side (66th St to 86th Street)
8. South of Upper West Side and Hell's Kitchen label
9. Financial district
10. Harlem
11. East Midtown (28th St to 50th St)
12. Downtown Brooklyn and around (Brooklyn heights, cobble hill)
13. South Brooklyn (black)
14. Astoria (Queens) (sky blue)
15. Bushwick (green)
16. Hell's Kitchen (dark blue)
17. JFK
18. La Guardia
19. Long Island
20. Queens
21. Williamsburg (darkblue)
22. Bronx (gray)

3.2.3 • Division into time slots

We intend to divide the day into time slots. As, contrary to locations, time is one-dimensional, instead of using clustering tools, we can basically use quantiles in order to form groups of the same size.
For example, if we want to divide the day into 24 time slots of equal size (i.e. same number of trips per day in this time slot on average), we compute $q_0, q_1, ..., q_{24}$ defined by

$$q_0 = 0 \tag{20}$$

$$\forall i \in \{1, ..., 24\}, \quad \text{Card}\{q_{i-1} \leq \text{pickup time}(t) \leq q_i | t \in \{\text{trips}\}\} = \frac{1}{24} \times \text{Card}\{\text{all trips}\} \tag{21}$$

Thus, we compute $q_1, .., q_{24}$ by:

```
slots = [0 for i in range(0,24)]
        for i in range(0,24):
            slots[i] = data['pickup_hour'].quantile((i+1)*1./24})
```

And we get for the january 2013 trips:

$$\begin{aligned}
q_0 &= 0\text{h} \\
q_1 &= 1\text{h } 11 \\
q_2 &= 2\text{h } 58 \\
q_3 &= 6\text{h } 20 \\
q_4 &= 7\text{h } 44 \\
q_5 &= 8\text{h } 40 \\
q_6 &= 9\text{h } 32 \\
q_7 &= 10\text{h } 27 \\
q_8 &= 11\text{h } 23 \\
q_9 &= 12\text{h } 15 \\
q_{10} &= 13\text{h } 06 \\
q_{11} &= 13\text{h } 57 \\
q_{12} &= 14\text{h } 47 \\
q_{13} &= 15\text{h } 35 \\
q_{14} &= 16\text{h } 29 \\
q_{15} &= 17\text{h } 24 \\
q_{16} &= 18\text{h } 09 \\
q_{17} &= 18\text{h } 48 \\
q_{18} &= 19\text{h } 27 \\
q_{19} &= 20\text{h } 07 \\
q_{20} &= 20\text{h } 49 \\
q_{21} &= 21\text{h } 34 \\
q_{22} &= 22\text{h } 18 \\
q_{23} &= 23\text{h } 04 \\
q_{24} &= 23\text{h } 59
\end{aligned}$$

We observe the slots are deformed compared to the division into 24 hours. The busier hours have the smallest slots (the smallest slot being 39 minutes 18h48 - 19h27) while the less busy hours have the widest slots (the widest slot is 3 hours and 22 minutes 2h58 - 6h20).
We can notice that the day slots are well balanced (they all around 45 minutes), whereas there is a big slot in the night. Thus, separating into proportionally balanced slots instead of raw division has the benefit of useless night slots and focusing on the busy hours of the day.

4 CONSTRUCTION OF MINI-CLUSTERS

Ioachim et al. [10] introduced the concept of *mini-clusters*. They define a mini-cluster as *a feasible segment of an itinerary where the vehicle is* never empty *between the first-pick-up point and the last drop-off point;* feasibility *is imposed with respect to the time window, capacity, priority and coupling constraints.*
We are looking for a set of *feasible* mini-clusters such that

- Each request is assigned to a single mini-cluster

- The cost necessary to cover the internal (within the mini-clusters) and the external (between the mini-clusters) path is minimal

This is formalized by the set partitioning formulation:

4.1 Set partitioning formulation

- n number of transportation requests
- $P = \{1, ..., n\}$: set of pick-up nodes
- $\forall i \in P, [a_i, b_i]$ time window to pick up passengers at nodes i
- $\forall i \in P, q_i$ passenger load at node i (non-negative for pick-ups, and non-positive for drop-off)
- Q maximum passenger capacity
- $D = \{n+1, ..., 2n\}$: set of delivery nodes : $n+i$ is the delivery node associated with i
- $N = \{0\} \cup P \cup D \cup \{2n+1\}$ set of vertices
- A set of arcs, to be defined further
- $\forall i, j \in A, d_{i,j}$ distance to travel from i to j
- $\forall i, j \in A, t_{i,j}$ travel time from i to j
- s fictive source node such that $\forall i \in P, c_{s,i} = \delta$, and $t_{s,i} = 0$,δ being the average external traveling cost between mini-clusters
- t fictive sink node such that $\forall i \in D, c_{i,t} = 0$, and $t_{s,i} = 0$
- M_{feas} the set of feasible mini-clusters : a mini-cluster is an ordered set of nodes $(i_0, ..., i_L)$ such that $L \in \mathbb{N}$, and the following conditions are satisfied :
  - The path respects time windows :

$$\exists (t_0, ..., t_L) \text{such that} \quad \forall l \in \{0, ..., L-1\} \quad t_{l+1} \geq t_l + t_{i_l, i_{l+1}} \tag{22}$$

$$\forall l \in \{0, ..., L\} a_{i_l} \leq t_l \leq b_{i_l} \tag{23}$$

  - If the mini-cluster contains a pick-up, it contains the associated delivery, and inversely
  - The capacity is never exceeded

$$\forall l \in \{0, ..., L\}, \sum_{l'=0}^{l} q_{i_{l'}} \leq Q \tag{24}$$

- c_m the cost of a mini-cluster, i.e, the sum of all the costs on the arcs within the mini-cluster
- $\gamma_{i,m}$: 1 if the mini-cluster m contains the request i, 0 otherwise

Decision variables

$$X_m = \begin{cases} 1 & \text{if } m \text{ is part of the solution} \\ 0 & \text{otherwise} \end{cases} \tag{25}$$

Objective function

$$z^* = \min_{m \in M} \sum_{m \in M} c_m X_m \tag{26}$$

Constraints Every request is in exactly one mini-cluster of the solution :

$$\forall i \in P, \sum_{m \in M_{\text{feas}}} \gamma_{i,m} X_m = 1 \tag{27}$$

Network definition To reduce the size of the problem, we limit the number of arcs of the network to be defined only between requests with identical triplets (pick-up zone, drop-off zone, time slot). As such, after adding for each request i the arcs $(0, i)$,$(i, n+i)$ and $(n+i, 2n+1)$, for each i and j, we may add to the network the following eight arcs : $(i, j), (j, i), (i, n + j), (j, n + i), (n + i, n + j), (n + j, n + i), (n + j, i), (n + i, j)$ if the pick-up and drop-off zone intersect, and the six arcs $(i, j), (j, i), (i, n + j), (j, n + i), (n + i, n + j), (n + j, n + i)$ otherwise. In the process we are eliminating other a priori feasible arcs. However, the likelihood that these would have passed the following stringent arc elimination tests, and be part of the optimal solution is rather small.

Resolution : Branch-and-Cut-and-Price The number of variables = Card(M_{feas}) is very large. A leading methodology for solving integer problems with a large number of variables is *Branch-and-Price*. It was first introduced for VRPTW by Desrochers, Desrosiers, and Solomon [7].
It corresponds to a Branch-and-Bound in which the linear relaxations are solved by column generation. When cutting planes are generated to strengthen the linear relaxations, the method is called *Branch-and- Cut-and-Price*.
Column generation (first introduced by Dantzig and Wolfe [6] in 1961) is an iterative process that can be applied for solving certain linear programs involving a huge number of variables such as the linear relaxation of the set partitioning model. In this context, the linear program is called the *Master Problem* (MP). At each iteration, a *Restricted Master Problem* (RMP) and an auxiliary problem, called the pricing problem or the *sub-problem*, are solved sequentially. The RMP is the MP restricted to a subset of its variables. It is solved by the simplex method to provide a primal and a dual solution. Given this dual solution, the sub-problem consists of finding negative reduced cost columns (= variables) for the RMP when some exist. When such columns are found, they are added to the RMP before starting a new iteration. If no negative reduced cost columns can be identified, the solution process stops and the current RMP primal solution is declared optimal for the MP.

