ADVANCED MODELLING FOR OPERATIONS ASSIGNMENT 1

Almir Gungor 10752670

Berk Ceyhan 10761821

Cagatay Onur 10781315

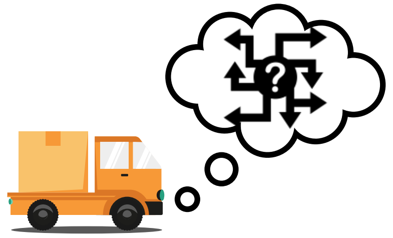
Lorenzo Ghedini 10586137

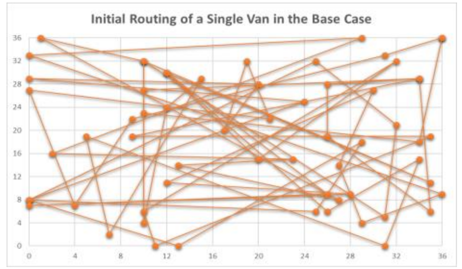
TABLE OF CONTENTS

WEAKNESSES OF THE BASE CASE

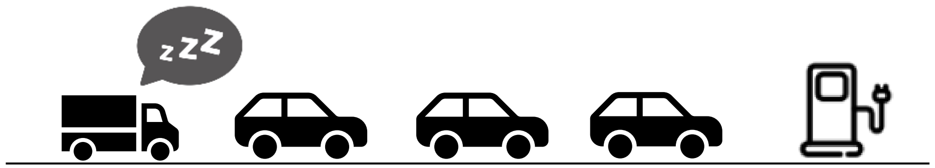
The following points are the ***major weaknesses*** of the base case. They are ***improved*** *in the project work.*

Routing Policy: There is no routing in the base case. Van just follows the order of customer index. Also, it is not possible to get routings for multiple vans. For optimizing the number of vans to deliver to each customer in the required timeslot, this project work offers a routing policy. The improved version will be explained in **slides [5:7].**



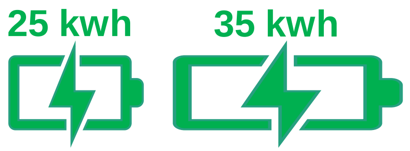


Charging Policy: In the base case, when charging is required, van goes to the closest charging station. This is not so desirable because the waiting time at the closing charging station might be very high. We provided a new charging policy, choosing the most convenient charging station. The improved version will be explained in **slides [8:16]**



Speed of Vans: To take an average deterministic speed is not sufficient to simulate the real-life problem. In order to make it more realistic, the speed can be converted to a **stochastic speed**. The improved version will be explained in **slide 19**.

Battery Size: In the base case, the same battery size is assumed to be fixed for all the vans. According to the routings, the required battery size for each van might differ. Thanks to a further calculation of the required battery sizes, the company can be more flexible in renting trucks. For example, a company can rent a more expensive truck (with a bigger battery) for the routing of 126 km while a less expensive truck (with a smaller battery) is sufficient for a routing of 96km. This can be a possible saving for the company. The improved version will be explained in **slide 17**.



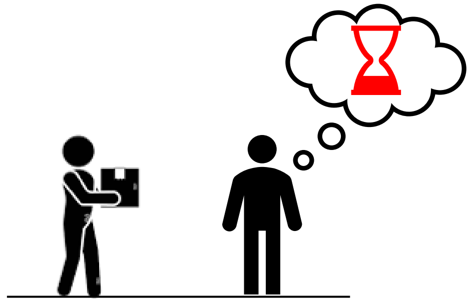
Weight Capacity of the Vans and Weight of the Loads: Both may affect the customer allocation to the vans and the energy consumed at any moment. Customer allocation might be affected because of the maximum number of orders carried in a van. With a domino effect, they might affect the tour time and even the number of vans required. In the description of the assignment, average order weight is given but not implemented in the base case. Yet, in the solution the weight capacity of the vans and average weight of the loads are implemented. This way, routings and energy usage values will be more accurate. Furthermore, the solution can even be improved by adding stochasticity to the average order weight. The improved version will be explained in **slide 18**.

WEAKNESSES OF THE BASE CASE

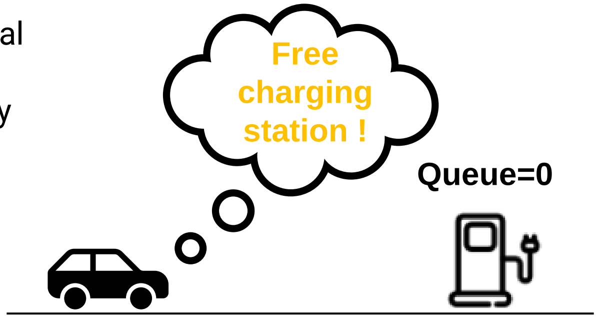
The following points are the ***minor weaknesses*** of the base case. They are **not implemented** in the new version of the model. In fact, neglecting these aspects is logical, considering the scope of the project, and the fact that implementing them will not bring a major gain. However, they are worth being kept in mind.

Space capacity of the vans: Average order volume can affect the customer allocation to each van because of the limitations of the number of orders carried by a van. It was not given as a parameter in the description of the assignment. If it is given, it can also be implemented. This can be implemented by generating the orders as an attribute of the agent "customer" by defining their weight and their volume at the start of the simulation and defining the customer allocation and routing according to it. This idea is **not implemented** in our model.

«Customers» can be another agent: They might have behaviours depending on the arrival time of vans. For example, they may have a desired delivery time and they may cancel the order if the van is late. This can represent the real-world situation better. This idea is **not implemented** in our model.



External arrivals of the cars to the charging stations with 1/45 uniform distribution The arrival frequency of external cars is a "fixed" probability in the base case. In a real world where the agents make rational decisions, a charging station with a short queue has a higher probability of being occupied by other cars such as with 1/30 uniform distribution. Modelling other cars also as agents who can observe the current waiting times of the charging stations and make decisions accordingly could represent the real-world situation better. This idea is **not implemented** in our model.

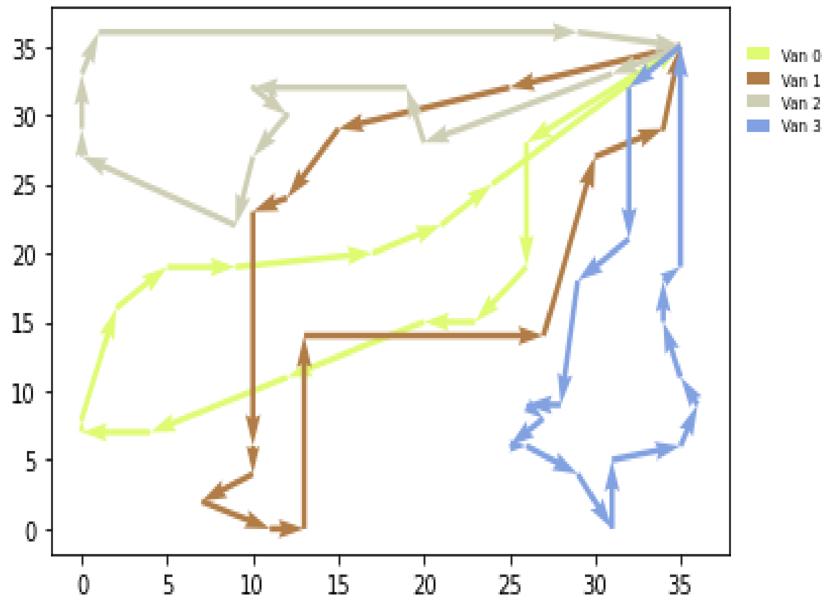


ROUTING

The routing issue was considered as a “multiple travelling salesman problem”. Two major approaches were followed during solution:

- **Optimal routes for each van:** The total distance travelled by each van is minimized. This approach needs to be combined with the following approach to improve the solution from one van to multiple vans case.
- **Minimum longest distance travelled amongst vans:** The total distance travelled by the van which travelled the most needs to be minimized in order to have a final picture where all the vans can travel similar distances. This will help the optimization of the routing, by balancing the route distance travelled by each van.

The figure on the right represents the visualization of the routing of the optimal case.



To determine this optimal routing for each van, *“ortools.constraint_solver”* library is used. This library was used since it consists of some interesting tools to solve the multiple travelling salesman problem. From this library, *“routing_enums_pb2”* and *”pywrapcp”* were imported since they are the elements responsible to import the constraints programming solver. Once imported, these two elements are responsible for finding the best solution of the routing problem. Running the simulation for an increasing number of vans is possible to see that for the solver the best solution correspond to the routing computed with 4 Vans, which has the minimum longest route of 126 km. According to the solver this is the best solution possible because also increasing the number of the vans the minimum longest route will be the same or even higher and so if the number of Van increases for the solver the solution will remain the same and so one or more vans will remain at the warehouse and just 4 will move. Of course the routing obtained in this way has some important limitations because it does the computation just in function of the distance and so it does not take into consideration the charging of the van or the fact that the velocity may be stochastic. For these reason we have to verify that this is effectively a good routing for our problem, by running multiple simulations.

Reference: https://developers.google.com/optimization/routing/vrp?hl=tr

ROUTING

The solution is reached by defining the following functions.

- **Main (Python Script Section 4.4):**

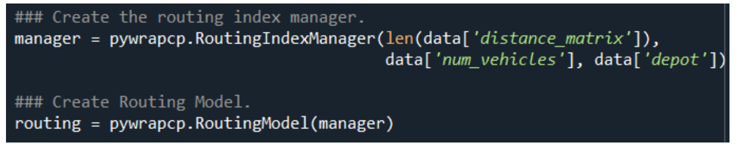
The main function is used to control the routing solution. It starts by taking data which is created by the function *"create data model"* and define two variables *"manager"* and *"routing"* to simplify the variable index usage.

