

# Meal Delivery Routing

## Final Report

Louisa Crijnen, María Ármann, Peter Derksen, Sanne de  
Wilde & Sara Lute

Project Optimization of Business Processes



## Introduction

For this project, we are considering a company that delivers meals around the city center of Paris. Our objective is to find the best way to deliver the meals such that both waiting time and operational costs are minimized. With different approaches, we will compare both the waiting time and costs of the approaches to find the best one.

Our first goal is to create a minimal viable product. When our minimal viable product is working we will be making improvements until we have a realistic product for the user. In this report, we will go through it step by step.

## Python Objects

Throughout the project, the focus will be on four objects: Orders, OrderBundle, Restaurants and Vehicles. These attributes will be referred to later in the report. All four have been made into an object in Python with the following attributes and functions.

- Vehicle:
  - vehicleId (int): We have 5 different types of vehicles, each type is assigned a ID number between 0 and 4.
  - capacity (int): How many meal boxes fit in the vehicle.
  - fixedCosts (int): Fixed cost of using the vehicle.
  - hourCosts (int): Hourly cost for using the vehicle.
  - kmCosts (int): Cost per kilometer.
  - allowedOnHighspeedRoads (boolean): Some vehicles are allowed on high speed roads, while others are not.
  - assignedOrder (order object): The order the vehicle is currently driving to the delivery location is assigned to the vehicle.
  - fulfilledOrders (list of objects): stores all the orders that were assigned to the vehicle on a day.
  - assignedBundle (orderBundle object): The bundle the vehicle is currently driving to the delivery location is assigned to the vehicle.
  - fulfilledBundles (list of objects): stores all the bundles that were assigned to the vehicle on a day.
  - travelledDistance (int): all distances that are travelled by the vehicle are added to this integer value.
  - isAvailable (function, returns boolean): this function takes in variables: currentTime, order and the time it takes for the vehicle to drive from its current location to the new order's location. By looking at our current time and the new upcoming order we can see if our vehicle is free to pick up the new order at the time it's ready.
- Restaurants:

- restaurantId (int): The ID number of the restaurant.
  - nearestNode (tuple): Using the lat and lon of the restaurant to find a node on the Paris Graph that is closest to the restaurant. This node will then be used to find the shortest path to or from the restaurant.
  - parallelDish (dict): How many dishes the restaurant can make parallel to each other. This will be a dictionary where the key is the type of meal and the value will be how many of that meal the restaurant can make parallel to each other.
  - cookingTime (dict): Cooking time. This will be added to the order time to find out the exact time when the order should be picked up. This will also be a dictionary where the key is the meal type and the value are the minutes.
  - openingHours (dict): Opening Time. In Hours. A dictionary where the key is the weekday and the value is the opening hour.
  - closingHours (dict): Closing time. In Hours. A dictionary where the key is the weekday and the value is the closing hour.
  - districtId (int): The ID number of the district that the restaurant is located in.
- Orders:
    - orderId (int): Order ID for all orders in a day.
    - nearestNode (tuple): Using the lat and lon of the order destination to find a node on the Paris Graph that is closest to the order destination. This will then be used to find the shortest path for our delivery vehicle.
    - orderTime (int): The time the order was placed. In seconds from start of the day 00:00.
    - orderRestaurant (int): the ID of the restaurant the order was placed with.
    - orderAmount (dict): A dictionary where the value is the meal type and the value is how many of that type were ordered.
    - orderSize (int): Summing up all the values in the dictionary orderAmount we get the total order size. This value will be used to determine how big the delivery vehicle should be.
    - productionTime (int): The cooking time of the order. This value will be added to the order time to get the pickup time for the delivery vehicle.
    - travelDistance (int): The shortest path from the restaurant to the delivery location in meters.
    - travelDuration (int): The time it takes the vehicle to deliver the order from the restaurant to the delivery location by choosing the shortest path. Given in seconds.
  - OrderBundle
    - orderBundleId: Each bundle has its own ID.
    - startTimeBundle: The time of the first pick up.
    - ordersInBundle (dict): A dictionary keeping track of all the orders that are in the bundle.

- `orderBundleSize` (int): Count of how many orders are within the bundle.
- `ordersIdSet` (list): A list of all order IDs within the bundle.
- `addOrder` (function): A function that adds an order to the bundle.
- `timeWithin` (function, returns boolean): checks if order time is within a certain time of the first order time in the bundle.
- `amountOrders` (function, returns int): adds up the sizes of all orders in a bundle.
- `splitBundle` (function, returns two dictionary): splits the bundle into two bundles.

## Minimal Viable Product

To set up the Minimal Viable Product we had an extensive brainstorm to determine what we wanted to include in the product. We approached this problem by first evaluating the assumptions we need to make for each variable to simplify the problem. Our goal for this part of the project is to get a working model, thus to have set up a model in which all orders are delivered from the chosen restaurant to the appropriate destination. This will in turn result in a variety of costs linked to the vehicles and a waiting time made up of both the production time and the delivery time. The following assumptions are made:

- a) All roads are the same, thus there is no distinction between high-speed and low-speed roads. This means all vehicles can drive on all roads.
- b) Every order has its own vehicle and there are infinite number of vehicles. For each order, there is a vehicle that brings it to the respective destination. This vehicle will be waiting in front of the restaurant, thus there is no travel time to the restaurant, but it can leave immediately when the order is ready.
- c) The type of vehicle is based solely on the capacity needed for the order.
- d) All orders are known at the start of the day.
- e) Restaurants have infinite capacity to prepare food, thus when an order arrives it can be prepared immediately and will only take the cooking time until the order is ready for delivery.

