

**Master Programme**

**Heuristic Optimization Methods**

REPORT - Project Capacitated Vehicle Routing Problem with Time Windows

**Antoine Vieville**

**Jodok Felder**

**Ac. year 2023/2024**

**Contents**

1 Best-found results 3

2 Description of the problem : CVRPWTW 4

**2.1** **Problem description** 4

**2.2** **Objective function** 4

**2.3** **Solution representation** 4

3 Greedy algorithm 5

**3.1** **Pseudocode** 5

**3.2** **Description** 5

**3.3** **Analysis** 6

4 Ant colony optimization 7

**4.1** **Pseudocode** 7

**4.2** **Description** 8

**4.3** **Analysis** 8

5 Discussion 9

**5.1** **Efficiency / Performance** 9

**5.2** **Local Search** 10

**5.3** **Problems** 10

**5.4** **Thoughts and general work process** 11

6 Plots 11

# Best-found results

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| instance | **1** | | | **2** | | |
| time constraint | 1m | 5m | un(15m) | 1m | 5m | un(20m) |
| number of vehicles | 21 | 21 | 21 | 49 | 45 | 41 |
| distance travelled | 5573.74 | 4862.63 | 4913.73 | 14232.89 | 12976.10 | 12332.36 |
| number of iterations | 21 | 103 | 306 | 4 | 18 | 71 |
| solution after iteration | 17 | 41 | 121 | 4 | 15 | 57 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| instance | **3** | | | **4** | | |
| time constraint | 1m | 5m | un(25m) | 1m | 5m | un(30m) |
| number of vehicles | 27 | 20 | 19 | 125 | 104 | 97 |
| distance travelled | 17762.30 | 11990.80 | 10845.19 | 57786.06 | 47158.16 | 41988.89 |
| number of iterations | 2 | 6 | 26 | 2 | 6 | 31 |
| solution after iteration | 2 | 6 | 25 | 2 | 5 | 29 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| instance | **5** | | | **6** | | |
| time constraint | 1m | 5m | un(50m) | 1m | 5m | un(120m) |
| number of vehicles | 44 | 38 | 34 | 105 | 99 | 95 |
| distance travelled | 69757.37 | 68597.99 | 55385.03 | 72648.15 | 68591.08 | 64018.67 |
| number of iterations | 1 | 2 | 19 | 1 | 3 | 60 |
| solution after iteration | 1 | 2 | 18 | 1 | 3 | 54 |

Programming language : Python

# Description of the problem : CVRPWTW

## **Problem description**

CVRPWTW stands for Capacited Vehicle Routing Problem with Time Windows. This type of problem consists in the following: for an instance, we are a given a maximum number of vehicles, the capacity of a vehicle (equal for all vehicles in our case) and a list of customers. Each customer has an assigned number/id, and a position on the map (x,y coordinates). Customers also have time constraints which adds a Time Window constraint to the Capacited Vehicle Routing Problem. Indeed, a customer cannot be visited/delivered before it is open (ready time), its delivery takes a certain amount of time (service time) and the customer cannot be visited after a certain time (due date).

A feasible solution to this problem is a list of routes, where each route represents the route of a vehicle, going through every customer only once. Each vehicle has to start from the depot and finish at the depot. The number of routes cannot exceed the maximum number of vehicles available.

## **Objective function**

The optimization of this problem consists in a dual objective function :

* Minimize the number of vehicles needed (1)
* Minimize the distance travelled (2)

It is noticeable that these two objectives don’t necessarily work together: we could find a solution with a lower distance by sending more vehicles to more populated regions which will result in a lower distance but a higher number of vehicles.

In our project, we consider that (1) has more importance than (2). Let us consider a solution A with a number of vehicle *N\_A* and distance travelled *D\_A*, and a solution B with *N\_B* and *D\_B* such that : *D\_A < D\_B* and *N\_B < N\_A*. Then, we will say that solution B is of better quality than solution A.

## **Solution representation**

A solution needs to contain the following information:

* Number of vehicles needed and the distance travelled
* For each vehicle, the route followed by: id of the visited customer and time of start of service at this customer.

To build such solutions, we implemented two classes: one from the data, and one for which we will initialize instances in the algorithm.

The Customer class represents customers, with all relevant attributes (id, position, time constraints).