4.2 Resolution of relaxation

```
Result: $(X^*_m, m \in M')$
for $i \in P$ do
    $L \leftarrow addConstraint(\sum_{m \in M_0} \gamma_{i,m} X_m = 1)$
end
$L \leftarrow addVariables(M_0)$;
$M \leftarrow M_0$ ;
$\bar{c}_m \leftarrow -1$;
$\sigma_i \in P \cup D \leftarrow 0$ ;
Procedure $ColumnGeneration(L)$
    while $\bar{c}_m < 0$ do
        $L \leftarrow addVariable(m)$;
        $M' \leftarrow M' \cup \{m\}$;
        $(z, (\pi_i, i \in P), (X^*_m, m \in M')) \leftarrow Simplex(L)$;
        for $i \in P$ do
            $\sigma_i \leftarrow \pi_i$
        end
        for $i, j \in A$ do
            $\bar{c}_{i,j} \leftarrow c_{i,j} - \sigma_i$
        end
        $(\bar{c}_m, m) \leftarrow SPPPDTW((\bar{c}_{i,j}, i, j \in A))$
    end
```

Algorithm 3: Column generation method

We associate with (27) the dual variables $\pi_i, i \in P$.
An iteration consists in

1. optimizing the restricted master problem in order to determine the current optimal objective function value $\bar{z}$ and dual multipliers $\pi_i, i \in P$
2. finding, if there is one, a variable X_m with *negative reduced cost*

$$\bar{c_m} = c_m - \sum_{i \in P} \pi_i \gamma_{i,m} \tag{28}$$

The reduced cost of a mini-cluster can be further evaluated as the sum of the reduced costs on the arcs contained in this mini-cluster. A mini-cluster contains nodes of all the existing types: origins, destinations, the source and the sink nodes. By solving the master problem we obtain n dual variables, which have to be assigned to the network. Let's define $\sigma_i, i \in N$ on all the nodes of the network:

$$\sigma_i = \begin{cases} 0 & \text{if } i = 0 \\ \pi_i & \text{if } 1 \leq i \leq n \\ 0 & \text{if } n+1 \leq i \leq 2n+1 \end{cases} \tag{29}$$

The reduced cost of a mini-cluster $m = (i_0, ..., i_L)$ becomes:

$$\bar{c_m} = \sum_{l=0}^{L} c_{i_l, i_{l+1}} - \sum_{i=1}^{L} \sigma_{i_l} = \sum_{l=0}^{L} (c_{i_l, i_{l+1}} - \sigma_{i_l}) \tag{30}$$

This relation implies that the assignment of the dual variables of the dual variables can be transferred from the nodes to the arcs. Hence, we can define $\bar{c_{i,j}}$ to be the reduced cost of an arc $(i, j) \in A$:

$$\bar{c}_{i,j} = c_{i,j} - \sigma_i \tag{31}$$

In order to find a mini-cluster of negative reduced cost, we solve the following problem :

$$\bar{c^*} = \min_{(x_{i,j}, i,j \in A),(t_i, i \in N),(y_i, i \in N)} \sum_{i,j \in A} \bar{c}_{i,j} x_{i,j} \tag{32}$$

$$\text{subject to} \quad \forall i \in P \cup D, \quad \sum_{j \in N} x_{i,j} = \sum_{j \in N} x_{j,i} \tag{33}$$

$$\sum_{j \in N} x_{s,j} = \sum_{j \in P} x_{j,t} = 1 \tag{34}$$

$$\forall i,j \in A, \quad t_j - (t_i + t_{i,j} x_{i,j}) \leq (1 - x_{i,j})(b_i - a_j) \tag{35}$$

$$\forall i \in N, \quad a_i \leq t_i \leq b_i \tag{36}$$

$$\forall i,j \in A, \quad y_j - (y_i + q_i) \leq (1 - x_{i,j})(Q - q_i) \tag{37}$$

$$\forall i \in N, \quad 0 \leq y_i \leq Q \tag{38}$$

$$\forall i,j \in A, \quad x_{i,j} \in \{0, 1\} \tag{39}$$

This a constrained pick-up and delivery shortest path problem on the network $G(N, A)$ with the costs $\bar{c}_{i,j}$ instead of $c_{i,j}$ on all the arcs $(i, j) \in A$.

Initial variables We form the initial basis of the LP's relaxation by first introducing the single request columns (i.e paths $0 \to i \to n + i \to 2n + 1$) into the master problem.

$$M_0 = \{(0, i, n + i, 2n + 1); i \in P\} \tag{40}$$

4.2.1 • Solving the sub-problem

The Shortest Path Problem with Pick-ups and Deliveries and Time Windows, is usually solved by dynamic programming, namely, by a labeling algorithm, introduced by Dumas, Desaulniers and Soumis in [8].

```
Result: $\mathcal{E} = \{\mathcal{P}\}$
$G.timeWindowShrink()$;
$G.arcElimination()$;
$\mathcal{E} \leftarrow$ {paths of length 3};
repeat
    for $\mathcal{P}_l(R,T,Z) \in \mathcal{E}$ do
        for $j \in V - R$ do
            if $(l,j)$ exists then
                $\beta \leftarrow$ True;
                $T' \leftarrow \max\{a_j, T + t_{l,j}\}$;
                if $T' > b_j$ then $\beta \leftarrow$ False ;
                /* time window violation                                  */
                if $j \in P$ then
                    $R' \leftarrow R \cup \{j\}$
                else if $j \in D$ and $n - j \in R$ then
                    $R' \leftarrow R - \{n - j\}$
                else
                    $\beta \leftarrow$ False /* pairing constraint violation  */
                end
                $Z' \leftarrow Z + \bar{c}_{l,j}$;
                if $\sum_{i \in R'} q_i > Q$ then $\beta \leftarrow$ False ;
                /* capacity violation                                     */
                if $\beta$ then $\mathcal{E}'.append(P'(R',T',Z'))$ ;
            end
        end
    end
    $\mathcal{E} \leftarrow \mathcal{E}'$;
    $\mathcal{E}.sortBy(l,R,T)$;
    for $\mathcal{P}_l(R,T,Z) \in \mathcal{E}, l \neq t$ do
        if $R = \emptyset$ then $\mathcal{E}.remove(\mathcal{P}_l(R,T,Z)))$ ;
        $\mathcal{E}.append(\mathcal{P}_t(\emptyset, T + t_{l,t}), Z))$ ;
        else
            for $i \in R$ do
                if $l \rightarrow n+i \rightarrow 2n+1$ is infeasible then $\mathcal{E}.remove(\mathcal{P}_l(R,T,Z))$ ;
            end
            for $i,j \in R, i \neq j$ do
                if $l \rightarrow n+i \rightarrow n+j \rightarrow 2n+1$ and $l \rightarrow n+j \rightarrow n+i \rightarrow 2n+1$ are infeasible then
                  $\mathcal{E}.remove(\mathcal{P}_l(R,T,Z))$ ;
            end
        end
    end
    for $\mathcal{P}_l(R,T,Z), \mathcal{P}'_l(R',T',Z') \in \mathcal{E}_k$ do
        if $R = R', T \leq T'$ and $Z \leq Z'$ then $\mathcal{E}.remove(\mathcal{P}'_l(R',T',Z'))$ ;
    end
    minZ $\leftarrow -\infty$;
    for $\mathcal{P}_t(R,T,Z) \in \mathcal{E}$ do
        if $Z \geq minZ$ then $\mathcal{E}.remove(\mathcal{P}_l(R,T,Z))$ ;
        else minZ $\leftarrow Z$ ;
    end
until $Card(\mathcal{E}) = 1$;
```

Algorithm 4: Labeling algorithm

Initialization We first apply a list of pre-processing steps to reduce the size of the problem.