With these assumptions and the created objects as mentioned above, we started to set up our algorithm. At first, we set up methods to read the given CSV files and assign the appropriate variables to the appropriate objects. Then we continued to write an algorithm that would assign both the restaurants and the order destinations to the nearest node in the Paris map, thus enabling us to use the nodes in the map for the shortest path problem.

Furthermore, we wrote a method that assigns an appropriate vehicle to the order. This method considers the size of the order and assigns the order to the vehicle with the smallest sufficient capacity.

After having set up our basic situation, where each restaurant has been assigned a node, each order destination is assigned a node and each order is assigned a vehicle. We can calculate the

shortest path for each order. With this, we calculate the costs of each order based on the given fixed and variable costs for each appropriate vehicle, as well as the waiting time for each order.

These elements make up our minimal viable product, which in turn are linked to our GUI. In our GUI we ask the user to upload a CSV containing all orders for a day, which will be given as input to the MVP above. After the file is submitted, the GUI will show certain metrics, such as average waiting time, the total costs, and the average costs per order.

## Improvements

We have thought of various ways in which we can improve our minimal viable product. In this report, we included the improvements that we considered to make the outcome as realistic as possible. The most significant improvements, and those that we find most essential to consider, are linked to the vehicles.

### 1 Single order per vehicle

First of all, we want to consider the situation where we do not assume the vehicles disappear after having delivered their order. For this, it is essential that we keep track of where the vehicles are at each point in time and we have to keep track of whether the vehicles are available. In this situation, we will write a method that will send a vehicle, when available, to the nearest possible order that they can pick up. In this way, we model a more realistic vehicle situation and at the same time, we can minimize the time the vehicles are idle and minimize the number of vehicles. This will in turn lower the costs since the fixed costs will significantly decrease.

#### 1.1 Estimation

In the next sections, the improvements for assigning vehicles will be discussed. For these improvements, we aim to find out when a vehicle is available, and later on where the closest vehicle is. To ensure a lower run time, we used an estimate for the distances of these vehicles. To do so, we calculated the point-to-point distance between all nodes. Since we know the latitude and longitude for each node we can calculate the point-to-point distance by using the Haversine formula. This distance will always be smaller than the distance calculated via Dijkstra's Algorithm. We therefore also introduced a correcting factor to get a better estimation. We estimated this factor by calculating both the shortest-path distance and point-to-point distance and calculating the factor between them respectively. This resulted in a correcting factor, rounded, to 1.5. So on average the shortest-path distance is 1.5 times longer than the point-to-point distance.

#### 1.2 Non-disappearing vehicles: First available vehicle (FAV)

To minimize cost, as well as make the approach more realistic compared to the MVP, the first step will be reusing the same vehicles over and over again. This means that instead of having a new vehicle for every order, that appears at the pick-up location and disappears after the drop-off, the vehicle drives to a new pick-up location after it has delivered its previous order. By doing this

the number of vehicles needed for each day is decreased. The fixed cost for a new vehicle is fairly high, so by decreasing the number of vehicles needed, the operational cost will be lowered.

### 1.2.1 Algorithmic approach

For the First Available Vehicle (FAV) Algorithm, we set the vehicle type for every vehicle to type 5 (with the largest capacity), to ensure that every single order fits. For this algorithm, we keep track of all the operating vehicles in a dictionary *allAssignedVehicles*. We fill this dictionary by looping through the orders, starting with the first order of the day, and perform the following steps to assign vehicles to the orders:

- a) If the order is the first one of the day, We assign the order to a new vehicle and add the vehicle to the *allAssignedVehicles* dictionary.
- b) For all other orders, we loop through the vehicles in the dictionary and check if it is available. To do so, we determine a vehicle to be available if it can arrive at the pick-up location of the concerning order at the time the order is ready (for this it is assumed that the vehicle stays at the last order's drop-off location until it will drive to a new pick-up location). This means that there will be no waiting time besides the production time of the meals and the shortest driving time between the restaurant and the order. To estimate the travel time from the vehicle's last drop-off location to the new pick-up location, the travel time estimation as described in section 1.1 is used, to decrease the run time.
  - If a vehicle is available, the order is added to the vehicle and added to the list of fulfilled orders. This means that if there are multiple vehicles available, the first one found in the dictionary is chosen, hence 'First Available Vehicle'.
  - If no vehicle is available, a new vehicle of type 5 is added to *allAssignedVehicles*.

## 1.3 Non-disappearing vehicles: Closest available vehicle (CAV)

Another approach to the version of the non-disappearing vehicle is taking the closest available vehicle instead of the first available vehicle. This will in turn decrease the total distance. With this, the cost can be further minimized as cost per kilometer is included in the total operational cost.

### 1.3.1 Algorithmic approach

This approach is very similar to the previous one. The largest difference is that instead of taking the first available vehicle, from our dictionary of vehicles, we take the closest available vehicle. This is done by sorting the dictionary *allAssignedVehicles* in terms of the distance between that vehicle's last drop-off location and the order's pick-up location (with the estimated travel distance as described in section 1.1). Using this sorted dictionary, we loop again through the vehicles to find the closest available vehicle, if any.

## 1.4 Non-disappearing vehicles: Best available vehicle (BAV)

In the previous approaches, we set the vehicle type of each assigned vehicle to 5 (since this would ensure that every single order could fit). However, the fixed and variable costs for vehicle type 5 are a lot higher than for the lower vehicle types. Therefore, it could make a big difference to select cheaper vehicle types when possible.