Then, it is necessary to define the distances between each customer location to use in the decision process for the optimal routings. For that, a ***"distance callback"*** function is defined.

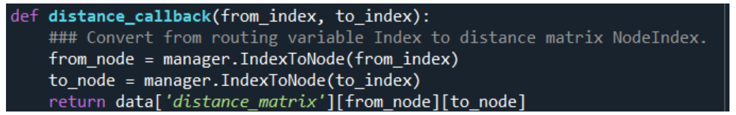
In this part, there is also the *"arc costs"* which define the cost of travel to be the distances of the arcs. In this way, the best routing can be defined by first creating a *"distance dimension"* which computes the cumulative distance traveled by each vehicle along its route. Then, a cost proportional to the maximum of the total distances along each route is set. In this part, a large coefficient (200) is chosen for the global span of the routes, which is the maximum of the distances of the routes. This makes the global span the predominant factor in objective function, so the program minimizes the length of the longest route.

Lastly before the solution, we have the ***"search parameters"*** in which the solver define the default search parameters from which we can define a heuristic method for finding the first solution of the problem through "path cheapest arc" which creates an initial route for the solver by repeatedly adding edges with the least weight that doesn't lead to a previously visited node (other than the depot).

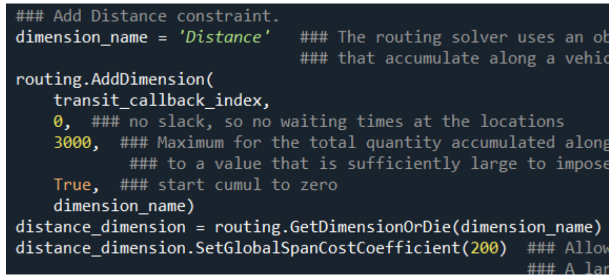
Once this initial routing is defined, it is possible to call the solver in order to return the solution and displays the optimal route



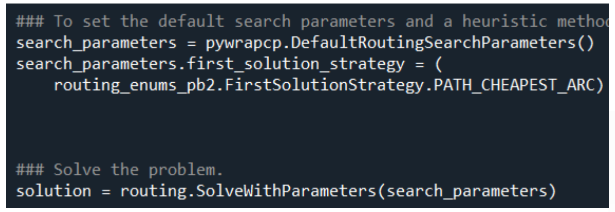
```
### Create the routing index manager.
manager = pywrapcp.RoutingIndexManager(len(data['distance_matrix']),
                                       data['num_vehicles'], data['depot'])

### Create Routing Model.
routing = pywrapcp.RoutingModel(manager)
```



```
def distance_callback(from_index, to_index):
    ### Convert from routing variable Index to distance matrix NodeIndex.
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)
    return data['distance_matrix'][from_node][to_node]
```



```
### Add Distance constraint.
dimension_name = 'Distance'   ### The routing solver uses an ob
                              ### that accumulate along a vehic
routing.AddDimension(
    transit_callback_index,
    0,  ### no slack, so no waiting times at the locations
    3000,  ### Maximum for the total quantity accumulated along
             ### to a value that is sufficiently large to impose
    True,  ### start cumul to zero
    dimension_name)
distance_dimension = routing.GetDimensionOrDie(dimension_name)
distance_dimension.SetGlobalSpanCostCoefficient(200)  ### Allow
                                                      ### A lar
```

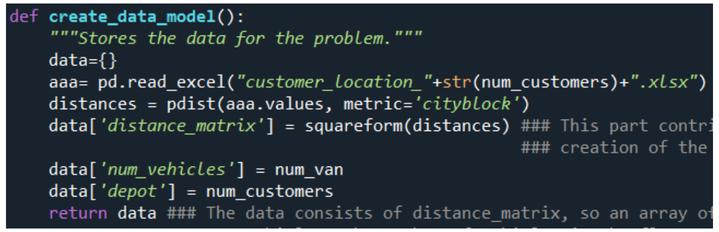


```
### To set the default search parameters and a heuristic metho
search_parameters = pywrapcp.DefaultRoutingSearchParameters()
search_parameters.first_solution_strategy = (
    routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)


### Solve the problem.
solution = routing.SolveWithParameters(search_parameters)
```

Reference: https://developers.google.com/optimization/routing/vrp?hl=tr

ROUTING

- **create_data_model (Python Script Section 4.1):** This function represents the data preparation part. Data is gathered as a matrix in which all the travel distance between the costumers and warehouse will be included as well as the number of customers, the number of vans and the starting point for the routing the warehouse.

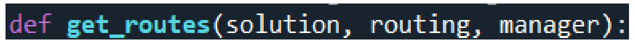


```
def create_data_model():
    """Stores the data for the problem."""
    data={}
    aaa= pd.read_excel("customer_location_"+str(num_customers)+".xlsx")
    distances = pdist(aaa.values, metric='cityblock')
    data['distance_matrix'] = squareform(distances) ### This part contri
                                                    ### creation of the
    data['num_vehicles'] = num_van
    data['depot'] = num_customers
    return data ### The data consists of distance_matrix, so an array of
```

- **distance_callback (Python Script Section 4.4.2)** of the indexes, manager.IndexToNode is used to define the from and to nodes.

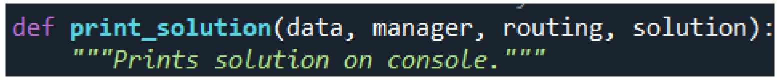
```
def distance_callback(from_index, to_index):
```

- **get_routes (Python Script Section 4.3):** This function is utilized to save the routes to a list or array. This has the advantage of making the routes available for the successive parts

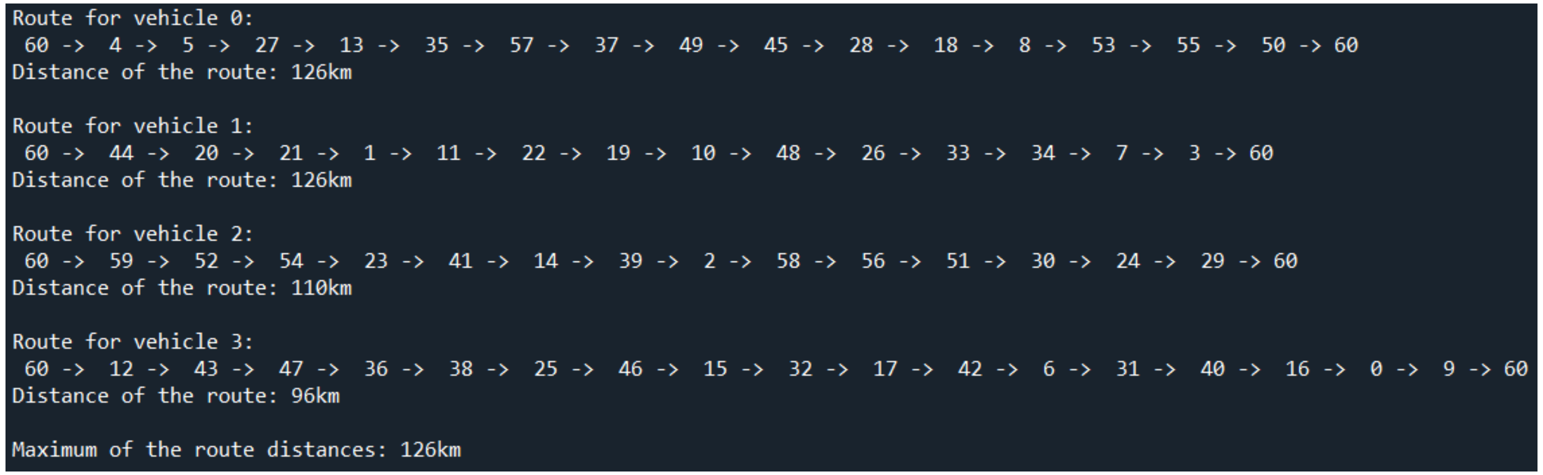


```
def get_routes(solution, routing, manager):
```

- **print_solution (Python Script Section 4.2):** This function is responsible for the display of the routes and the computation of the total distance travelled by the vans. The following is the output of the function.



```
def print_solution(data, manager, routing, solution):
    """Prints solution on console."""
```



```
Route for vehicle 0:
 60 ->  4 ->  5 ->  27 ->  13 ->  35 ->  57 ->  37 ->  49 ->  45 ->  28 ->  18 ->  8 ->  53 ->  55 ->  50 -> 60
Distance of the route: 126km

Route for vehicle 1:
 60 ->  44 ->  20 ->  21 ->  1 ->  11 ->  22 ->  19 ->  10 ->  48 ->  26 ->  33 ->  34 ->  7 ->  3 -> 60
Distance of the route: 126km

Route for vehicle 2:
 60 ->  59 ->  52 ->  54 ->  23 ->  41 ->  14 ->  39 ->  2 ->  58 ->  56 ->  51 ->  30 ->  24 ->  29 -> 60
Distance of the route: 110km

Route for vehicle 3:
 60 ->  12 ->  43 ->  47 ->  36 ->  38 ->  25 ->  46 ->  15 ->  32 ->  17 ->  42 ->  6 ->  31 ->  40 ->  16 ->  0 ->  9 -> 60
Distance of the route: 96km

Maximum of the route distances: 126km
```

CHARGING: Base Charging Policy vs New Charging Policy

The new charging policy has 3 main pillars, that will be explained in the following slides in detail: **Charging Station Selection**, **Charging Amount** and **When to Charge**.