Shrinking of time windows This is done by reducing the upper bounds of the time windows so that for $i = 1, ..., n$ the partial paths $n+i \to 2n+1$ and $i \to n+i \to 2n+1$ are admissible for all values $T_{n+i} \in [a_{n+i}, b_{n+i}]$. The upper and lower bounds are successively defined as

$$b_i = \min\{b_i, b_{n+i} - t_{i,n+i}\} \tag{41}$$
$$a_i = \max\{a_i, t_{0,i}\} \tag{42}$$
$$a_{n+i} = \max\{a_{n+i}, a_i + t_{i,n+i}\} \tag{43}$$

Arc elimination The arc set is reduced by applying the arc elimination criteria introduced in Dumas, Desrosiers, and Soumis in [8]. The constraints are used to eliminate the following inadmissible arcs

1. *priority* Arcs $(0, n+i)$ and $(n+i, i), i = 1, ..., n$ are eliminated
2. *vehicle capacity*
   If $q_i + q_j > Q$, $i, j \in \{1, ..., n\}, i \neq j$, then the following arcs are eliminated : $(i, j), (j, i), (i, n+j), (j, n+i), (n+i, n+j)$
3. *time windows*
   if $a_i + t_{i,j} > b_j, i, j \in \{1, ..., 2n\}$, then the arc (i, j) is eliminated
4. *time window and pairing of requests*
   - arc $(i, n+j)$ is eliminated if the path $j \to i \to n+j \to n+i$ is infeasible with $T_j = a_j$
   - arc $(n+i, j)$ is eliminated if the path $i \to n+i \to j \to n+j$ is infeasible with $T_i = a_i$
   - arc (i, j) is eliminated if the path $i \to j \to n+i \to n+j$ and $i \to j \to n+j \to n+i$ are infeasible with $T_i = a_i$
   - arc $(n+i, n+j)$ is eliminated if the path $i \to j \to n+i \to n+j$ and $j \to i \to n+i \to n+j$ are infeasible with $T_i = a_i$, and with $T_j = a_j$

Initialization with paths of length three We initialize the dynamic programming algorithm with paths of length three, all the paths beginning at the source node s. Each iteration a of the algorithm extends all the paths created during iteration $K-1$ to paths of length K.The difficulty of the problem is the high number of feasible paths that we have to treat during an iteration; this number increases in the beginning with the number of iterations and then decreases toward the end. An efficient way to eliminate this difficulty is to find effective criteria that eliminate the non-feasible paths as early as possible.
To greatly reduce the computational burden of the dynamic programming algorithm, at every iteration, comparisons are made between the reduced costs and time of paths visiting the same subset of nodes in different orders and having the same last visited node. The paths that are dominated are eliminated. In this section we show how this elimination process can be performed correctly and efficiently for the first three iterations without needing the values of the dual multipliers, necessary to compute the reduced costs. The dynamic programming algorithm may then be directly initialized with the list of paths of length three.
To prove the previous assertion, we analyze the path construction process. at start, n paths of length 1 are created, $s \to i$, with $1 \in P$. At the second iteration of the dynamic programming algorithm, these paths are extended. We classify the resulting paths of length 2 with respect to the subsets of visited nodes, in order to easily identify the paths that must be inputted to the elimination process. The paths of length two are of two types:

1. $0 \to i \to n+i, i \in P$ they can immediately be extended to the sink node $2n+1$ because the vehicle is empty at the last visited node. However, the corresponding mini-clusters (columns) have already been introduced into the master problem as the initial basis, so they need not be considered further in the column generation phase. The paths are eliminated.

2. $0 \rightarrow i \rightarrow j, i, j \in P$

The next iteration creates the paths of length three. We extend the second type of path presented above to paths of length three. The possible new paths contain three origins or two origins and one destination:

1. $0 \rightarrow i \rightarrow j \rightarrow k, 1 \leq k \leq n$ if the arc (j, k) exists and the time window, capacity, non-cycle and post-feasibility tests are satisfied

2. $0 \rightarrow i \rightarrow j \rightarrow n+i$ if the arc $(j, n+i)$ exists

3. $0 \rightarrow i \rightarrow j \rightarrow n+j$ if the time window constraints and the post-feasibility criterion are satisfied.

For the first type of path, we have to first verify the feasibility, and then the non-cycle and post-feasibility criteria. For the second type, the existence of the arc $(j, n+i)$ ensures that all the sub problem's constraints are satisfied. For the third type, the arc $(j, n+j)$ always exists so we need to verify the time window constraints and the post-feasibility criterion.
For paths with three origins, we may have to examine if any paths can be eliminated. This is so if paths $\mathcal{P}_A : 0 \rightarrow i \rightarrow j \rightarrow k$ and $\mathcal{P}_B : 0 \rightarrow j \rightarrow i \rightarrow k$ are generated. The same is true for the paths $0 \rightarrow i \rightarrow j \rightarrow n+i$ and $0 \rightarrow j \rightarrow i \rightarrow n+i$,where the set of origin nodes visited and the last node are identical for both paths. We shall next show that the elimination process can be performed without needing the dual multipliers values. The analysis is given for paths with three origins but is identical for paths with two origins and one destination.
Let us consider the two paths $\mathcal{P}_A$ and $\mathcal{P}_B$ of reduced costs $\bar{c_A}$ and $\bar{c_B}$ a,d $\pi_c, \sigma_0, \sigma_i, \sigma_j, \sigma_k$ the dual variables associated with a give iteration of the column generation algorithm. Then, we have :

$$\bar{c_A} = (1-\pi_c)(c_{0,i} + c_{i,j} + c_{j,k}) - \sigma_0 - \sigma_i - \sigma_j - \sigma_k \quad (44)$$

$$\bar{c_B} = (1-\pi_c)(c_{0,j} + c_{j,i} + c_{i,k}) - \sigma_0 - \sigma_j - \sigma_i - \sigma_k \quad (45)$$

The elimination process is thus reduced to comparing only the arcs costs and it can be performed in the initialization phase.
At the third iteration of the sub-problem, a very large number of paths are generated and have to be examined; we generate all the paths of length three only once for all the calls to the sub-problem made during the column generation method. This greatly reduces the number of operations made by the dynamic programming algorithm. For every call to the sub-problem, the dynamic programming algorithm directly starts by extending these paths of length three to paths of length four and then performs the elimination process with respect to the current reduced arc costs. This new initialization method reduced the total execution time of the mini-clustering approach by up to 40% for Ioachim et al. [10].