### 1.4.1 Algorithmic approach

For the Best Available Vehicle (BAV), instead of choosing vehicle type 5 when adding new vehicles, the vehicle type is chosen based on the order size and the capacity of the vehicle. Moreover, when checking which vehicles are available, we also check whether the vehicle fits the order. If there is no vehicle available with enough capacity, a new vehicle is created dependent on the size of the order. If there are multiple vehicles that are available and have enough capacity, the distance and size of the vehicle are not (yet) taken into account. The next section will discuss both improvements together. In this case, vehicle 3 never gets chosen, since it has the same capacity at vehicle 2 but is a later choice. We have chosen not to account for this and keep the algorithm this way since the fixed costs of vehicle 3 are lower, while the km costs are lower for vehicle 2. In the long run, therefore, it is optimal to choose vehicle 2.

## 1.5 Non-disappearing vehicles: Best and closest available vehicle (BCAV)

The last version of the non-disappearing vehicle improvement is the one combining the previous ones. This approach results in the largest improvement from the MVP so far.

### 1.5.1 Algorithmic approach

The approach for the Best and Closest Available Vehicle (BCAV) is similar to the ones described above but includes both improvements. While looping through all the orders of the day, the first order is assigned a new vehicle with the first vehicle type that fits the order. For the next orders, the dictionary of vehicles is sorted in terms of their distance to the order location. Then, of these vehicles, it is checked whether it is available and whether that order fits into the vehicle. If this is not the case, a new vehicle is added with the first vehicle type that fits that specific order.

## 2 Bundled orders

Up until now, it has been assumed that all vehicles only deliver one order at a time. The next improvement is to give the user an option of combining orders. Combining, or bundling, orders might result in more larger vehicles, but fewer vehicles in total. This might also minimize the costs, by decreasing the total amount of travelled kilometers. This approach might lower the total operational cost but increase the overall waiting time of the delivery. This trade-off will be left up to the user.

For this part, we want to base our choice of whether to combine orders or not on two things. Firstly we want to look at the time between orders and make a threshold. The threshold would

be the maximum number of minutes our vehicle waits for the next order. Secondly, we want to make sure that our vehicle only combines orders if their pick-up and delivery locations are not too far apart. To do this we want to divide our Paris map into districts so each vehicle only combines orders if they will be delivered to locations within the same district and/or neighboring districts and if they will be picked up from locations within the same district and/or neighboring districts.

## 2.1 Bundling the orders

The bundles are created based on input from the user. These inputs include information about the time frame all orders in the bundle should be picked up within and the distance between the different pick-up locations.

When writing the algorithm that creates the bundles three different functions were written. The first one is the easiest one where we only bundle orders together if they are placed on the same day, at the same restaurants, and if their delivery locations are within the same region. These regions are either single districts of the city of Paris or bundled neighboring districts. The decision of how many districts are bundled together is based on a radius parameter. The distance between the center of two districts has to be within this radius for them to be bundled together. For this approach, the time of the orders is not considered. This version resulted in a lot of orders being bundled together that are placed many hours apart. This is not ideal.

The second version is the first improvement from the first one. In this second version, both distances between the pick-up locations and distances between the drop-off locations are taken into account. This means that within the same bundle, orders can be picked up from multiple restaurants. Again, like before, the restaurant regions and drop-off regions are based on a radius parameter. The radius for the restaurant regions is an input for the user. The user can determine a maximum radius in the GUI, but it has a default value of 1 km. The delivery district has a fixed radius of 500 meters. Both default values were chosen out of convenience when bundling orders. By trying multiple values for the radius parameters these numbers resulted in well balanced bundle sizes. Even though this is the case, the user can still have an effect on this by choosing it's own radius for the restaurant regions (bundling together more restaurants).

The final and last version is the one that is used in the algorithm behind the GUI. This version is the most improved one and also the most realistic one. Now the time of the orders is taken into account. The user can determine a time frame such that all orders within a bundle have to be picked up within that time frame. If an order has a pick-up time outside of that time frame, it will not be included in the bundle. The default value for the time frame is 10 minutes.

In all versions, it is made sure that no bundle sizes exceed the maximum capacity of the largest vehicle. This way we know that all bundles fit in a vehicle.



### 2.1.1 Algorithmic approach: Same restaurant

To keep track of the bundled orders, the object `orderBundle()` is used. To begin with we need to create the delivery regions. The delivery regions are created by bundling the districts together based on the radius parameter mentioned before. When bundling the districts together we loop through the districts. If the current district has already been assigned to a region we continue to the next one. If it has not been assigned to a region we find all of its neighbouring districts based on the radius parameter. All neighbouring districts that have not been assigned already will be bundled together with the current districts into a region. This is done until all districts have been assigned to a region.

By looking at one region at a time we create all bundles within that region. This is done for all regions. Within the loop over all regions we have a loop that runs through the list of all orders. All orders are in ascending order of order time. Inside the loop there are two checks:

- a) First of all we use the location attribute, called *nearestNode*, of the current order to check if the delivery location of that order is located within the region we are looking at. If yes, we do further checks, but if not we go straight to the next order in the list.
- b) For all orders that belong to the region we are looking at, we have a inner for loop that loops through all current bundles. If the restaurant of the first order in some bundle is the same as the restaurant for the order we are looking at, we append our order to that bundle. If no bundle fits that criteria we create a new bundle within our region and append our current order to that bundle.