BASE CHARGING POLICY

1.Charging Station Selection

Go to the **closest charging station.**

Weakness 1: The closest station can have a very long queue, making it a relatively worse option.

2.Charging Amount

Charging 20% (fixed) of the battery size

Weakness 2: 20% may be too much or too little. If it needs more than 20%, it will spend unneccessary time for charging for a second time. If the van needs less than 20%, it will spend unneccessary time charging excessive amount of battery.

3.When to Charge?

Monitor the battery level, and go charging when it is at the critical level (the battery about to die)

Weakness 3: In this way, the van misses many good opportunities (free charging stations) along the route. Being forced to charge only at the critical level, it may be forced to make a poor decision of going to a charging station with a very long queue.

NEW CHARGING POLICY

1.Charging Station Selection

Go to the **best charging station** which is determined by calculating charging station scores for each single charging station by considering:

. Current position of the vans

. The current waiting times and locations of the charging stations

. The location of the next customer

!!! NOTE: Since different vans have different current positions, the charging station scores will be different for different vans (e.g. a charging station close to a van will have probably a better score for that given van). This issue will be deepened in the following sections.

Advantage 1: The van will choose the most convenient charging station, not necessarily the closest one.

2.Charging Amount

Calculate **extra needed energy** in advance, depending on the route distance calculated at the beginning, and charge this extra needed energy.

For example:

Total travel energy needed= 28 kwh Battery size= 25kwh

Extra needed energy= (Total travel energy needed - Battery size) * safety = (28-25)*1.15 = 3.45 kwh

Advantage 2: The van will save time by eliminating unneccesary charging time, and also eliminating multiple charging due to the charging amount is calculated in advance

3.When to Charge?

When there is enough space in the battery to charge extra needed energy, start searching for good alternatives. The van will have the chance of postponing the charging until a good alternative is available.

For example :

Battery size= 25kwh Extra needed energy= 3.45 kwh

When battery level<=battery size-extra energy need=25-3.45=21.55 kwh $\longrightarrow$ start searching for a station

Advantage 3: The van will have more chance to optimize the charging station selection.

CHARGING: New Charging Policy: Charging Station Selection

The main objective of new charging policy is to calculate the **charging station score**

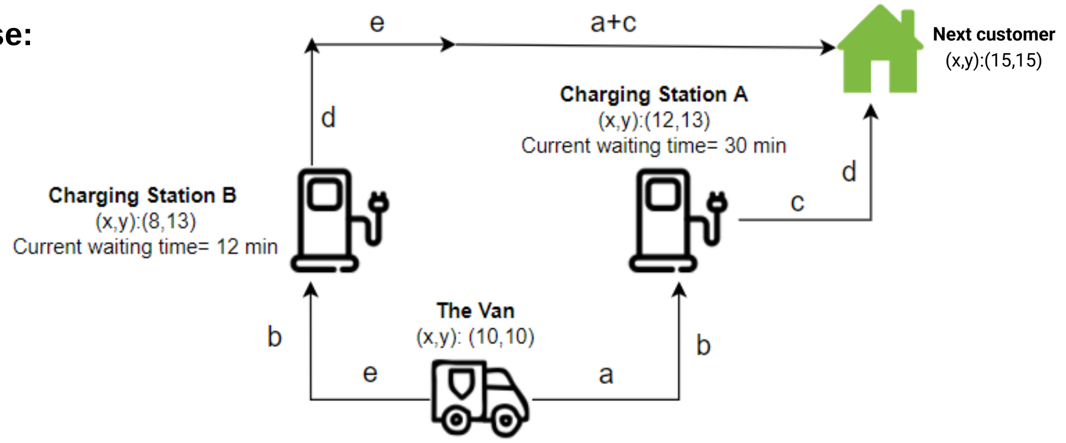
Charging Station Score =
Expected Waiting Time in the Queue + Extra Travel Time

Expected Waiting Time [EWT]:
This score is determined by making probability calculations using:
**Current Waiting Times at the charging stations [CWT]*
**Distance to arrive a charging station [DA]*
**Time to arrive to a charging station [TTT]=[DA]/speed*
**Probability of an external vehicle arriving before our van 1/45 uniform dist. (Will be explained in slide 11)*
** Mean charging time of external vehicles 25 min*
The **LOWER** this score, the **BETTER** option it is

Extra Travel Time [ETT]:
This score is penalizing the charging stations which increase the travelled distance.
The **LOWER** this score, the **BETTER** option it is.

Charging Station Score [CSS]:
The **LOWER** this score, the **BETTER** option it is.

Example Case:



Charging Station A	Charging Station B
EWT A = (CWT – TTT) + (TTT/45 * 25) = (30-10)+(10/45)*25 = **25.55 min** *Note: TTT= DA/(30km/h)=5 km/(30km/h)= 10 min*	**EWT A** = (CWT – TTT) + (AT * TTT * CTE) = (12-10)+(10/45)*25 = **7.55 min** *Note: TTT= DA/(30km/h)=5 km/(30km/h)= 10 min*
ETT B = ((a+b+c+d)-(a+b+c+d))/(30km/h) = **0 min** *Going to A does not cost anything in terms of distance, because it is located in between the van's location and the next customer's location (in the **no-cost zone**).*	**ETT B** = ((a+b+c+d+2*e)-(a+b+c+d))/(30km/h) = 2*e/(30km/h)=**8 min** *Going to B **costs extra 8 min** to the van.*
CSS A = 0 + 25.55 = **25.55 min**	**CSS B** = 8 + 7.55 = **15.55 min**

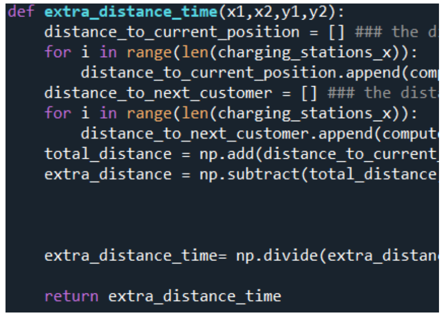
CONCLUSION: Choose B
(**lower** charging station score)

(See slides 10,11,12,13 for detailed explanations)

CHARGING: New Charging Policy: Charging Station Selection

The script for calculations are explained in this section.

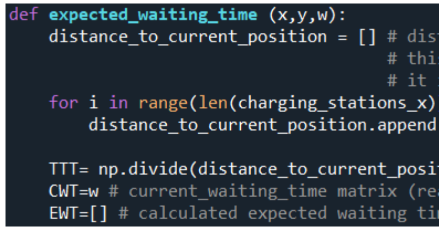
Extra Travel Time (Script Section: 5.1)



```
def extra_distance_time(x1,x2,y1,y2):
    distance_to_current_position = [] ### the d
    for i in range(len(charging_stations_x)):
        distance_to_current_position.append(com
    distance_to_next_customer = [] ### the dist
    for i in range(len(charging_stations_x)):
        distance_to_next_customer.append(comput
    total_distance = np.add(distance_to_current
    extra_distance = np.subtract(total_distance



    extra_distance_time= np.divide(extra_distan

    return extra_distance_time
```

(x1,y1): Coordinates of the van
(x2,y2): Coordinates of the next customer
distance_to_current_position: The array containing the distances of the charging stations to the current position of a van
distance_to_next_customer: The array containing the distances of the charging stations to the location of the next customer
total_distance= distance_to_current_position + distance_to_next_customer
extra_distance= total_distance - (Distance between the van and the customer)
extra_distance_time= converting extra_distance to time

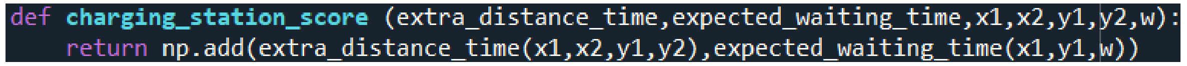
Expected Waiting Time (Slide 11 for detailed explanations) (Script Section: 5.2)



```
def expected_waiting_time (x,y,w):
    distance_to_current_position = [] # dis
                                      # thi
                                      # it
    for i in range(len(charging_stations_x)
        distance_to_current_position.append

    TTT= np.divide(distance_to_current_posi
    CWT=w # current_waiting_time matrix (re
    EWT=[] # calculated expected waiting ti
```

distance_to_current_position: The array containing the distances of the charging stations to the current position of a van
TTT: Converting distance_to_current_position to time
CWT: Current waiting times of the charging stations
EWT: Expected waiting time to calculate for each single charging station

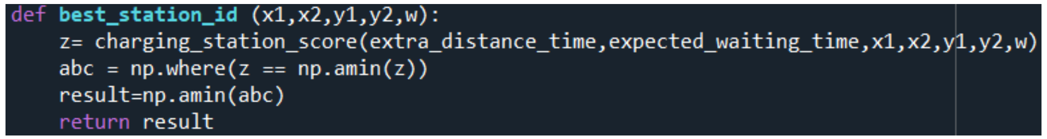
Charging_station_score (Script Section: 5.3)



```
def charging_station_score (extra_distance_time,expected_waiting_time,x1,x2,y1,y2,w):
    return np.add(extra_distance_time(x1,x2,y1,y2),expected_waiting_time(x1,y1,w))
```

(x1,y1): Coordinates of the van
(x2,y2): Coordinates of the next customer
w: current waiting time matrix of the charging stations
charging_station_score = extra_distance_time + expected_waiting time
Note: *Calculating charging_station_score is the* ***main objective*** *of these calculations.*
The van will choose the charging station with ***THE LOWEST*** *charging station score.*

best_station_id (Script Section: 5.4)



```
def best_station_id (x1,x2,y1,y2,w):
    z= charging_station_score(extra_distance_time,expected_waiting_time,x1,x2,y1,y2,w)
    abc = np.where(z == np.amin(z))
    result=np.amin(abc)
    return result
```

This function gets the id of the charging station with the lowest charging_station_score.
This id will be used to make the van go to that specific charging station.