Definition of the labels We denote by $\mathcal{P}_l^k$ the path number k from the departure of node s to node l. This path is admissible if it respects the time window, priority and capacity constraints, and if it satisfies the pairing constraints when $l = t$. A label (S_l^k, T_l^k, Z_l^k) is associated with the path $\mathcal{P}_l^k$, where:

- S_l^k is the set of nodes visited on the path k
- $T_l^k \in [a_l, b_l]$ is the time of service at node l on the path k
- $Z_l^k \in \mathbb{R}$ is the sum of the costs on the path k

This information is sufficient to calculate on this path the load $Y_l^k = \sum_{i \in S_l^k} q_i$.
Let R_l^k be the subset of S_l^k containing the pickup nodes which have been visited, but whose corresponding delivery nodes have not been visited : $R_l^k = \{i \in S_l^k \cap \{1, ..., n\} | n+i \notin S_l^k\}$. A label (R_l^k, T_l^k, Z_l^k) is then associated with the admissible path $\mathcal{P}_l^k$. It is not sufficient to construct elementary paths (i.e. paths without cycle) as R_l^k contains no information on the transportation requests whose pickup and delivery nodes have both been visited.
The dynamic programming approach considers, for each label, the state (R_l^k, T_l^k) with a cost of Z_l^k. An attempt

may be made to extend a path $\mathcal{P}_l^k$ to a node j if arc (l,j) exists: one then obtains a new path $\mathcal{P}_j^{k'}$ with associated label $(R_j^{k'}, T_j^k, Z_j^k)$ calculated as follows :

$$R_j^{k'} = \begin{cases} R_l^k \cup \{j\} \text{ if } j \in P \text{ and } j \notin R_l^k \\ R_l^k - \{n-j\} \text{ if } j \in D \text{ and } n-j \in R_l^k \end{cases} \tag{46}$$

$$T_j^{k'} = \max\{a_j, T_l^k + t_{i,j}\} \text{if } T_j^{k'} \leq b_j \tag{47}$$

$$Z_j^{k'} = Z_l^k + \bar{c_{l,j}} \tag{48}$$

To be admissible, this path must also satisfy the capacity constraint for a pickup point : $Y_j^{k'} = Y_l^k + q_j \leq Q$ if $i \in \{1, ..., n\}$.
The optimal solution of the shortest path (with possibility of cycles) will be given by the path $\mathcal{P}_t^{k^*}$ with a minimal cost value $Z_t^{k^*}$: its label will be of the form $(\emptyset, T_t^{k^*}, Z_t^{k^*})$.
A path with a cycle will be obtained only if the network contains a negative cost cycle satisfy the same transportation requests twice while respecting the time windows for pickup and delivery. This kind of network is not common if the time windows associated with the nodes are small in comparison with the travel times between nodes.

Label elimination In the following paragraphs, we present four state elimination methods based respectively on the never-empty characteristic of mini-clusters, on the notion of a non-post-feasible label, on the notion of dominance between labels, and on the order of treatment of the labels. The elimination of a label results in the elimination of the associated path.

Mini-clusters are never empty As $R(S_l^k)$ is the subset of S_l^k containing the pickup nodes which have been visited, but whose corresponding delivery nodes have not been visited, a mini-cluster is empty when $R(S_l^k) = \emptyset$. Those paths are directly extended to t.

Post-feasibility A label associated with a path $\mathcal{P}_l^k$, admissible from node 0 to node l, which cannot be extended from node l to the return node $2n+1$ while respecting the time window and pairing constraints is called a non post-feasible label; such a label can be eliminated.

Proposition 4.1 *A label $(R(S_l^k), T_l^k, Z_l^k)$ such that $R(S_l^k) \neq \emptyset, i \in R(S_l^k)$ and $T_l^k > a_l$ is eliminated if the path extension $l \to n+i \to 2n+1$ is infeasible.*

Proposition 4.2 *A label (R_l^k, T_l^k, Z_l^k) such that $R_l^k \neq \emptyset, i, j \in R_l^k$ and $T_l^k > a_l$ is eliminated if both path extensions $l \to n+i \to n+j \to t$ and $l \to n+j \to n+i \to t$ are infeasible*

In the post-feasibility test is, in practice, limited to a subset of at most two delivery nodes.

Dominance between labels

Proposition 4.3 *If two labels are such that $R_l^k = RS_l^{k'}, T_l^k \leq T_l^{k'}$ and $Z_l^k \leq Z_l^{k'}$, then the label $(R_l^{k'}, T_l^{k'}, Z_l^{k'})$ can be eliminated.*

Order of treatment of labels Given the label $(\emptyset, a_0, 0)$ at node 0 at the first iteration, at iteration $K \geq 2$ of the algorithm we construct paths visiting exactly K nodes based on the paths generated in the preceding iteration. Once the labels from iteration $K-1$ have been treated, it is not necessary to store them to generate new labels and they are eliminated. For paths finishing at the arrival node t, we store only the label associated with the least cost admissible path with a cardinality less than or equal to K: this path may constitute the optimal solution to the shortest path problem.
Labels are not treated one by one. The labels at a node with the same set $R(.)$ are grouped together, thus facilitating the application of Proposition 4.3. The labels are sorted in increasing time order. The paths are extended simultaneously. This accelerates the verification of the capacity, priority, and pairing constraints and the elimination criteria.

4.2.2 • Solution of the master problem

The simplex algorithm is used to solve the master problem. This method gives the marginal costs σ_i necessary for the column generation and enables easy re-optimization each time new columns are generated. It produces a continuous optimal solution.

4.3 Obtaining optimal integer solutions

4.3.1 • The cuts

When the value $X_c = \sum_{m \in M} c_m X_m$ of the objective function is fractional, a cut is added to round this number up to at least the next integer: $\sum_{m \in M} c_m X_m \geq \lceil X_c^* \rceil$. The master problem is then reoptimized generating new columns as needed. If the solution is still non-integer, a second cut is added to round up this quantity to the next integer.

```
Result: $(X_m^*, m \in M')$
while $z^* = \sum_{m \in M'} c_m X_m^* \notin N$ do
    $L \leftarrow addConstraint(\sum_{m \in M'} c_m X_m > \lceil z^* \rceil)$;
    $(X_m^*, m \in M') \leftarrow ColumnGeneration(L)$
end
```

When the number of mini-clusters $X_0 = \sum_{m \in M} X_m$ is fractional, two branches are created, $\sum_{m \in M} X_m \geq \lceil X_0^* \rceil$ and $\sum_{m \in M} X_m \leq \lfloor X_0^* \rfloor$. On each branch, the sub-problem may generate new columns.

At the end of the column generation scheme, including the use of the two additional cuts, the procedure gives out Z_{inf} which is a primal lower bound on the objective function value.

4.3.2 • Branch-and-bound enumeration tree

One possible branching strategy sets at 0 or 1 the variables $X_{i,j}$ associated with the arcs which make up the mini-cluster $m \in M$. For a mini-cluster m satisfying n_m requests, $2n_m + 1$ branches are created. The first branch fixes the $2n_m + 2$ arcs at 1. For $k \in \{2, ..., 2n_m + 1\}$, the k^{th} branch fixes the first $2n_m + 1 - k$ arcs at 1 and the $(2n_m + 1 - k)^{th}$ at zero.
Suppose that variable X_m is fractional and that the associated mini-cluster is given by $0 \to 1 \to 2 \to n+2 \to n+1 \to 2n+1$. Five branches are created, as follows :

- $X_{0,1} = X_{1,2} = X_{2,n+2} = X_{n+2,n+1} = X_{n+1,2n+1} = 1$
  The solution includes the mini-cluster $0 \to 1 \to 2 \to n+2 \to n+1 \to 2n+1$. To find the other mini-clusters, rows 1,2, $n+1$ and $n+2$ are removed from the master problem, as are all the clusters containing one of these requests. The set partitioning problem is reoptimized generating new routes as needed.

- $X_{0,1} = X_{1,2} = X_{2,n+2} = 1$ and $X_{n+2,n+1} = 0$
  The solution includes a mini-cluster beginning by $0 \to 1 \to 2 \to n+2 \to k$ with $k \in \{1, ..., n\} - \{1, 2\}$. To complete this mini-cluster, and to find the other routes forming a solution, rows 1, 2, and $n+2$ of the master problem are combined and called row $1 \cap 2 \cap n+2$. Of all the mini-clusters containing either request 1 or 2, only those starting by $0 \to 1 \to 2 \to n+2 \to k$ with $k \in \{1, ..., n\} - \{1, 2\}$ are retained. These are assigned the value 1 on row $1 \cap 2 \cap n+2$ of the master problem. Similarly, in the sub-problem, requests 1, 2 and $n+2$ are combined. The new node $1 \cap 2 \cap n+2$ has the associated time interval $[a_1, \min\{b_1, b_2 - t_{1,2}\}]$ and SOC interval $[0, 1 - SOC(1 \to 2)]$. Arcs that arrived at 1 now arrive at $1 \cap 2 \cap n+2$ and those arriving at $n+2$ are eliminated. Arcs leaving 1 and arcs leaving or arriving at 2 are eliminated as is the arc $(n+2, n+1)$. Other arcs leaving $n+2$ now leave $1 \cap 2 \cap n+2$ with costs and durations increased by $c_{1,2} + c_{2,n+2}$ and $t_{1,2} + t_{2,n+2}$ respectively.