When we have looped through all of the regions all orders should belong to a bundle. The algorithm returns a dictionary of dictionaries. The key of the outer dictionary is the region number with the value being a dictionary of all bundles within that region. The dictionary looks as follows:

```
T {0: {0: <bundle.object >, 1: <bundle.object >}, 1: {0: <bundle.object >, 1: <bundle.object >, 2: <bundle.object >}, 2: {0: <bundle.object >}....}
```

This example shows how the dictionary would look for the first three regions. In this example region 0 has two bundles, region 1 has three bundles and region 2 has one bundle. Each bundle has one or multiple orders.

### 2.1.2 Algorithmic approach: Restaurant district

*Restaurant district* is an improved version of the *Same restaurant version*. In this case, the orders may belong to different restaurants and still be bundled together. For this version, we look at the distance between the restaurants in the same way we look at the distance between the delivery locations.

Again, the outer loop loops through all regions while the inner loop loops through the list of all orders. Now the checks are as follows:

- a) First of all we use the location attribute of the current order to check if the delivery location

of that order is located within the region we are looking at. If yes, we do further checks, but if not we go straight to the next order in the list.

- b) For all orders that belong to the region we are looking at, we have an inner for loop that loops through all current bundles. If the restaurant of the first order in some bundle is in the same region as the restaurant for the order we are looking at we append our order to that bundle. If no bundle fits those criteria we create a new bundle within our region and append our current order to that bundle.

This version gives more flexibility for bundling orders, resulting in more orders being bundled together compared to the previous version.

### 2.1.3 Algorithmic approach: Restaurant district & Time

This last version of the algorithm is a further improvement from the *Restaurant district* one. This means that the algorithmic approach is the same, but the checks are different. We again loop through all the districts in the outer loop and the orders in the inner loop. In this version the time frame is added into the checks. The checks in this last and final version are as follows:

- a) The first check is always checking for the delivery location of the order. This is a criteria in all three versions. We use the location attribute of the current order to check if the delivery location of that order is located within the region we are looking at. If yes, we do further checks, but if not we go straight to the next order in the list.
- b) For all orders that belong to the region we are looking at, we have an inner for loop that loops through all current bundles. If the restaurant of the first order in some bundle is in the same region as the restaurant for the order we are looking at we append our order to that bundle. If no bundle fits that criteria we create a new bundle within our region and append our current order to that bundle.
- c) In this version an extra check is implemented where we check whether the order time of the current order falls within the time frame given. Now within the previous loop, where we check whether the restaurants are located in the same region, we keep track of the order time of the first order in the bundle. If the order time of the current order is within the time frame from the first order, it is added to the bundle. Otherwise it is added to a new bundle.

This algorithm greatly increases the number of bundles and decreases the average bundle size compared to the previous one.

## 2.2 Assigning vehicles to bundles

For assigning vehicles to the bundles created, the four different approaches from before are again used; First Available Vehicle, Best Available Vehicle, Closest Available Vehicle, and Best and Closest Available Vehicle. Before running the algorithms, the bundles are sorted on order time of the first order in the bundle. When sorting the bundles like this, the algorithms work very similarly to the single orders. Firstly, the correct pick-up and drop-off order of the orders within the bundle needs to be determined. In this case, we assume that the vehicle first performs all the pick-ups,

and then all the drop-offs. With this route, to check the availability of a vehicle and to find the closest vehicle, the last drop-off location of the previous bundle and the first pick-up order of the corresponding bundle are used.

## Results

The results show the differences between the performance of the different approaches. We have chosen to evaluate the total distance, waiting time, and the number of vehicles. These are all essential elements when it comes to costs and customer satisfaction. To keep the customer satisfied, we have to try to minimize the waiting time, but to keep the manager happy we want to minimize the costs. To minimize the costs we want to minimize the number of vehicles and total distance travelled. In the table below you see the results for these metrics for the given algorithms. These results are for February 20th, 2021, the largest file, with no vehicle constraints and the default values chosen.

Algorithm	Single Order	Bundles
FTF	Number of Vehicles: 3146 Total Distance: 22137	Number of Vehicles: 2969 Total distance: 22520 km
FAV	Waiting time: 20.31 Number of Vehicles: 227 Empty Driving: 12301 Total Distance: 34439	Waiting time: 22.41 Number of Vehicles: 224 Empty Driving: 11538 Total distance: 34922
CAV	Waiting time: 20.31 Number of Vehicles: 277 Empty Driving: 7704 Total Distance: 29841	Waiting time: 22.41 Number of Vehicles: 246 Empty Driving: 10769 Total distance: 34154
BAV	Waiting time: 20.31 Number of Vehicles: 244 Empty Driving: 11910 Total Distance: 34048	Waiting time: 22.41 Number of Vehicles: 234 Empty Driving: 11032 Total distance: 34417
BCAV	Waiting time: 20.31 Number of Vehicles: 290 Empty Driving: 7603 Total Distance: 29740	Waiting time: 22.41 Number of Vehicles: 262 Empty Driving: 10925 Total Distance: 34309

Figure 1: Results for both single and bundles orders

As is clear in the table above, when bundling orders, you see that the waiting time increases, which in this case makes sense since one car can deliver multiple orders, thus increasing the waiting time. While it is also clear that the bundling of orders decreases the number of vehicles for all algorithms. Thus reducing the fixed costs, while it is also clear that the total distance is increased, thus variable costs would be increased. It is also noticeable that the total distance for the bundles seems to be increasing from the single order algorithms. This behavior is caused by the way the bundles are created. This behavior will be explained further in the validation part of the report.