(Python Script 5.2)

CHARGING: New Charging Policy: Charging Station Selection Probability Calculations

The best charging station is chosen, and the van starts to go there. It would be shame if another car goes to the charging station before our van! The longer the time passes until the van reaches the station, the more the risk of another car reaching before our van. So, **the probability of an external car arriving before us** is a crucial factor that needs to be calculated. It is calculated and added in the calculation of EWT. The section explains the corresponding concept and the script.

Formula 1:

EWT= CWT-TTT+ TTT/45 * 25

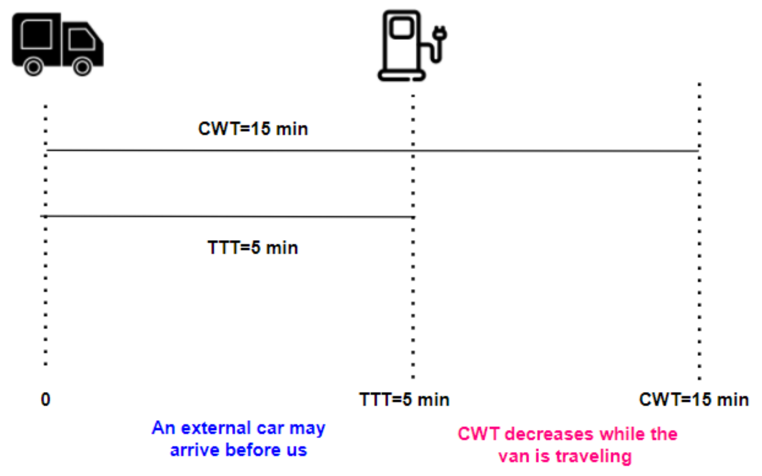
CWT-TTT: Waiting time caused by current queue. We substract TTT, because while the van is travelling to that charging station, the current waiting time will decrease by TTT

TTT/45: The expected value of the cars arriving before our van arrives. Multiplying this by **25** will give us expected increase of the current waiting time caused by external vehicles arriving

Abbreviations:

EWT: Expected waiting time
CWT: Current Waiting Times at the charging stations
TTT: Time needed to arrive a charging station
1/45: Uniform distribution of an external car arriving
25: Expected charging time of external cars

Example for Formula 1


CWT=15 min
TTT=5 min
0
TTT=5 min
CWT=15 min
An external car may arrive before us
CWT decreases while the van is traveling

EWT= 15-5+5/45*25= 12.77 minutes

Formula 1 will be used in **Condition 1, Condition 2** and **Condition 3**, all of which referring to the fact that **when the van arrives to a charging station, the CWT will be still>0**

Condition 1

```
if CWT[i]>=TTT[i]:
    EWT.append(CWT[i]-TTT[i]+TTT[i]/45*25)
```

When CWT is greater than TTT, the waiting time will not be over until our van arrives. this formula allows a very easy calculation of the expected waiting time. (like in the example above)

Condition 2

```
elif CWT[i]<TTT[i]:
    if TTT[i]-CWT[i]>=25:
        if CWT[i]/45*25+CWT[i]>TTT[i]:
            EWT.append(TTT[i]*25/45+CWT[i]-TTT[i])
```

For example:
TTT=90 CWT=60
90/45=**2** external cars will arrive until our van arrives.
60+25***2** >90
This indicates that when the van arrives to the charging station, the CWT>0, so this formula applies

Condition 3

```
elif TTT[i]-CWT[i]<25:
    if CWT[i]>45:
        EWT.append((TTT[i]*25/45)+CWT[i]-TTT[i])
```

For example:
TTT=90
CWT=80
90/45=**2** external vehicles arrive before our van
25*2+80>90
This indicates that when the van arrives to the charging station, the CWT>0, so this formula applies.

(Python Script 5.2)

CHARGING: New Charging Policy: Charging Station Selection Probability Calculations

Formula 2:

The following EWT formula will be used under condition 4:

Condition 4

```
elif CWT[i]<TTT[i]:
    elif TTT[i]-CWT[i]<25:
        elif CWT[i]<=45:
            EWT.append(CWT[i]/45*(25+CWT[i]-TTT[i]+(TTT[i]-CWT[i])/45*25)
                      +(45-CWT[i])/45*(TTT[i]-CWT[i])/45*(25-(TTT[i]-CWT[i])/2))
```

$$\text{EWT}=\frac{CWT}{45}*\left(25+CWT-TTT+\frac{TTT-CWT}{45}*25\right)+\frac{45-CWT}{45}*\frac{TTT-CWT}{45}*\left(25-\frac{TTT-CWT}{2}\right)$$

Situation A

External vehicle will arrive in the first CWT minutes (10 minutes in the example).

CWT/45: The probability of an external car arriving in CWT minutes

25+CWT-TTT: These arriving cars will increase the current waiting time by 25, in addition to CWT, but the waiting time will decrease while the van is travelling there (-TTT)

(TTT-CWT)/45: The probability of an external car arriving between the time interval [CWT,TT]

Situation B

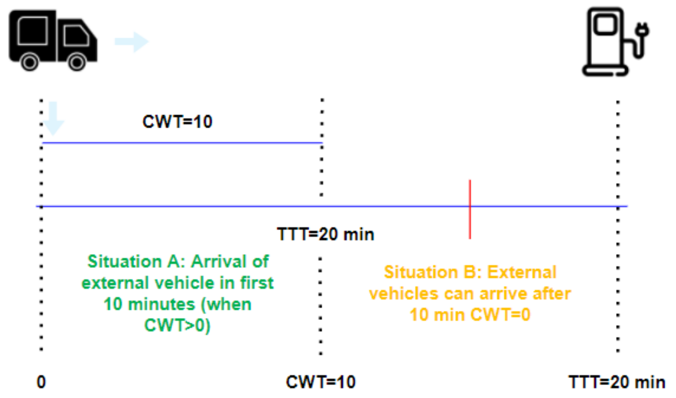
No external cars arrive in first CWT minutes (10 minutes in the example).

(45-CWT)/45: The probability of no car arriving in first CWT minutes

(TTT-CWT)/45: The probability of a car arriving in the time interval [CWT,TTT]

(TTT-CWT)/2: The expected arrival time of the external car will be at the middle of time interval [CWT,TTT]

Example for Formula 2



CWT=10
TTT=20 min
Situation A: Arrival of external vehicle in first 10 minutes (when CWT>0)
Situation B: External vehicles can arrive after 10 min CWT=0
0
CWT=10
TTT=20 min

EWT= (10/45)*(25+10-20+(20-10)/45*25) + (45-10)/45*(20-10)/45* (25-(20-10)/2) = **8.02 minutes**

Formula 3:

The following EWT formula is used under condition 5

Condition 5

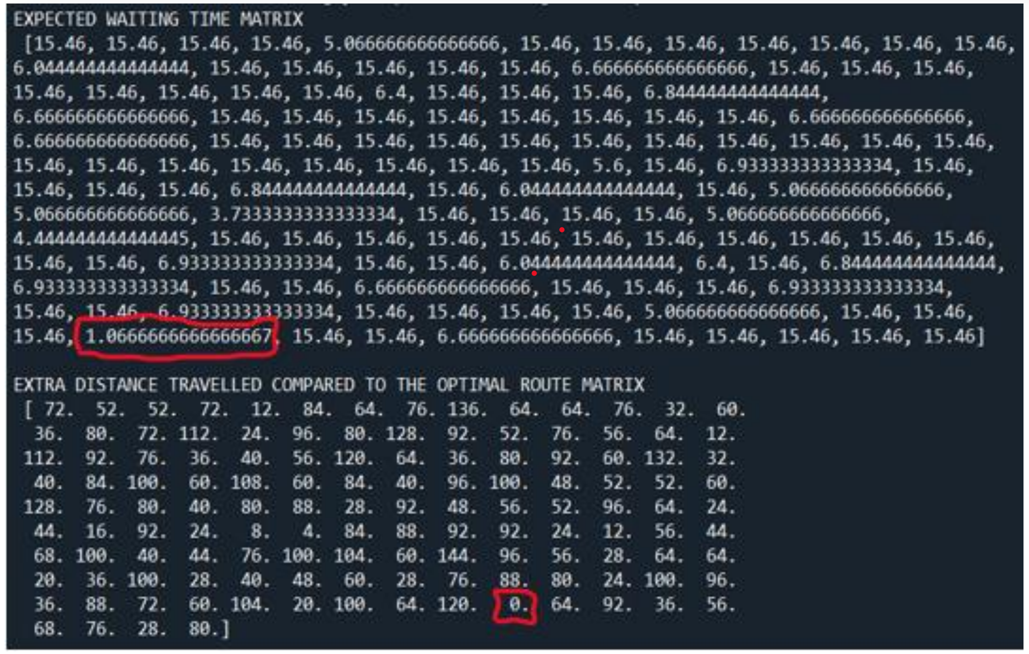
```
elif CWT<TT:
    if TT-CWT>=25:
        elif CWT[i]/45*25+CWT[i]<=TTT[i]:
            EWT.append(15.46)  # 15.46 is
```