- $X_{0,1} = X_{1,2} = 1$ and $X_{2,n+2} = 0$
  The solution includes a mini-cluster beginning by $0 \to 1 \to 2 \to k$ with $k \in \{1, ..., n\} - \{1, 2\}$. To complete this mini-cluster, and to find the other routes forming a solution, rows 1 and 2 of the master problem are combined and called row $1 \cap 2$. Of all the mini-clusters containing either request 1 or 2, only those starting by $0 \to 1 \to 2 \to k$ with $k \in \{1, ..., n\} - \{1, 2\}$ are retained. These are assigned the value 1 on row $1 \cap 2$ of the master problem. Similarly, in the sub-problem, requests 1 and 2 are combined. The new node $1 \cap 2$ has the associated time interval $[a_1, \min\{b_1, b_2 - t_{1,2}\}]$ and SOC interval $[0, 1 - SOC(1 \to 2)]$. Arcs that arrived at 1 now arrive at $1 \cap 2$ and those arriving at 2 are eliminated. Arcs leaving 1 and as is the arc $(2, n+2)$. Other arcs leaving 2 now leave $1 \cap 2$ with costs and durations increased by $c_{1,2}$ and $t_{1,2}$ respectively.

- $X_{0,1} = 1$ and $X_{1,2} = 0$
  The solution includes a mini-cluster beginning by $0 \to 1 \to k$ with $k \in \{1, ..., n\} - \{1, 2\}$ and $(1, 2)$ is removed from the sub-problem's network

- $X_{0,1} = 0$
  Arc $(0, 1)$ is removed from the sub-problem's network

Note that for a large part of the mini-clusters, those for which all the pick-up nodes are treated before all the drop-off nodes, due to capacity constraints, we have $n_m \leq Q$, so we have at most $2Q + 1$ branches. For typical taxis, we have $Q = 4$, so at most $2Q + 1 = 9$ branches.

5 ELECTRIC VEHICLE ROUTING PROBLEM WITH RECHARGING STATIONS AND TIME WINDOWS

For this phase of the solution, I first tried to adapt the previous column-generation algorithm for the E-VRPTW, with a set partitioning formulation with route variables instead of mini-clusters. The difficult part of this adaptation was the sub-problem. Indeed, the sub-problem comes down to an optimal path problem for electric vehicles with time windows and recharging stations. Several studies studied the case of a single electric vehicle traveling from an origin to a destination by trying to determine an optimal path with respect to energy consumption, cost or time. Artmeier et al. [1] focus on the ability to recuperate energy through regenerative braking, but not at recharging stations. Sweda and Klabjan [16] considered the problem of determining a minimum-cost path for electric vehicles and integrated the idea of allowing the battery to be recharged along the path, but certain conditions had be satisfied in order for the state-space to be discrete and their backward recursion algorithm to be applicable, in particular, they could not include nodes with negative costs, which is the case in the sub-problem. Finally, there is not any method yet that could apply for our optimal path problem.

Thus, I took a whole different method and I chose to use a meta-heuristics method by adapting Schneider et al.'s method for Electric Vehicle Routing with recharging Stations and Time Windows in [13].
In this method, we only allow full charges at the charging station : when a vehicle goes to a charging station, it does not leaving the station until it is fully charged.

Parameters

- F' set visits to recharging stations, dummy vertices of set of recharging stations F
- F'_0 set visits to recharging stations, dummy vertices of set of recharging stations F including depot instance 0

- F'_{n+1} set visits to recharging stations, dummy vertices of set of recharging stations F including depot instance $n+1$
- V set of mini-clusters $V = \{1, ..., N\}$
- $V' = V \cup F'$ Set of mini-cluster vertices including visits to recharging stations,
- h charge consumption rate
- time window $[e_v, l_v]$
- travel distance $d_{v,w}$
- travel time $c_{v,w}$
- time service s_v at mini-cluster v
- energy cost of service at mini-cluster v : ε_v
- Q battery capacity
- g recharging rate (time/battery)

Decision variables

- $$x_{i,j} = \begin{cases} 1 & \text{if } (i,j) \text{ is used} \\ 0 & \text{otherwise} \end{cases} \tag{49}$$
- τ_i beginning of service at vertex i
- y_i remaining battery capacity on arrival at vertex i

Constraints

- each mini-cluster must have one trip leaving away from it
$$\forall i \in V, \sum_{j \in V'_{N+1}, i \neq j} x_{i,j} = 1 \tag{50}$$
- charging vertices must not be visited more than once (dummy vertices created in order to make several visits to charging stations possible)
$$\forall i \in F', \sum_{j \in V'_{N+1}, i \neq j} x_{i,j} \leq 1 \tag{51}$$
- flow conservation : at each vertex, the number of incoming arcs is equal to the number of outgoing arcs
$$\forall j \in V', \sum_{j} x_{i,j} - \sum_{k} x_{k,i} = 0 \tag{52}$$
- dynamics of time when i is a mini-cluster :
$$\forall i \in V_0, \forall j \in V'_{N+1}, i \neq j, \tau_j - \tau_i - (t_{i,j} + s_i) \leq M(1 - x_{i,j}) \tag{53}$$
- dynamics of time when i is a charging station :
$$\forall i \in F', \forall j \in V'_{N+1}, i \neq j, \tau_j - \tau_i - (t_{i,j} + g(Q - y_i)) \leq M(1 - x_{i,j}) \tag{54}$$

- time window

$$\forall j \in V', e_j \leq \tau_j \leq l_j \tag{55}$$

- dynamics of charge when i is a mini-cluster

$$\forall i \in V, \forall j \in V'_{N+1}, 0 \leq y_j \leq y_i - \varepsilon_i + (Q - hd_{i,j})x_{i,j} + Q \tag{56}$$

- dynamics of charge when i is a charging station visit vertex

$$\forall i \in V, \forall j \in V'_{N+1}, 0 \leq y_j \leq Q - hd_{i,j}x_{i,j} \tag{57}$$

Objective

$$\min_{(x_{i,j})} \sum_{i \in V'_0, j \in V'_{N+1}, i \neq j} d_{i,j}x_{i,j} \tag{58}$$

The solution uses route improvement heuristics, which consists in modifying a solution in order to obtain a new improved solution.