## 2.3 GUI results

The GUI results from running the algorithm show all metrics for the chosen day as well as all metrics for the chosen vehicle on that day. For the metrics it is worthwhile to mention two things.

The waiting time, when driving single orders, consists of the production time of the order plus the travel time from the pick up location to the drop of location (calculated with Dijkstra's shortest path algorithm). The waiting time starts when an order is placed and ends when it has been delivered. The average waiting time is then the total waiting time for all orders divided by the number of orders. For bundles the waiting time consists of the production time of all orders plus the delivery of all orders. Again, the waiting time starts when the first order has been placed and ends when the last order has been delivered. The average waiting time for bundles is then the total waiting time for all bundles divided by the number of bundles. For interpretation of the average waiting time it should thus be mentioned that in both cases the average waiting time is based on the upper limit of the waiting time.

The results per day also show the fixed costs. It is good to realize that these fixed costs are interesting in terms of how expensive the vehicles are that are being used that day, but in real life, these will not have to be purchased every day. We included it to give a picture of the types of cars used, but in practice, these costs are spread over multiple years.

## Validation

The main changes in the project include the different routes the vehicles drive. Some approaches include the vehicles driving back in forth to pick up and deliver each order, while other approaches include the vehicles driving routes where they pick up multiple orders and then deliver them. To make sure our algorithm works as expected the routes are plotted. For the single order algorithms, non-disappearing vehicles, the routes for each vehicle are plotted, showing the whole delivery day for a certain vehicle. For the bundled orders, the route for each bundle is plotted. Furthermore, in both cases, we checked whether each order is assigned only once, and not delivered multiple times, which was correct. Thus we validated that each order is only considered once.

For the single-order version, the routes between pick-up and drop-off locations are always based on the shortest path between the two. For the MVP the routes are only the shortest path as the vehicles appear at the pick-up location and disappear after drop-off. For the non-disappearing vehicle versions, the routes include the shortest path between the two as well as including the shortest path between the last drop-off location and the new pick-up location. For this approach, the validation plots are used to compare the driving distance between the drop-off of an order to the pick up of the next order.

Furthermore, we want to validate the improvements we make by checking if the algorithm actually does what we expect it to do. For single-order deliveries, the drive between pick-up and drop-off will always be the same, no matter which approach we use, however, it may be expected that the drive between an order drop-off and the new order pick-up location will change. It is expected that by choosing the closest vehicle, as we do in CAV, the distances will be shorter than

by choosing the first available vehicle, which we do in BAV. It can also be expected that when capacity is taken into account that the distances will get a little longer again. Therefore this means the CAV should have a smaller distance travelled with an empty car in comparison to BAV. As you can see in table 1, the distance travelled between orders is significantly lower for CAV than for BAV, suggesting the algorithm works better in finding the closest vehicle. The shortest distance, however, is found with BCAV. This is the case because there are more vehicles, thus significantly increasing the possibility of having cars closer than other instances. This is also the case in the bundled order situation, since the difference in the algorithms are in choosing the closest available vehicle. Since for the bundles the distance is also lower, we see that the closest available vehicle is effective in minimizing the distance in both single and bundled orders.

Algorithm Type	FAV	CAV	BAV	BCAV
Total distance travelled between orders (km) (single)	12301	7704	11910	7603
Total distance travelled between orders (km) (Bundle)	11538	10769	11032	10925

Table 1: Distance travelled between orders for all algorithms

As mentioned in the results, the increase in travelled distance between single order and bundles is an expected output. The increase occurs due to increased distance for bundles which consists of three or more orders. The problem for larger bundles is the way they are combined. Consider one bundle, named 'x', with three orders, named 'a', 'b', 'c' respectively and 'a' being the first order in the bundle. The current bundling algorithm considers each possible order case by case to be added to a bundle. For bundle x we check if order b is within a certain reach of order a and after that we check if order c is also in a certain reach of order a. The problem is that we currently do not perform any checks on the combination of order b and order c. This means that it could occur that order b and order c are completely different direction from order a, which results in a increase inefficient bundle and pick-up route. If this behavior is really undesired by the user then the user could either decrease the waiting time or the range of the restaurant region. This problem also only occurs for days with many orders, where big bundles are possible (like we see in the results for the largest day).

Furthermore, an important step to validate the use of BAV and BCAV is to test whether these algorithms choose smaller vehicles than vehicle 5. When the capacity of a vehicle increases, the fixed costs, hourly costs and km costs all increase, thus making a vehicle more expensive. To minimize costs, it is therefore effective to try to maximize the number of small vehicles. This is why we wrote the algorithm choosing the best available vehicle. As is clear from the figures below, both BAV and BCAV choose, when possible, smaller vehicles, thus decreasing the fixed costs and variable costs. Therefore we see that for single order this is effective.