EWT= 15.46

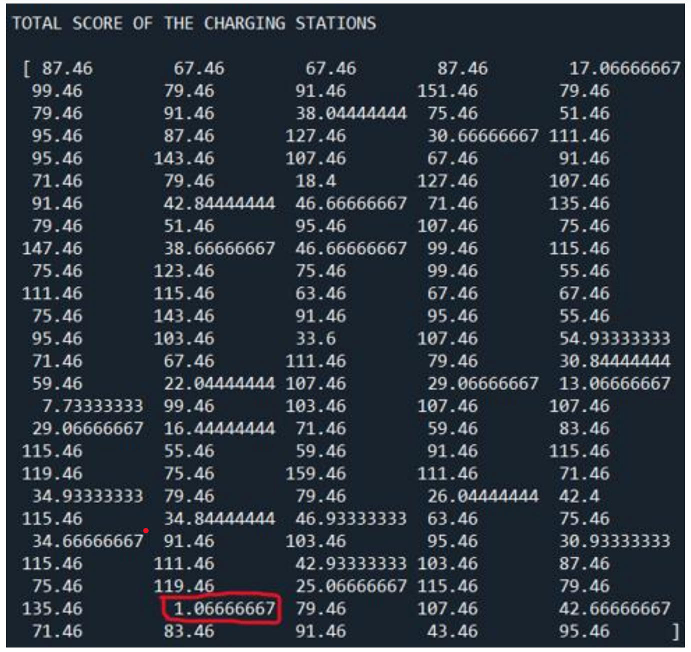
«15.46» is the average waiting time at charging stations, determined by running 1000 simulations specifically for charging stations. This value is used **only when the charging station is very far away from the van**, so that short-term calculations cannot be done, and we need an average waiting time to use as a proxy.

For example: TTT=100 min , CWT=5 min , **EWT**=15.46 min

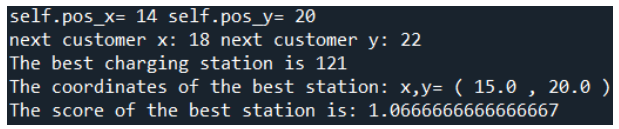
CHARGING: New Charging Policy: Charging Station Selection Output



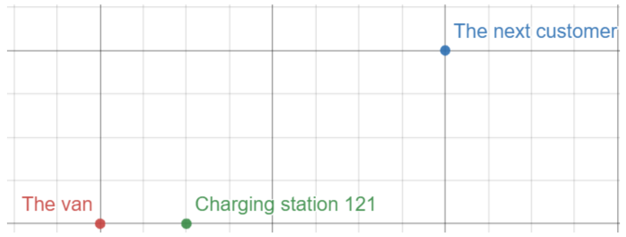
```
EXPECTED WAITING TIME MATRIX
 [15.46, 15.46, 15.46, 15.46, 5.066666666666666, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46,
6.044444444444444, 15.46, 15.46, 15.46, 15.46, 15.46, 6.666666666666666, 15.46, 15.46, 15.46,
15.46, 15.46, 15.46, 15.46, 15.46, 6.4, 15.46, 15.46, 15.46, 6.844444444444444,
6.666666666666666, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46, 6.666666666666666,
6.666666666666666, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46,
15.46, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46, 5.6, 15.46, 6.933333333333334, 15.46,
15.46, 15.46, 15.46, 6.844444444444444, 15.46, 6.044444444444444, 15.46, 5.066666666666666,
5.066666666666666, 3.7333333333333334, 15.46, 15.46, 15.46, 15.46, 5.066666666666666,
4.444444444444445, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46, 15.46,
15.46, 15.46, 6.933333333333334, 15.46, 15.46, 6.044444444444444, 6.4, 15.46, 6.844444444444444,
6.933333333333334, 15.46, 15.46, 6.666666666666666, 15.46, 15.46, 15.46, 6.933333333333334,
15.46, 15.46, 6.933333333333334, 15.46, 15.46, 15.46, 15.46, 5.066666666666666, 15.46, 15.46,
15.46, 1.0666666666666667, 15.46, 15.46, 6.666666666666666, 15.46, 15.46, 15.46, 15.46, 15.46]

EXTRA DISTANCE TRAVELLED COMPARED TO THE OPTIMAL ROUTE MATRIX
 [ 72.  52.  52.  72.  12.  84.  64.  76. 136.  64.  64.  76.  32.  60.
  36.  80.  72. 112.  24.  96.  80. 128.  92.  52.  76.  56.  64.  12.
 112.  92.  76.  36.  40.  56. 120.  64.  36.  80.  92.  60. 132.  32.
  40.  84. 100.  60. 108.  60.  84.  40.  96. 100.  48.  52.  52.  60.
 128.  76.  80.  40.  80.  88.  28.  92.  48.  56.  52.  96.  64.  24.
  44.  16.  92.  24.   8.   4.  84.  88.  92.  92.  24.  12.  56.  44.
  68. 100.  40.  44.  76. 100. 104.  60. 144.  96.  56.  28.  64.  64.
  20.  36. 100.  28.  40.  48.  60.  28.  76.  88.  80.  24. 100.  96.
  36.  88.  72.  60. 104.  20. 100.  64. 120.   0.  64.  92.  36.  56.
  68.  76.  28.  80.]
```



```
TOTAL SCORE OF THE CHARGING STATIONS

 [ 87.46        67.46        67.46        87.46        17.06666667
  99.46        79.46        91.46       151.46        79.46
  79.46        91.46        38.04444444  75.46        51.46
  95.46        87.46       127.46        30.66666667 111.46
  95.46       143.46       107.46        67.46        91.46
  71.46        79.46        18.4        127.46       107.46
  91.46        42.84444444  46.66666667  71.46       135.46
  79.46        51.46        95.46       107.46        75.46
 147.46        38.66666667  46.66666667  99.46       115.46
  75.46       123.46        75.46        99.46        55.46
 111.46       115.46        63.46        67.46        67.46
  75.46       143.46        91.46        95.46        55.46
  95.46       103.46        33.6        107.46        54.93333333
  71.46        67.46       111.46        79.46        30.84444444
  59.46        22.04444444 107.46        29.06666667  13.06666667
   7.73333333  99.46       103.46       107.46       107.46
  29.06666667  16.44444444  71.46        59.46        83.46
 115.46        55.46        59.46        91.46       115.46
 119.46        75.46       159.46       111.46        71.46
  34.93333333  79.46        79.46        26.04444444  42.4
 115.46        34.84444444  46.93333333  63.46        75.46
  34.66666667  91.46       103.46        95.46        30.93333333
 115.46       111.46        42.93333333 103.46        87.46
  75.46       119.46        25.06666667 115.46        79.46
 135.46         1.06666667  79.46       107.46        42.66666667
  71.46        83.46        91.46        43.46        95.46       ]
```



```
self.pos_x= 14 self.pos_y= 20
next customer x: 18 next customer y: 22
The best charging station is 121
The coordinates of the best station: x,y= ( 15.0 , 20.0 )
The score of the best station is: 1.0666666666666667
```



Charging Station Score = Expected Waiting Time + Extra Travel Time= 1.06667 + 0 = 1.06667

Lets check if the calculation is correct:

- **Extra Travel Time** ***(Slide 9 for example)***

Charging station 121 location: (15,20)
Vans location: (14,20)
Next Customer's location: (18,22)

Due to the charging station being located in between the van's location and the next customer's location, the extra travel time will be 0 , which is corrrect.

- **Expected Waiting Time** ***(See slide 12 for the formula and an example)***

CWT= 0
TTT=((15-14)+(20-20))/(30km/h)= 2 minutes
EWT= 2/45*(25-(2-0)/2)=1.06667

Conclusion

The van will go to **Charging Station 121,** which has the **lowest** charging station score= 1.06667

In this way, the advantage will be for the van:

- Travel less extra distance
- Wait less time in the queue

Even though the stochastic nature of the external cars' arrival to the charging stations determines the real waiting times, this calculations helps making the best decision the minimize it. The performance of this new policy minimizes the waiting times at the queues, which is demonstrated **statistically in slide 16**.

CHARGING: New Charging Policy: Charging Amount

Python Script Section 4.2

```
Tot_distance=[]
```

```
Tot_distance.append(route_distance)
```

When the routings are defined, total distances are recorded

Python Script Section 10

```
#Creating vans:
for i in range(self.num_vans):
    a = Van("Van_"+str(i), self,van_routes[i],Tot_distance)
    self.schedule_vans.add(a)
```

In the class GreenModel, when the vans are created, total distance data is passed to all the vans.

Python Script Section 8

```
def __init__(self, unique_id, model,routing,distance)
```

```
self.needed_energy = needed_energy(distance[int(self.id_van[4:])],self.battery_size)
```

Self.needed_energy is calculated by using needed_energy function, using the distance, and battery size data specific to each van. This is the **extra energy needed**, which is the amount of energy exceeding the battery size of a van

Python Script Section 5.6

```
def needed_energy(distance,battery_size):
    e = 0.218 + 1.359 / avg_speed - 0.003*avg_speed + 2.981*(10**(-5))*avg_speed**2
    travel_energy = distance*e*safety_energy_level
    needed_energy=0
    if travel_energy-battery_size>=0:
        needed_energy=travel_energy-battery_size
    else:
         needed_energy=0
    return needed_energy
```

```
safety_energy_level=1.15
```

The energy is calculated by the formula provided in the base case, and using the distance data coming from the routing. Additionally, safety_enery_level is added, taken into account stochastic speed and its effect on the total energy needed.