1. pre-processing step removing infeasible arcs
2. generation of initial solution S with a given number of vehicles. Infeasible solutions are allowed and evaluated based on a penalizing cost function
3. perform a feasibility phase during which the number of vehicles m is increased after no feasible solution has been found for a given number η_{feas} of iterations
4. After a feasible solution is found, another η_{dist} iterations are used to improve traveled distance : The search is guided by a VNS component described later. It uses the current VNS neighborhood $\mathcal{N}_\kappa$ to generate a random perturbation that serves as an initial solution for tabu iterations of the TS phase. The acceptance criterion of the VNS is based on simulated annealing (SA)

5.1 Pre-processing

An arc can be removed from the set of possible arcs if one of the two following conditions holds:

$$v \in V'_0, w \in V'_{N+1} \quad \text{and} \quad e_v + s_v + t_{v,w} > l_w \tag{59}$$

$$v, w \in V \quad \text{and} \quad \forall j \in F'_0, i \in F'_{N+1}, h(d_{j,v} + d_{v,w} + d_{w,i}) + \varepsilon_v + \varepsilon_w > Q \tag{60}$$

5.2 Generation of initial solution

1. set a maximum number of routes
2. all mini-clusters are sorted in increasing order of the sum of the angle between the depot, a randomly chosen point and the origin point of the mini-cluster, and the angle between the depot, the same randomly chosen point, and the end point of the mini-cluster
   The "angle" is calculated via the (latitude, longitude) coordinates
3. mini-clusters are iteratively inserted into the active route at the position causing minimal increase in traveled distance until a violation of battery capacity occurs
4. If a violation occurs, we activate a new route until at most the predefined number of routes are opened

```
$\mathcal{N}_\kappa \leftarrow$ set of VNS neighborhood $\forall \kappa = 1, ..., \kappa_{\text{max}}$;
$S \leftarrow generateInitialSolution()$;
$\kappa \leftarrow 1$;
$i \leftarrow 0$;
$feasibilityPhase \leftarrow$ True;
while $feasibilityPhase \vee (\neg feasbilityPhase \wedge i < \eta_{dist})$ do
    $S' \leftarrow generateRandomPoint(\mathcal{N}_\kappa(S))$;
    $S'' \leftarrow applyTabuSearch(S', \eta_{\text{tabu}})$ ;
    if $acceptSA(S'', S)$ then
        $S \leftarrow S''$;
        $\kappa \leftarrow 1$
    else
        $\kappa \leftarrow \kappa \pmod{\kappa_{\text{max}}} + 1$
    end
    if $feasibilityPhase$ then
        if $\neg feasible(S)$ then
            if $i = \eta_{feas}$ then
                $S \leftarrow addVehicle(S)$;
                $i \leftarrow -1$
            end
        else
            $feasibilityPhase \leftarrow$ False;
            $i \leftarrow -1$
        end
    end
    $i \leftarrow i + 1$
end
```

Algorithm 5: VNS-TS algorithm for E-VRPTW

5. The battery capacity violation is determined under the assumption that no recharging possibility exists
6. To consider time window requirements, a mini-cluster u is only allowed to be inserted between successive vertices i, j if $e_i \leq e_u \leq e_j$

The last rule helps to fulfill time windows constraints, but feasibility is only guaranteed concerning battery capacity for all routes but the last.

5.3 Generalized cost function

VNS/TS allows infeasible solutions during the search process. We evaluate a solution by means of the following generalized cost function:

$$f_{\text{gen}} = f(S) + \beta P_{\text{tw}}(S) + \gamma P_{\text{batt}}(S) + P_{\text{div}}(S) \tag{61}$$

where $f(S)$ denotes the total traveled distance, $P_{\text{tw}}(S)$ the time window violation,$P_{\text{batt}}(S)$ the battery capacity violation, $P_{\text{div}}(S)$ a diversification penalty, and β and γ are factors weighting the violations. The penalty factors are initialized (β_0, γ_0) and dynamically updated between a given lower $(\beta_{\min}, \gamma_{\min})$ and upper bound $(\beta_{\max}, \gamma_{\max})$. To balance between diversification and intensification, they are increased by a factor δ after every η_{penalty} consecutive iterations in which the respective constraint has been violated and divided by δ if the respective constraint was satisfied.
In the following, we describe the efficient calculation of the constraint violations.

Battery capacity violation To calculate battery capacity violations, we define the following two variables for each vertex of a route $r = (v_0, .., v_{n+1})$: $\Gamma^{\rightarrow}_{v_i}$ contains the battery charge that is needed to travel either from the last recharging station visit or from the depot to vertex v_i, in addition to the SOC cost of the visit, and $\Gamma^{\leftarrow}_{v_i}$ is the battery charge that is needed to travel from v_i to either the next recharging station or the depot:

$$\Gamma^{\rightarrow}_{v_i} = \begin{cases} h.d_{v_{i-1},v_i} & \text{if } v_{i-1} \in F'_0 \\ \Gamma^{\rightarrow}_{v_{i-1}} + h.d_{v_{i-1},v_i} + \varepsilon_i & \text{otherwise} \end{cases} \qquad i = 1, ..., n+1 \tag{62}$$

$$\Gamma^{\leftarrow}_{v_i} = \begin{cases} h.d_{v_i,v_{i+1}} & \text{if } v_{i+1} \in F'_{n+1} \\ \Gamma^{\leftarrow}_{v_{i-1}} + h.d_{v_{i-1},v_i} + \varepsilon_i & \text{otherwise} \end{cases} \qquad i = 0, ..., n \tag{63}$$

$$\tag{64}$$

Using these variables, the battery capacity violation of a route r can be defined as the sum of individual violations at every visit to a recharging station and on return to the depot:

$$P_{\text{batt}}(r) = \sum_{v_i \in \text{Vert}(r) \cap F'_{N+1}} \max(\Gamma^{\rightarrow}_{v_i} - Q, 0) \tag{65}$$

Thus, changes in battery capacity violation can be calculated in $\mathcal{O}(1)$ for all neighborhood operators described later by calculating the change in battery capacity violation of the recharging stations immediately following the points of vertex insertion, removal, or merging of both routes.

Time window violation The approach is based on the notion of time travel, i.e., the calculation of the violation at a customer that follows a customer with a time window violation is executed as if a travel back in time to the latest feasible arrival time at the preceding (violating) customer had taken place. By putting a penalty only on the first vertex where a time window is violated instead of propagating the violation along the entire route, the approach avoids penalizing good customer sequences only because they occur after a time window violation. Another important advantage of the approach is that potential time window violations for inter-route moves can be calculated in $\mathcal{O}(1)$ for most of the classical neighborhood operators like Relocate, Exchange, Or-Opt and 2-opt*.
More precisely, for the VRPTW, by storing forward and backward time window penalty slacks, it is possible to

calculate in constant time the time window penalties of a route $r_1 = (v_0, ..., u, w, ..., v_{n+1})$ that is constructed from two partial routes $(v_0, ..., u)$ and $(w, ..., v_{n+1})$ or a route $r_2 = (v_0, ..., u, v, w, ..., v_{n+1})$ that is constructed by inserting a vertex v between two partial routes $(v_0, ..., u)$ and $(w, ..., v_{n+1})$.
This is not always possible if recharging stations are present because the recharging time at a station depends on the battery charge, which itself depends on the traveled distance to the recharging station. If the partial route $(w, ..., v_{n+1})$ contains a recharging station z, i.e., $w, .., z+1$, slack variables have to be recalculated by traversing the partial route $w, ..., z+1$ for r_1 and the partial route $v, ..., z+1$ for r_2. Note that a recharging station in the first partial route $((v_0, ..., u)$ or the vertex to be inserted v being a recharging station does not necessitate a recalculation.
We introduce the following notations :

- $\tilde{a_{v_i}}$ extended earliest start time of service at a customer v_i, and $\tilde{a'_{v_i}}$ extended earliest arrival time to the depot, calculated as follows:

$$\tilde{a_{v_0}} = e_0 \qquad \tilde{a'_{v_0}} = e_0 \tag{66}$$
$$\tilde{a'_{v_i}} = \tilde{a_{v_{i-1}}} + s_{v_{i-1}} + c_{v_{i-1},v_i} \qquad i = 1, ..., n+1 \tag{67}$$
$$\tilde{a_{v_i}} = \max(\tilde{a'_{v_i}}, e_{v_i}) \quad \text{if } \tilde{a'_{v_i}} \leq l_{v_i} \quad i = 1, ..., n+1 \tag{68}$$
$$\tilde{a_{v_i}} = l_{v_i} \quad \text{if } \tilde{a'_{v_i}} > l_{v_i} \quad i = 1, ..., n+1 \tag{69}$$