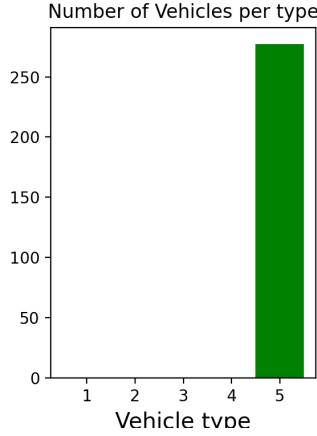


Figure 2: Vehicle types for CAV

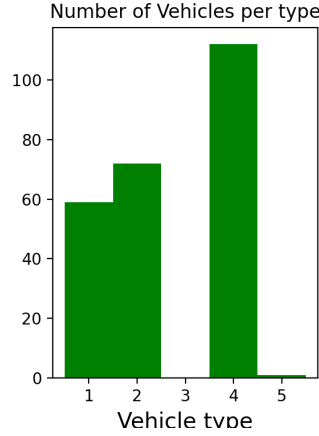


Figure 3: Vehicle types for BAV

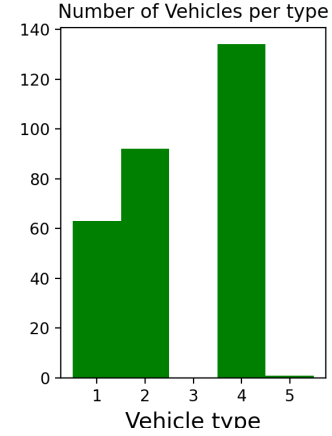


Figure 4: Vehicle types for BCAV

We also have to test this for bundled orders, which as seen in the graphs below is also correct. BAV and BCAV more often choose smaller vehicles, thus minimizing costs.

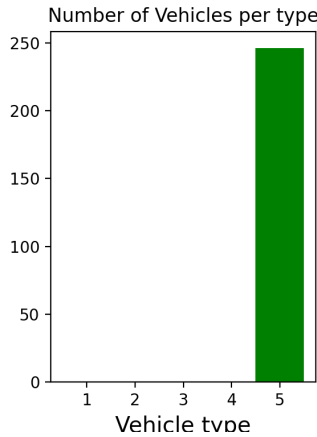


Figure 5: Vehicle types for CAV Bundles

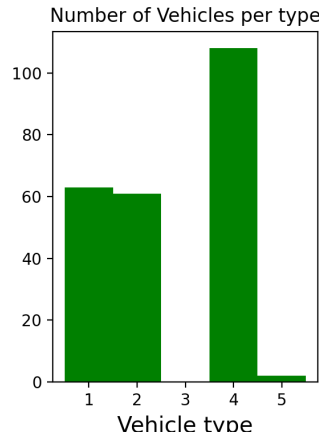


Figure 6: Vehicle types for BAV Bundles

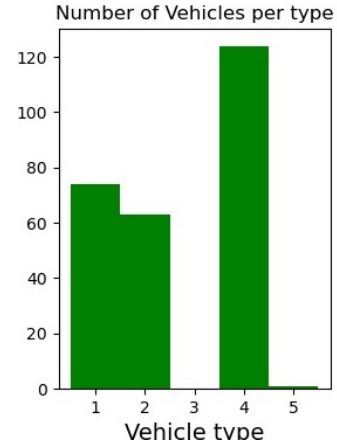


Figure 7: Vehicle types for BCAV Bundles

Lastly, we want to validate the use of bundles and we do this by evaluating the number of vehicles used for each day for single and multi orders. If we use bundles, thus bring multiple orders at once we expect that less orders will be needed. If you look at table 2, you see this is the case. For all algorithms, there is a decrease in the number of vehicles used. There is a difference in decrease which in our eyes is logical. The largest difference is found in FTF, which is due to the fact that there are less orders that are brought at once. Thus the bundling algorithm leads to 177 less vehicles, thus at least 177 bundles. The number of bundles can be increased when the waiting time interval for a vehicle is increased. In all other algorithms you can see the number of vehicles is decreased, thus decreasing the fixed costs.

Algorithm Type	FTF	FAV	BAV	CAV	BCAV
Number of vehicles (single)	3146	227	277	244	290
Number of vehicles (bundle)	2969	224	246	234	262

Table 2:  
Number of vehicles per type

We also wanted to validate the differences in results if the user inputs different values. We validated that if the user only fills in 0's for the number of vehicles per type, it gives the same results as with CAV. Furthermore, if other numbers are filled in, the algorithm fills it in as constraints and will not go over the constraint per vehicle type. Lastly we wanted to validate that when the user increases the pickup time frame the number of orders that are brought in a bundle increase. We validated that this is the case by looking at the waiting time, which increased in we increase this time frame. This means the customers have to wait longer, thus cars are waiting for extra orders more frequently. Furthermore, we also validated this by the fact that the distance cars drive empty is reduced. This is the case since more orders are bundled, meaning driving from a finalized bundle to a new bundle occurs less, thus minimizing the empty driving time.

## GUI

Now for the GUI, we chose to work with PySimpleGUI. With no experience, the PySimpleGUI seemed to be easy to learn and implement.

## MVP

The MVP relies on a lot of assumptions. All these assumptions prevent the user from making a lot of choices. This results in the MVP only relying on the order file for the current day.

For the MVP, the GUI looks as follows:

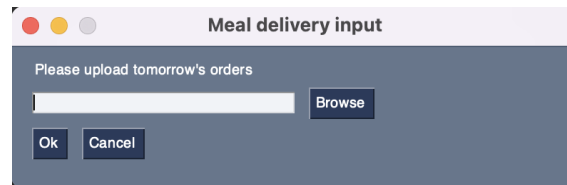


Figure 8: First window GUI in the MVP

When the user has uploaded the order file for the day, all choices are made with assumptions and by pressing the 'Ok' button the user will be given the results inside the terminal.

## Improved version

The goal of the project is to build a decision support system for the user. This means that the user should be able to make different decisions and see how the result changes. By improving the

MVP, the GUI can also be improved. The first decision for the user is to determine whether it wants orders to be bundled or not. By bundling the orders the number of vehicles can possibly be minimized, but without bundling the orders they can be delivered at the lowest possible waiting time. This decision is left up to the user. The user presses *Yes* for bundled orders but *No* for single orders. This is done in the same step as the file is uploaded, as can be seen in the picture below.