Python Script Section 8.5

```
if self.needed_energy>0: # standard cha
    charging_size = self.needed_energy
```

In the «check_charge» function of the class Van, the charging size is set to self.needed_energy

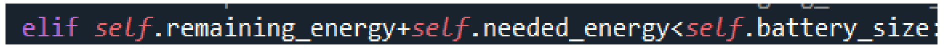
The Benefits of This New Policy

- **Minimize the number of visits to charging stations:** By measuring the needed energy in advance (depending on the route distance determined at the start of the tour), the van will charge its battery only once with the amount of needed energy. This prevents the van from making multiple visits to charging stations.
- **Prevent excessive charging time:** By measuring the needed energy in advance (depending on the route distance determined at the start of the tour), the van will charge battery more than it needs. In this way, it will save time.

CHARGING: New Charging Policy: When to Charge?

a) When to start scouting for a convenient charging station? (Early Charging Policy)

Python Script Section 8.3.1.2

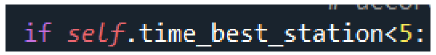


```
elif self.remaining_energy+self.needed_energy<self.battery_size:
```

When there is **enough space** in the battery to charge extra needed energy, start searching for good alternatives, depending on the **charging_station_score** (explained in slide 10)
For example:
Total travel energy needed= 28 kwh
Battery size= 25kwh
Extra needed energy= (Total travel energy needed - Battery size) * safety = (28-25)*1.15 = 3.45 kwh
When battery level<=battery size-extra energy need=25-3.45= **21.55**
So, start searching after the battery level decreases below **21.55** kwh
NOTE: This policy allows the van to **early charge**, so before the battery is at the critical level.

b) Postponement of the charging station selection decision

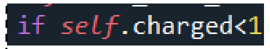
Python Script Section 8.3.1.2



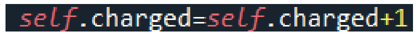
```
if self.time_best_station<5:
```

After the van starts to calculate charging_station_score of different charging stations, if all the scores are greater than 5 (meaning that there is no good alternative), the van postpones its decision of charging, until «check_charge» function of the class van is run again.
NOTE: When the van has very little battery energy left, it will choose the best station even if all the scores are higher than 5. This is explained in section **c)** of this slide.
NOTE: The threshold of «5» can be changed as a managerial decision.

Python Script Section 8.3.1.2



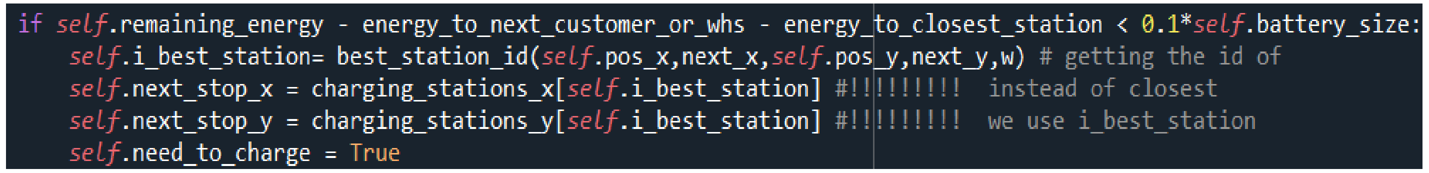
```
if self.charged<1
```



```
self.charged=self.charged+1
```

self.charged indicates if the van is early charged or not. When early charge is done, this value becomes 1, and no more early charge is done.

c) When the van reaches the critical battery level



```
if self.remaining_energy - energy_to_next_customer_or_whs - energy_to_closest_station < 0.1*self.battery_size:
    self.i_best_station= best_station_id(self.pos_x,next_x,self.pos_y,next_y,w) # getting the id of
    self.next_stop_x = charging_stations_x[self.i_best_station] #!!!!!!!!!  instead of closest
    self.next_stop_y = charging_stations_y[self.i_best_station] #!!!!!!!!!  we use i_best_station
    self.need_to_charge = True
```

Since our policy lets the van to postpone its decision of charging station selection, we need an additional policy in case of many postponements are done, and the van's battery level is at the critical level. Therefore, when the van reaches the critical battery level (as in the base case) it will go to a charging station with the lowest charging station score.

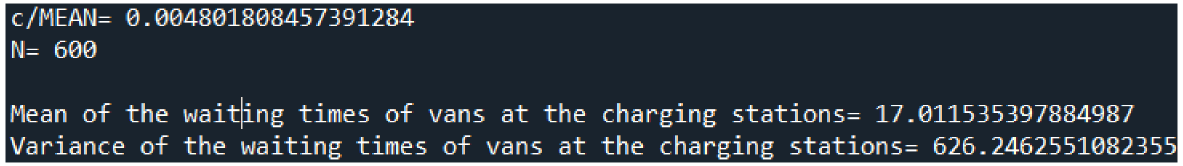
The Benefits of This New Policy

- **Early Charging Policy:** The van will have more chance to minimize the waiting time at the queues. Since it starts searching for good alternatives very early, it will have a bigger chance of finding a charging station with a lower waiting time, which will save time.
- **Postponement of the charging station selection decision:** Thanks to this policy, the van does not go to a charging station immediately. If there is no good alternative available, it postpones the decision, to find a better opportunity later on.
- **When the van reaches the critical battery level:** This policy make sures that the van will charge necessary energy if it has very little energy left. If the van postpones its decision making too many times, this policy will be a safety belt.

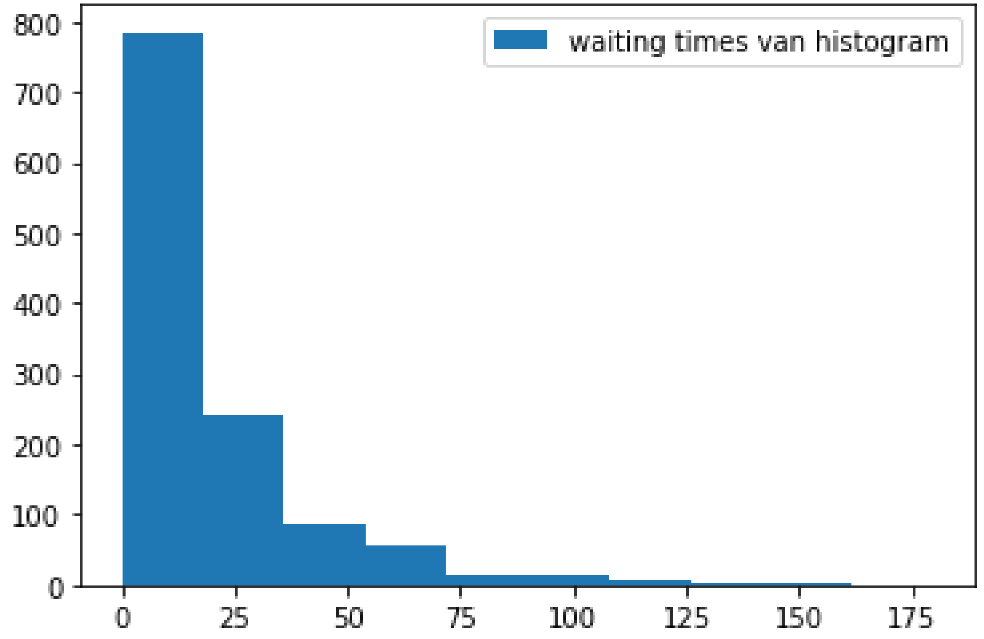
CHARGING: Base Case Charging Policy vs New Charging Policy: Performances

Having introduced the new charging policy, we should check if it improves the performance of the last mile delivery. Normally, this issue will be explained in detail in **Assignment 2;** however, you can see briefly the impact of this new charging policy below, statistically shown by running the simulation N times. Here, **the battery sizes of the vans are decreased to 25 khw,** in order to force the vans to go to charging stations, because with 35 kwh batteries the vans do not even need any charging.

Base Case Charging Policy Performance

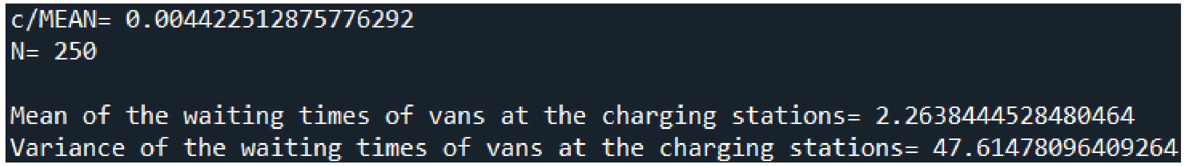


```
c/MEAN= 0.004801808457391284
N= 600

Mean of the waiting times of vans at the charging stations= 17.011535397884987
Variance of the waiting times of vans at the charging stations= 626.2462551082355
```

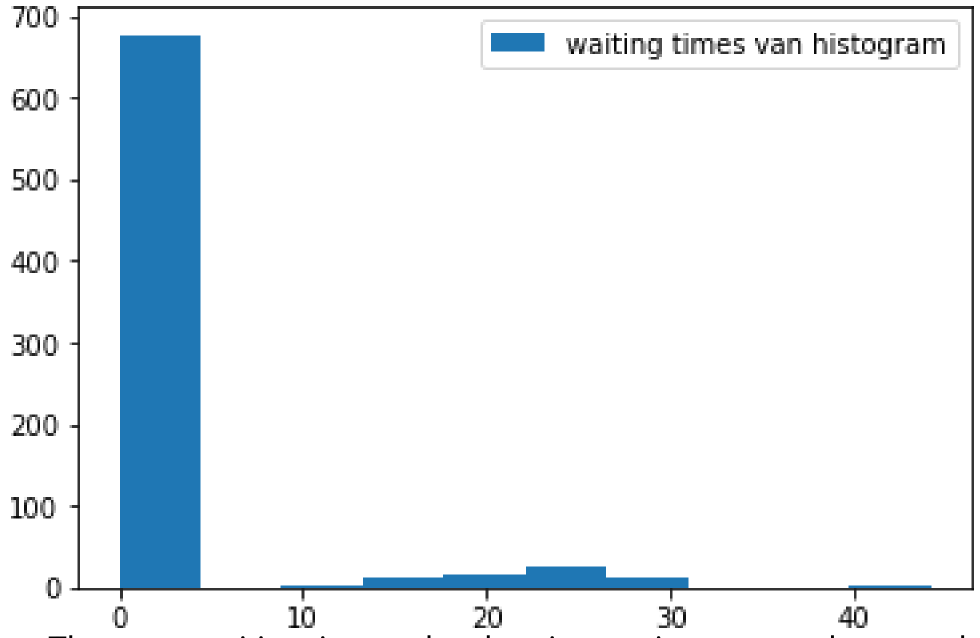


The mean waiting time at the charging station queues is 17 minutes, and more importantly, having a huge variability, occasionally causing unexpectedly huge waiting times at the queues. No doubt that this base policy is not minimizing the waiting time at the queues.