- forward time window penalty $\forall i \in \{0, .., n+1\}$

$$TW^{\rightarrow}_{v_i} = \sum_{j=0}^{i} \max(\tilde{a'_{v_i}} - l_{v_j}, 0) \tag{70}$$

- $\tilde{z_{v_i}}$ extended latest start time of the service at a customer v_i and

$$\tilde{z'_{v_i}}$$

extended latest arrival time to the depot, calculated as follows:

$$\tilde{z_{v_0}} = e_0 \qquad \tilde{a'_{v_0}} = e_0 \tag{71}$$
$$\tilde{z'_{v_i}} = \tilde{z_{v_{i+1}}} - s_{v_i} - c_{v_i,v_{i+1}} \qquad i = 0, ..., n \tag{72}$$
$$\tilde{z_{v_i}} = \max(\tilde{z'_{v_i}}, l_{v_i}) \quad \text{if } \tilde{z'_{v_i}} \geq e_{v_i} \quad i = 0, ..., n \tag{73}$$
$$\tilde{z_{v_i}} = e_{v_i} \quad \text{if } \tilde{z'_{v_i}} < e_{v_i} \quad i = 0, ..., n \tag{74}$$

- backward time window penalty slack $\forall i \in \{0, .., n+1\}$

$$TW^{\leftarrow}_{v_i} = \sum_{j=0}^{i} \max(e_{v_j} - \tilde{z'_{v_i}}, 0) \tag{75}$$

Let us consider a route $r = (0, ..., x, v, y, ..., 0)$ constructed by inserting a node v in between two partial paths $(0, .., x)$ and $(y, ..., 0)$.
We distinguish three insertion cases:

1. Customer v is reached within its time window, i.e., we neither have to wait nor do we have to time travel backward
2. Customer v is reached before the start of the associated time window, i.e., we have to wait before service can be started
3. Customer v cannot be reached before the end of its time window, i.e., we have to time travel backward at v

Case 1: Reach mini-cluster v within Time Window

$$TW(r) = TW_x^{\rightarrow} + TW_y^{\leftarrow} + \max(0, \tilde{a}_x + s_x + c_{x,v} - (\tilde{z}_y - c_{v,y} - s_v)) \tag{76}$$

Case 2 : wait at mini-cluster v If $\tilde{a_x} + s_x + c_{x,v} < e_v$:
At mini-cluster x with $TW_x^{\rightarrow}$, we start service at $\tilde{a}_x$ and arrive at mini-cluster v at $\tilde{a}_x + s_x + c_{x,v}$. We reach v before its time winodw, wait until e_v and the arrival at y can be calculated by adding $q_v + c_{v,y}$ to e_v. We have to arrive a y before $\tilde{z}_y$ for the time window violation $TW_y^{\leftarrow}$ to be valid. If the arrival time at y is later, we add the difference as time window violation :

$$TW(r) = TW_x^{\rightarrow} + TW_y^{\leftarrow} + \max(0, -\tilde{z}_y + c_{v,y} + s_v + e_v)) \tag{77}$$

Case 3 : arrive late at mini-cluster v If $\tilde{a_x} + s_x + c_{x,v} > l_v$:
At mini-cluster x with $TW_x^{\rightarrow}$, we start service at $\tilde{a}_x$ and arrive at mini-cluster v at $\tilde{a}_x + s_x + c_{x,v}$. The arrival time at y can be calculated by adding $\tilde{z}_y + c_{v,y} + s_v$ to l_v. Thus, the following holds :

$$TW(r) = TW_x^{\rightarrow} + TW_y^{\leftarrow} + \tilde{a}_x + s_x + c_{x,v} - l_v + \max(0, l_v - \tilde{z}_y + c_{v,y} + s_v)) \tag{78}$$

This can be reduced to the following equation that covers all three cases:

$$TW(r) = TW_x^{\rightarrow} + TW_y^{\leftarrow} + \max(0, \tilde{a}_x + s_x + c_{x,v} - l_v) + \max(0, \max(e_v, \min(l_v \tilde{a}_x + s_x + c_{x,v})) + s_v + c_{v,y} - \tilde{z}_y) \tag{79}$$

5.4 The Variable Neighborhood Search Component

General VNS algorithm Given a predefined set of neighborhood structures and the current best solution S,

1. a neighboring solution S' is randomly generated in the *shaking phase* by means of the neighborhood structure $\mathcal{N}_\kappa$, VNS proceeds as follows:
2. a greedy local search is applied on S' to determine the local minimum S''
   - If S'' improves on the current best solution S, the VNS algorithm accepts the solution S'' and restarts with neighborhood $\mathcal{N}_1$ and the new starting solution S''
   - if S'' is worse than the incumbent best solution, S'' is refused. In this case, VNS performs a random perturbation move according to the next more distant neighborhood structure $\mathcal{N}_{\kappa+1}$, starting again with S

In our hybrid VNS/TS algorithm, the shaking phase is equal to the standard VNS approach, but the intensification phase and the acceptance criterion clearly differ.

Shaking phase In every iteration, our VNS performs a random perturbation move according to the predefined neighborhood structure $\mathcal{N}_\kappa$. The neighborhood structures are all defined by means of the *cyclic-exchange operator*. In the cyclic-exchange, customer sequences of arbitrary length are simultaneously transferred between routes.

Local search Our κ neighborhood structures are defined according to two parameters : the number of routes that form the cycle, and $\lambda_{\max}$. In each route r_k, we randomly select the number of successive vertices that form the trans-location chain in the interval $[0, \min\{\Lambda_{\max}, n_k\}]$, where n_k denotes the number of customers and stations contained in r_k, and $\Lambda_{\max}$ denotes the maximum number of trans-located vertices. The initial vertex of the trans-location chain is randomly chosen for each route.
Contrary to the local descent commonly used in VNS approaches, we use a TS to improve the randomly generated solution S' in the intensification phase. The TS, which is detailed later, is run for η_{tabu} iterations. Note that the perturbation move is added to the tabu list to prevent its reversal.

Acceptance criterion Subsequently, we compare the best solution S'' found during the TS to the initial solution S.Instead of accepting only improving solutions, we use an acceptance criterion based on the meta-heuristic SA to further diversify the search.
To be more precise, we always accept improving solutions, whereas deteriorating solutions are accepted according to the probability $e^{-(f(S'')-f(S))/T}$.
At the beginning of the search, we initialize the temperature T to T_0 in a way that a solution value $f(S'')$, which is Δ_{SA} worse than $f(S)$, is accepted with a probability of 50%. In this way, deteriorating solutions are often accepted, which helps to diversify the search. After every VNS iteration, the temperature is linearly decreased with a cooling factor that is chosen such that the temperature is below 0.0001 during the last 20% of iterations. By continuously decreasing the temperature during the search, an intensification is achieved and, finally, only improving solutions are accepted.

5.5 The Tabu Search Component

The TS phase starts from the solution S' generated by the perturbation move of the VNS component. In each iteration, the composite neighborhood $\mathcal{N}(S) = \cup_{v \in S} \mathcal{N}(v, S)$ where $\mathcal{N}(v, S)$ is a set of moves around vertex v for the current solution S, generated by applying the following neighborhood operators on every arc in the list of generator arcs:

- 2-opt*
- relocate
- exchange
- *stationInRe* (a new, problem-specific operator)

Each move is evaluated, and the best nontabu move is performed. A move is superior if it has a lower generalized cost function value.
In the definition, w denotes another vertex that may or may not belong to the same route as v. v^- (w^-) and v^+ (w^+) denote the predecessor and the successor of v (w).