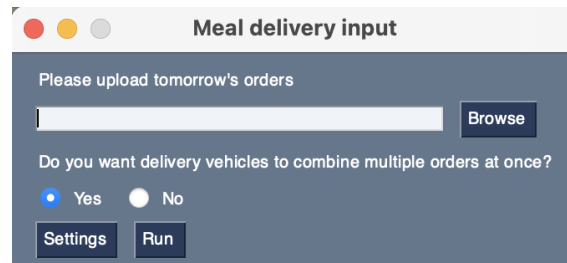


Figure 9: First window GUI in the final version

### With bundles

By deciding to bundle the orders more options based on that decision appear for the user. These options can be seen in the following picture. Type 1, type 2, type 3 and type 4 refer to the four smallest vehicles out of the five. By filling in numbers the user puts constraint on how many vehicle of that type will be used. This can be helpful when the manager already has a certain amount of some vehicles that he would prefer to use. The maximum distance between restaurants within a region is the radius parameter mentioned before. By making that value higher, might result in more restaurants being bundled together, leading to more orders per bundle.

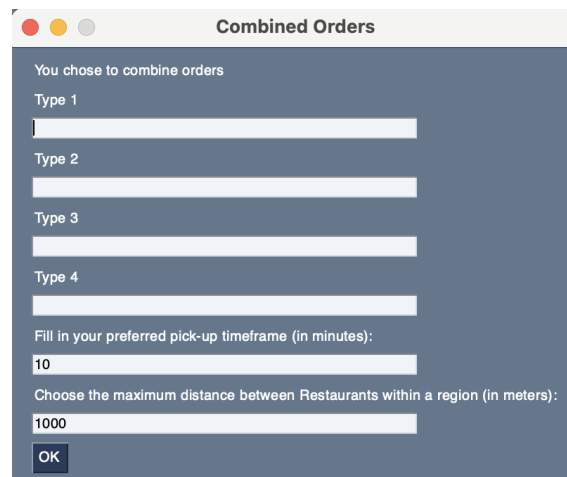


Figure 10: Settings window for user input with bundles



## Without bundles

By deciding to make the vehicle only deliver one order at a time more options based on that decision appear for the user. Again, the types refer to the vehicle types. The numbers are in ascending order, where type 1 is the smallest vehicle and type 4 is the largest.

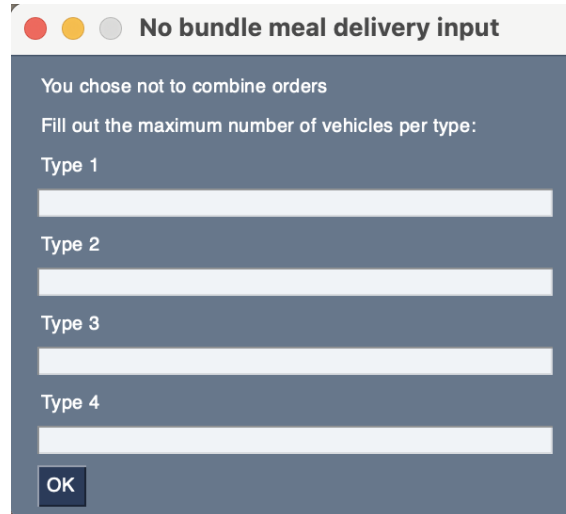


Figure 11: Settings window for user input without bundles

## GUI user manual

For the GUI to work properly these steps should be followed. Please see the `README.md` file to ensure all proper packages are installed and then run the DSS from the `mainScript.py`. From here the steps are as follows:

- Step 1: Press the *Browse* button to upload a file of orders for a whole day. (See Figure 9)
- Step 2: Choose whether the vehicle combines multiple orders at once or not. Either *Yes* to combine orders or *No* to keep single orders.
- Step 3: Press the *Settings* button. A new window appears. (Even if you want to change no settings, you always have to open the window) (See Figures 10 and 11)
- Step 4: Fill in (if any) the preferred constraints. These must be positive integers.
- Step 5: Press the *Ok* button to close the current settings window. (You cannot re-open the settings)
- Step 6: Press the *Run* button on the original window to run the code with the chosen constraints. The algorithm is run and might take up to a few minutes.
- Step 7: A second window opens with three columns with statistics about the entire day. At the right-most side of the *Result* window you find a drop-down box for the vehicles used that day. Choose a vehicle and press the *Show details* button. (See Figure 12)



Figure 12: Second window with daily metrics, plots, and vehicle selector

Step 8: A third window opens and various metrics for that vehicle for the whole day including a plot with all routes the vehicle drives during the day are shown. (See Figures 13 and 14 at the bottom of the paper)