New Charging Policy Performance



```
c/MEAN= 0.004422512875776292
N= 250

Mean of the waiting times of vans at the charging stations= 2.2638444528480464
Variance of the waiting times of vans at the charging stations= 47.61478096409264
```



The mean waiting time at the charging station queues decreased from 17 minutes to **only 2 minutes,** and also with **very very low variability**. This clearly shows the contribution of this new charging policy to last mile delivery performance.

BATTERY SIZE

25 kwh 35 kwh

Base Case

```
battery_size = 35 #kWh
```

```
self.battery_size = battery_size
```

In the base case, the battery size is fixed to 35kwh. However, when using multiple vans, the vans may have different battery sizes. The company may decide using different vans with different battery sizes, due to the fact that the vans with larger batteries may cost more than the vans with smaller batteries. Therefore, the code must be adjusted to let the decision makers set different battery sizes for multiple vans.

Modified Version

Python Script Section 2.2

```
battery_size = [35,35,35,35] #kwh
```

Battery size is now an array, containing different values for different vans.

Python Script Section 8.1.1

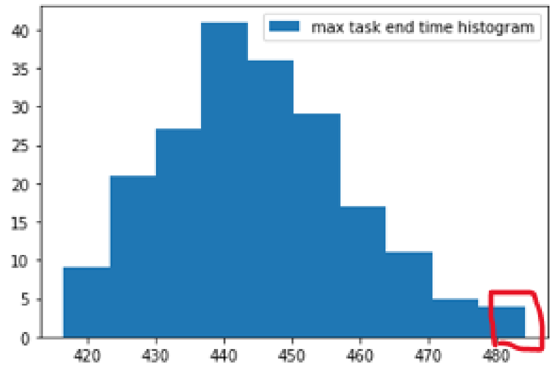
```
self.battery_size = battery_size[int(self.id_van[4:])] # [kWh]
self.remaining_energy = battery_size[int(self.id_van[4:])]
```

self.battery_size and self.remaining_energy attributes are adjusted. Int(self.id_van[4:]) is getting the id of the van, since the name of the van is «van_0».

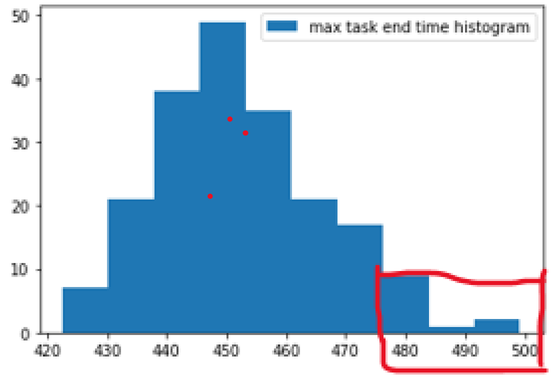
Possible Managerial Applications Deriving From Flexible Battery Sizes

In the **Assigment 2,** usage of different battery sizes for different vans will be considered deeply. However, here we would like to briefly mention how important this aspect will be. Below, you can see how the managers can identify an opportunity to save money:

Max Tour End Time of Multiple Vans
Battery Sizes:[35,35,35,35]



Max Tour End Time of Multiple Vans
Battery Sizes:[30,30,20,20]



The managers will easily identify that **even with** smaller batteries [30,30,20,20] all the vans can complete in 8 hours. This is **an opportunity** to rent **cheaper vans** with smaller batteries (of course if this is an option for the company). In this way, possible saving opportunites can be identified. A **hypothetical business case** will be solved in **Assignment 2** to adress this issue.

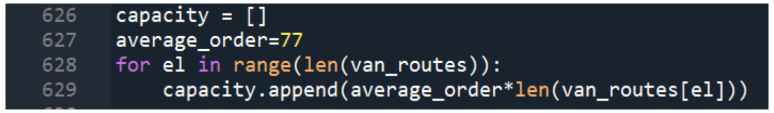
€ € €

CAPACITY OF THE VANS

As mentioned in the problem each van has a limited capacity of 1.8 ton. This results in the fact that doesn't matter the length of the routing, the company rents only a single type of van for each mission. It can be assumed that the company is renting these trucks for the problem of the assignment. In this case, this project work offers an optimization for also the renting costs of the vans. This section talks about how to calculate the required capacity for each van. With this information, analysis of the work can be done beforehand, and more suitable vans can be rented beforehand.

The cost can be optimized by calculating the load requirements of each van.
Capacity requirement of each van is calculated by the following formula and the script:

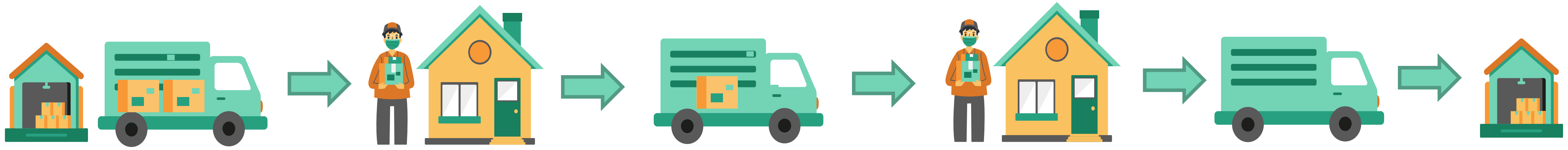
Required Capacity =
#Customers Visited * Average Order Weight

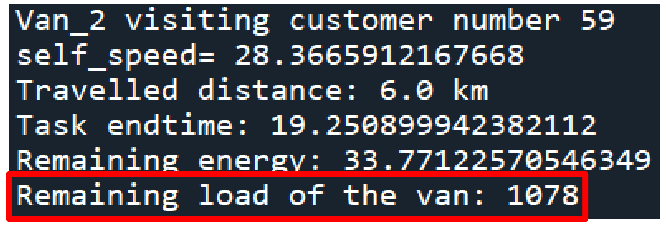


```python
capacity = []
average_order=77
for el in range(len(van_routes)):
    capacity.append(average_order*len(van_routes[el]))
```

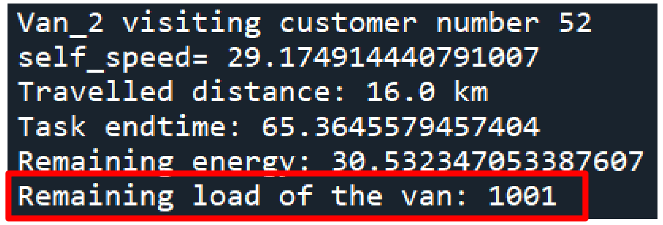
capacity: an array containing the required capacity of each van
average_order : average order weight
len(van_routes): #customers visited in each routing

Each van starts their journey with full amount of load which is equal to the required capacity. One package is reduced from the van at each customer location. At the end of the journey, the van comes back to the warehouse with a loading of 0.





```
Van_2 visiting customer number 59
self_speed= 28.3665912167668
Travelled distance: 6.0 km
Task endtime: 19.250899942382112
Remaining energy: 33.77122570546349
Remaining load of the van: 1078
```



```
Van_2 visiting customer number 52
self_speed= 29.174914440791007
Travelled distance: 16.0 km
Task endtime: 65.3645579457404
Remaining energy: 30.532347053387607
Remaining load of the van: 1001
```

STOCHASTIC SPEED

Python Script Section 8.1.2

```
self.speed=0
```

The self.speed starts from the value 0

Python Script Section 8.3

```
if stochastic_speed==1:
    self.speed=random.normalvariate(avg_speed,stdev_speed)
elif stochastic_speed==0:
    self.speed=avg_speed
```

In the «check_charge» function, the speed is updated stochastically, each time the «check_charge» function is executed.

Python Script Section 8.3

```
stochastic_speed=1  # 0: deterministic / 1: stochastic
avg_speed = 30 #km/h - Average speed
stdev_speed=2 # standard deviation of speed
```

The script allows the user to decide whether to use stochastic speed or not (stochastic_speed=1 means stochastic speed is used) also allowing the user to change the parameters of the stochastic speed (avg_speed and the standard deviation of the speed

Python Script Section 8.4

```
self.remaining_energy -= compute_energy(self.pos_x,
                                        self.next_stop_x, self.pos_y,
                                        self.next_stop_y,self.speed)
```

```
self.task_endtime += compute_time(distance_next_stop,self.speed)
```

self.speed attribute is used to compute energy and time in the move function.

Python Script Section 8.4

```
def compute_energy(x1, x2, y1, y2,speed):
    distance = compute_distance(x1, x2, y1, y2)
    e = 0.218 + 1.359 / speed - 0.003*speed + 2.981*(10**(-5))*speed**2
    energy = e*distance
    return energy
```

```
def compute_time(distance,speed):
    return (distance/speed)*60
```

2 already existing functions, compute_time and compute_energy, are modified to include speed as an input, being able to compute time and energy consumed depending on the stochastic speed.