2-opt*(v, S)

- remove (v^-, v) and (w, w^+), and add (v, w) and (v^-, w^+)
- remove (v, v^+) and (w^-, w), and add (v, w) and (v^+, w^-)

We apply 2-opt* for inter-route moves (v and w must belong to different routes) and define the operator for moving recharging stations, i.e., we allow the removal and insertion of arcs, including recharging stations.

Out-Relocate(v, S)

- Insert v between w^- and w, and add (v^-, v^+)
- Insert v between w and w^+, and add (v^-, v^+)

Out-Relocate is also defined for recharging stations and applied as intra and inter-route operator.

In-Relocate(v, S)

- Insert w between v^- and v, and add (w^-, w^+)
- Insert w between v and v^+, and add (w^-, w^+)

In-Relocate is also defined for recharging stations and applied as intra and inter-route operator.

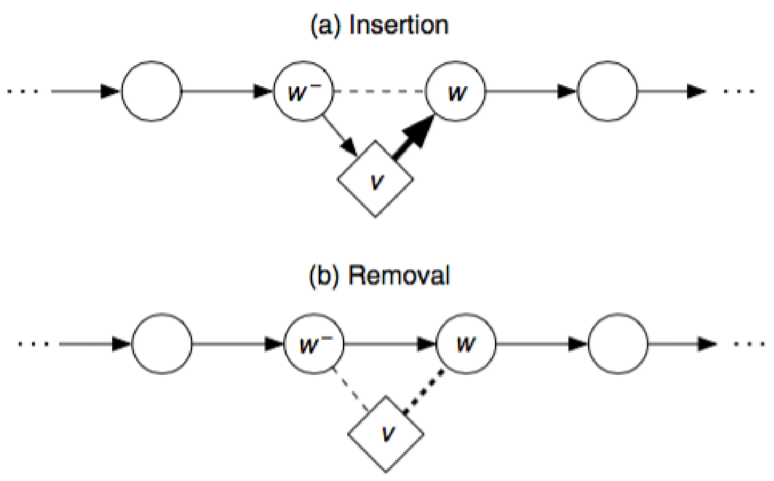

Figure 6: Insertion and Removal of a recharging station with the stationInRe operator

Exchange(v, S)

- Insert v between w and $(w^-)^-$, and insert w^- between v^- and v^+
- Insert v between w and $(w^+)^+$, and insert w^+ between v^- and v^+

The exchange operator is applied for inter and intra-route moves, but is not defined for recharging stations, i.e., we exclude the swapping of a recharging station with a customer or another station.

stationInRe operator The operator is defined for all generator arcs (v, w), where v is a recharging station. Let w^- denote the predecessor of vertex w. If the arc (v, w) is not part of the current solution, stationInRe performs the insertion as depicted in Figure 5.5(a): remove (w^-, w) and add (w^-, v) and (v, w). If the arc is already present, a recharging station is removed as shown in Figure 5.5(b) : remove (w^-, v) and (v, w) and add (w^-, w).
We set every arc ξ that is deleted from the solution by the execution of a move tabu, i.e., we forbid the reinsertion of the arc into specific parts of the solution for a specified number of iterations called *tabu tenure*. Because station visits have a strong effect on charge levels and also on time windows due to the recharging times incurred, we define the tabu attribute (ξ, k, μ, ζ). It prohibits the insertion of arc ξ into route r_k between μ and ζ , where $\mu, \zeta \in F_{0,n+1}$ denote either a station or the depot. In this way, we allow the reinsertion of an arc into a different part of the route. The tabu tenure ϑ is randomly drawn from the interval $[\vartheta_{\min}, \vartheta_{\max}]$. The tabu status of a move is lifted if a so-called aspiration criterion is met. In our case, if a feasible new best solution is generated.

Continuous diversification mechanism To further diversify the search, we adapt the continuous diversification mechanism presented in Cordeau, Laporte, and Mercier (2001) to VRPTW.
To this end, we define vertex-based attributes (u, k, μ, ζ) to describe that customer/station u is positioned between stations/depot μ and ζ in route r_k. In this way, each solution S can be characterized by the attribute set $B(S) = \{(u, k, \mu, \zeta)\}$. For each attribute, the frequency $p_{u,k,\mu,\zeta}$ of its addition to a solution in previous moves is memorized and used to penalize solutions according to the frequency of their attributes. Thus, we guide the search to explore the possibilities of using different stations and different positions of customers and stations (relative to other stations or the depot) within a route. A solution S' that deteriorates the current solution is penalized by:

$$P_{\text{div}}(S') = \lambda_{\text{div}} f(S') \sqrt{|V'|m} \sum_{(u,k,\mu,\zeta)} p_{u,k,\mu,\zeta} \tag{80}$$

where λ_{div} denotes the diversification factor used to control the amount of diversification. The scaling factor for $f(S')\sqrt{|V'|m}$ establishes a relation between the diversification penalty and the traveled distance and the investigated instance size ($|V'|$ customers and recharging visits and m vehicles). The TS procedure stops after η_{tabu} iterations.

LIST OF ALGORITHMS

REFERENCES

[1] Andreas Artmeier et al. “The optimal routing problem in the context of battery-powered electric vehicles”. In: *Workshop: CROCS at CPAIOR-10, Second International Workshop on Constraint Reasoning and Optimization for Computational Sustainability, Bologna, Italy*. 2010.

[2] John Barco et al. “Optimal routing and scheduling of charge for electric vehicles: Case study”. In: *arXiv preprint arXiv:1310.0145* (2013).

[3] L Bodin, B Golden, and A Assad. “ROUTING AND SCHEDULING OF VEHICLES AND CREWS–THE STATE OF THE ART”. In: (1981).

[4] Lawrence D Bodin and T Sexton. “The multi-vehicle subscriber dial-a-ride problem”. In: *TIMS studies in Management Science* 2 (1986), pp. 73–86.

[5] R Borndorfer et al. “Telebus Berlin: Vehicle routing scheduling in a dial a ride system”. In: *Konrad Zuse Zentrum fur Information Technik Berlin* (1997).

[6] George B Dantzig and Philip Wolfe. “The decomposition algorithm for linear programs”. In: *Econometrica: Journal of the Econometric Society* (1961), pp. 767–778.

[7] Martin Desrochers, Jacques Desrosiers, and Marius Solomon. “A new optimization algorithm for the vehicle routing problem with time windows”. In: *Operations research* 40.2 (1992), pp. 342–354.

[8] Yvan Dumas, Jacques Desrosiers, and Francois Soumis. “The pickup and delivery problem with time windows”. In: *European Journal of Operational Research* 54.1 (1991), pp. 7–22.

[9] Sevgi Erdoğan and Elise Miller-Hooks. “A green vehicle routing problem”. In: *Transportation Research Part E: Logistics and Transportation Review* 48.1 (2012), pp. 100–114.

[10] Irina Ioachim et al. “A request clustering algorithm for door-to-door handicapped transportation”. In: *Transportation Science* 29.1 (1995), pp. 63–78.

[11] Bernhard Meindl and Matthias Templ. “Analysis of commercial and free and open source solvers for linear optimization problems”. In: (2012).

[12] *NYC Taxis : a day in the life*. `http://whttp://nyctaxi.herokuapp.com/`. Accessed: 2016-08-30.

[13] Michael Schneider, Andreas Stenger, and Dominik Goeke. “The electric vehicle-routing problem with time windows and recharging stations”. In: *Transportation Science* 48.4 (2014), pp. 500–520.

[14] Paul Shaw. “Using constraint programming and local search methods to solve vehicle routing problems”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 1998, pp. 417–431.

[15] Marius M Solomon. “Algorithms for the vehicle routing and scheduling problems with time window constraints”. In: *Operations research* 35.2 (1987), pp. 254–265.

[16] Timothy M Sweda and Diego Klabjan. “Finding minimum-cost paths for electric vehicles”. In: *Electric vehicle conference (IEVC), 2012 IEEE international*. IEEE. 2012, pp. 1–4.