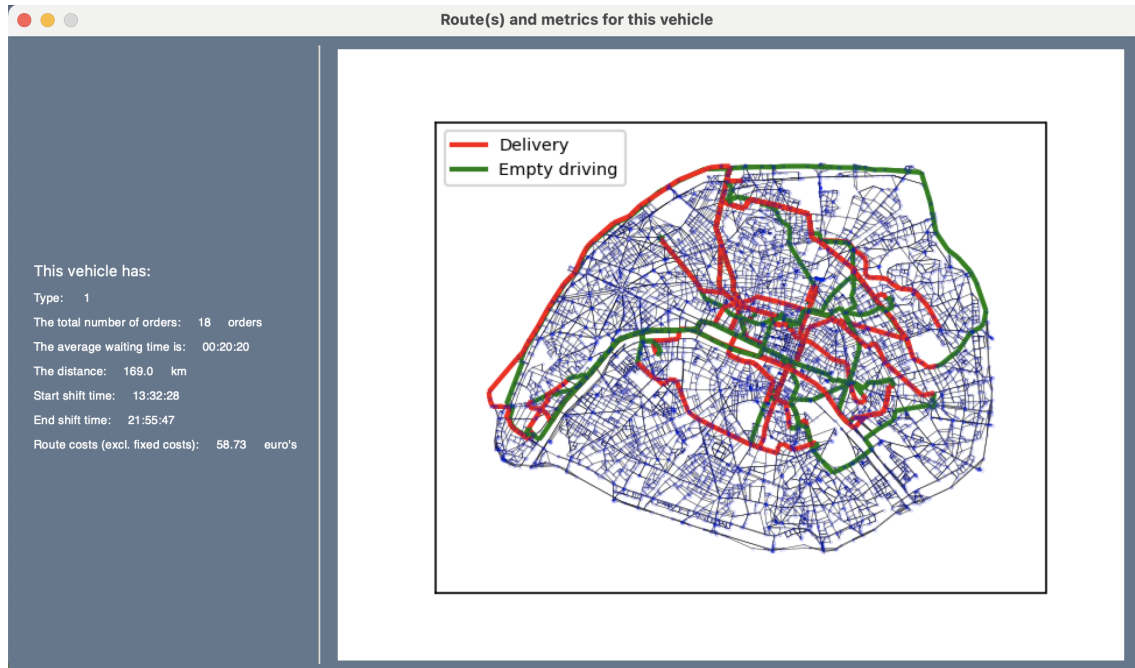


Figure 13: Third window with vehicle-specific metrics and plotted routes - Single order

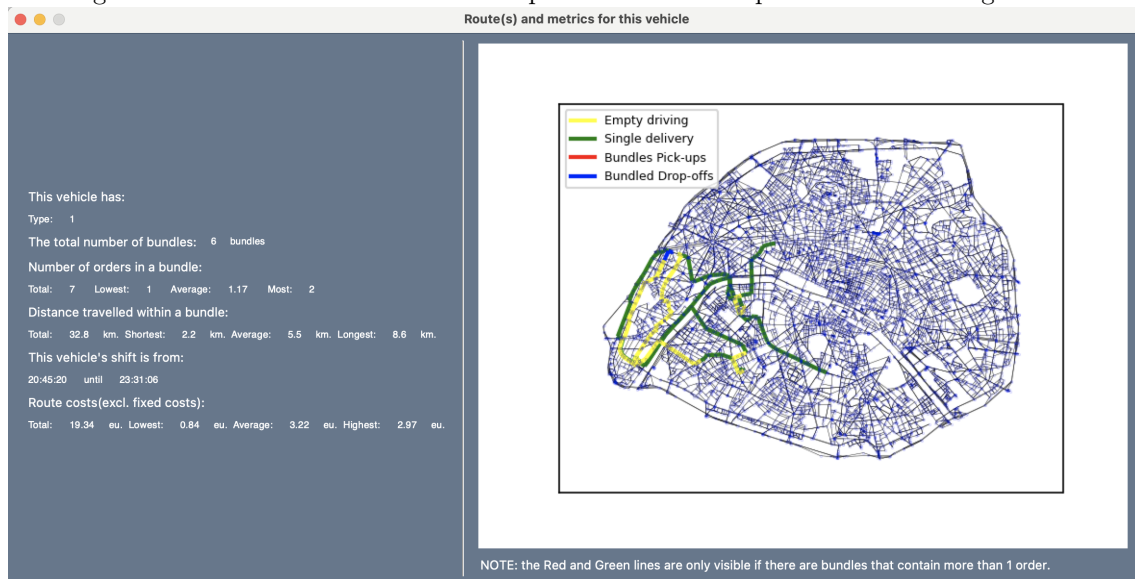


Figure 14: Third window with vehicle-specific metrics and plotted routes - Multi order

## Further improvements

- In this algorithm, we define a vehicle to be available if it can arrive at the pick-up location of the order at the time the order is ready. If the vehicle were to be able to be there 5 minutes later, for example, this vehicle is classified as not available. For further improvements, it might be interesting to find a balance between the minimum waiting time that we incorporated in this way into our algorithm and the number of vehicles.

- b) To find the closest vehicle, we use an estimation of the distances using the point-to-point distance, instead of the correct route itself. For example, if a vehicle is on the other side of the Seine than an order is, they might be close to each other point-to-point wise, but the route crossing the river might be a lot longer. A future improvement would be to find a way to tune these estimates for such outliers.
- c) When routing our bundles, we assume that vehicles will first pick up all orders and then drop off these orders. There could be a big improvement if we let vehicles mix these pick-ups and drop-offs to decrease the distance.
- d) In choosing the best vehicle type in our Best-Available Vehicle Algorithm, we only take into consideration the capacity of the car and the order size. A next improvement could be to take the distance and kilometer costs into account as well and find a way to balance these fixed and variable costs when choosing the best vehicle type.
- e) We assume that there is an infinite capacity to prepare orders at restaurants. Therefore, orders do not have to queue in the restaurant to be made. This is not a very realistic assumptions, and a further improvement would be to incorporate queues for orders in restaurants.
- f) Improve the bundling algorithm such that it makes more efficient bundles for days with many orders. This could be done by giving bundles a general direction which new order should be in, this prevents unnecessary driving back and forth from the first order.