The Impact of the Stochastic Speed on the System Performance

- **Stochastic travelling time:** The travelling time varies depending on the stochastic speed, making it harder to predict the tour end time of the vans. This stochasticity may force the company to increase the number of vans, in order to cope with unexpectly slow average speed. This actually represents the real world situation more realistically, in which the speed may vary according to traffic situation in that given city.

```
Van_3 going back to the warehouse
self_speed= 27.617164630146995
```

- **Stochastic energy consumption:** As the speed of the vans are used as an input to calculate the energy used, the stochasticity of the speed also effects the energy consumption. This may effect the energy needed to complete a tour, which may force the company to have vans with bigger batteries as a safety factor.

THE NUMBER OF VANS NEEDED

NOTE: This part will be deeply explained in the **Assignment 2;** however, here we are briefly introducing you the procedure according to which the right number of vans can be decided., since it is also required in the Assignment 1.

Objective Function: Minimize the number of vans

Constraint: All vans must complete the tour in 8 hours

Case 1 **Number of vans:** 3 **Battery Size:** 35 kWh for all vans **Charging Policy:** The new charging policy **Speed:** Stochastic

Step 1: Data Collection

The first step is to **collect the data** of **the maximum of the tour end times** by running multiple simulations:

Tour End Times of a run:

Van_0: **574**
Van_0: 552
Van_0: 439

The recorded value will be **574 minutes,** which is the highest of the all tour end times. Because, our main goal is to have all our vans complete the tour in 8 hours, not the average van.

Python Script Section 7.3

```
task_end_time=[]
max_task_end_time=[]
```

Python Script Section 8.4

```
if self.pos_x==warehouse_x and self.pos_y==warehouse_y:
    task_end_time.append(self.task_endtime)
```

Python Script Section 8.4

```
maximum=max(task_end_time)
max_task_end_time.append(maximum)
```

The Output

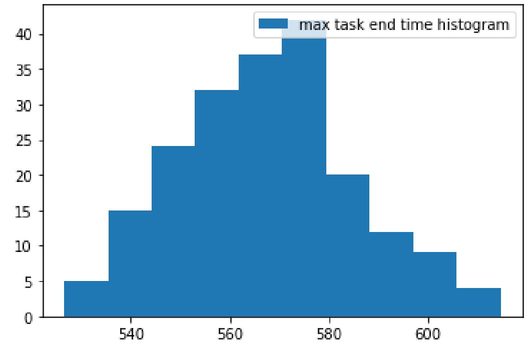
```
Tour end times of the vans:
 [439.26095918753583, 552.3171391878694,
574.3930744673742]

Max tour end time is:
 [574.3930744673742]
```

Step 2: Calculating Statistics

The statistics of **max tour end times** are calculated.

```
c/MEAN= 0.004359196410598535
Number of runs= 200
```

```
**Max Tour End Time Statistics***
Mean= 567.4366434622311
Sample variance= 314.6900022838198
Variance of the sample mean= 1.573450011419099
P(max throughput time>480)= 0.9999995865785468
```



Assuming a normal distribution and making calculations, in 0.999 of the cases 3 vans are **not enough** to satisfy 8 hours constraint

Step 3: Decide Number of Vans

The decision maker will decide if the probability of exceeding 8 hours is acceptable or not. In this case, **3 vans are not enough** to satisfy this 8 hours constraint.

Step 4: Try Different Alternatives

Since 3 vans are not enough to satisfy 8 hours constraint, a different setting must be tried.

In the next slide, 4 vans will be evaluated.

Case 1 Conclusion:

3 vans are **not enough** to satisfy 8 hours constraint
4 vans will **be tried** in the following slide

THE NUMBER OF VANS NEEDED

NOTE: This part will be deeply explained in the **Assignment 2;** however, here we are briefly introducing you the procedure according to which the right number of vans can be decided., since it is also required in the Assignment 1.

Objective Function: Minimize the number of vans

Constraint: All vans must complete the tour in 8 hours

Case 2

Number of vans: 4

Battery Size: 35 kWh for all vans

Charging Policy: The new charging policy

Speed: Stochastic

Step 1: Data Collection

The first step is to **collect the data** of **the maximum of the tour end times** by running multiple simulations:

Tour End Times of a run:

Van_0: **452**
Van_0: 428
Van_0: 426
Van_0: 391

The recorded value will be **452 minutes.**

Python Script Section 7.3

```
task_end_time=[]
max_task_end_time=[]
```

Python Script Section 8.4

```
if self.pos_x==warehouse_x and self.pos_y==warehouse_y:
    task_end_time.append(self.task_endtime)
```

Python Script Section 8.4

```
maximum=max(task_end_time)
max_task_end_time.append(maximum)
```

The Output

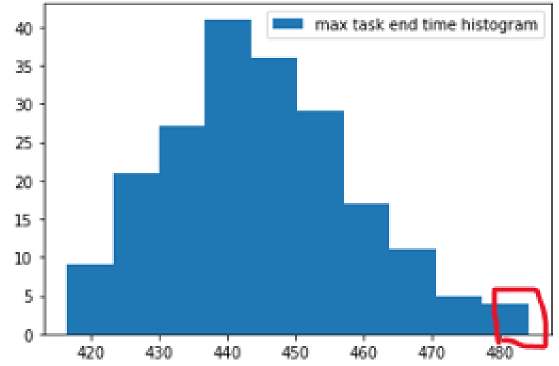
```
Tour end times of the vans:
 [391.8165474813942, 428.90330549982775,
426.57053962543944, 452.4042017514322]

Max tour end time is:
 [452.4042017514322]
```

Step 2: Calculating Statistics

The statistics of **max tour end times** are calculated.

```
c/MEAN= 0.004349490234387911
Number of runs= 200
```

```
**Max Tour End Time Statistics***
Mean= 445.08792434253513
Sample variance= 192.75411513897564
Variance of the sample mean= 0.9637705756948782
P(max throughput time>480)= 0.005957892790837027
```



Assuming a normal distribution and making calculations, in 0.995 of the cases 4 vans are enough to satisfy 8 hours constraint

Step 3: Decide Number of Vans

The decision maker will decide if the probability of exceeding 8 hours is acceptable or not. In this case, **4 vans** are **perfectly enough** to satisfy it.

Step 4: Try Different Alternatives

With the given battery sizes (35 kwh for all vans), 4 vans are easily satisfying the 8 hours constraint. In this case, the **battery sizes were so large** for the vans that they **did not** even need to **go any charging stations.**

```
***Waiting Times At The Charging Station Statistics:***
The vans did not need any charging in these runs
```

Here, the managers can start thinking: «What if we use vans with **smaller** batteries, which are **cheaper**, and still can satisfy 8 hours constraint?

In the **Assignment 2,** we will solve a **hypothetical business case**, in which the managers will try to decide the battery sizes of the vans, which satisfies 8 hours tour time constraint, and also minimizing the daily cost of the fleet of the vans.

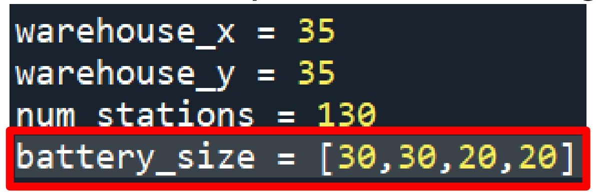
NOTE: The main contribution of our new charging policy will be seen in the Assignment 2. Here, due to 35kwh is a very large battery for the given routing, the vans do not need any charging at all.

CONCLUSION

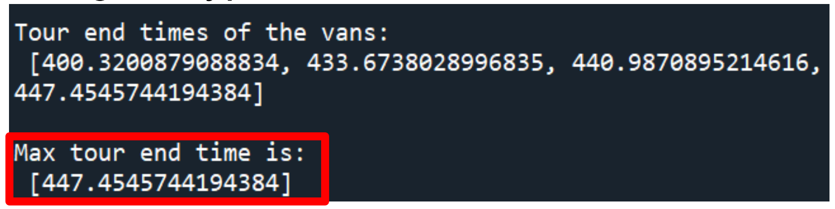
In this project work, the weakness of the given Last Mile Delivery Problem are analyzed, and possible solution methods are generated for the most important weaknesses. In the end, the problem is solved with the given constraints and the 8 hours constraint is satisfied with **four vans** with a very high confidence interval.

Yet, the given constraints might limit to reach the best results. In this sense, there are some issues that can be improved in the managerial part:

- The first aspect is that the company might need to rent a fleet of identical vans with a **battery size of 35kWh**. In this case, the constraint for a van selection will be the van with the longest routing. This might be more costly than necessary. A possible solution to reduce the cost is to customize the battery size of each van. In other words, *renting different types of vans*. The following figure represents that it is possible to reach a solution with the same number of vans in the given time limit with vans having different (and less than the base case) battery sizes.This issue will be deepened in the **Assignment 2** by solving an hypothetical business case.



```
warehouse_x = 35
warehouse_y = 35
num_stations = 130
battery_size = [30,30,20,20]
```



```
Tour end times of the vans:
 [400.3200879088834, 433.6738028996835, 440.9870895214616,
447.4545744194384]

Max tour end time is:
 [447.4545744194384]
```

- The second aspect to consider is the **total time** assigned for each routing. Some vans come back to the warehouse a bit earlier then the other ones. This creates a discrete amount of unused capacity of these vans. This can be solved by allowing some *flexibilities* in time-wise on each route or by *utilizing* these vans for other works in those unused time-slots.