The Vehicle class represents vehicles. Its attribute are id/number of the vehicle, its cargo (available payload), position, time of start of service for the current customer, and its route. In the route attribute, we store the id of the visited customer and the time of the vehicle. It is this route attribute that will allow us to build solutions. This class also has the methods add\_customer, which allows us to add a selected customer to the vehicle’s route, as well as wait\_time / inverse\_wait\_time, which give information on how long the vehicle will potentially have to wait for the customer to open.

# Greedy algorithm

## **Pseudocode**

**Start** greedy algorithm

**Input** : instance

solution 🡨 empty list, (customers list, vehicle capacity, max number of vehicle) from instance

compute distance matrix (distance btw each customers)

Initialize a truck/vehicle loaded with *vehicle capacity*

**Repeat** :

Order the remaining customers by distance to truck ascending

Select the first customer meeting the constraints, remove it from the list of remaining customers add it to truck’s route

If no valid customer is found, send truck to depot and initiate a new truck

**Until** all customers have been visited

**Return** solution

**End** greedy algorithm

## **Description**

Greedy heuristics :

The first logical approach we thought of to solve this problem was the following: if we want to minimize the number of vehicles needed and the distance travelled, going from a customer to its closest neighbor until all customers are served is a good option. Therefore, we implemented a greedy algorithm that follows this logic, by sorting the remaining list of customers at each iteration by distance to the truck. This allowed us to produce our first feasible solutions, with a “quite good” quality.

After analyzing the routes in output, we noticed a first flaw in this logic: the wait time. This approach only bases its selection of customer by distance. However, if the current closest customer is closed and only opens late, for example with a ready time close to the depot due date, then the truck is likely to visit few, if not only one customer on this route. Thus, there surely exists a way to take this waiting aspect into account so we iteratively serve customers that are close to the truck’s current location, and open or about to open (with a reasonable waiting time). We found that everything lies in the evaluation of this “reasonable waiting time”. This value depends on the instance. We found that for instance 1, for a waiting threshold of 150, we could reach 23 trucks, which is better than the 26 the standard greedy approach gave us. However, this value is going to be different for each instance, if we don’t want the algorithm to fail. Considering we are only going to use this greedy approach to produce feasible solutions as initial solutions for our later algorithm, we didn’t spend more time researching these thresholds.

Overall, working on this first greedy algorithm was very productive. It allowed us to implement our first ideas on how to tackle the problem, and realize which aspects would be harder to implement or more computationally intensive. We also got a better understanding of key features for the problem: selection of customers by distance and waiting time.

## **Analysis**

Performance :

Ultimately, the point of this first approach was to produce feasible solutions that we could later use as initial solutions. We saw that, with too hard of a waiting time constraint, the algorithm would fail to converge for some instances (max number of trucks reached).

However, with soft enough waiting constraints, we were able to build 6 feasible solutions. We could then compare these solutions to each instance’s theoretical minimum.

The theoretical minimum is the minimum number of trucks you would obtain if the problem was not subject to time windows constraints. It can be computed by calculating the total customer demand (sum of each customer’s demand) divided by the vehicle capacity. Therefore, the number of vehicles obtained with time windows can only be larger or equal to this minimum. If we get a number of vehicles close to this minimum, we can say our solutions are satisfactory and conclude that our approach was relevant.

For our greedy algorithm:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Instance | **1** | **2** | **3** | **4** | **5** | **6** |
| min nb of vehicles | 18 | 36 | 17 | 72 | 19 | 90 |
| nb of vehicles obtained | 26 | 60 | 49 | 125 | 64 | 111 |

We concluded that our solutions, even if far from optimal for some instances, were satisfactory. We can also notice that the algorithm performs better for some instances than others, meaning the waiting aspect impacts instances differently. This will be interesting to keep in mind.

# Ant colony optimization

## **Pseudocode**

**Ant solution construction**

**Start** ant solution construction (*ant search*)

**Input** : customer lists, vehicle capacity, max number of vehicle alpha, beta, distance & inverse distance matrix, pheromone matrix

Solution 🡨 empty list

Initialize a truck loaded with *vehicle capacity*

**Repeat** :

find all customers meeting the constraints, and compute their probability to be selected :

*p = (pheromone value \*\* alpha \* inverse distance \*\* beta \* inverse waiting time)\**

randomly select a customer according to their probability, remove it from the list of remaining customers and add it to truck’s route

if no valid customer is found, send truck to depot and initiate a new truck

**Until** all customers have been visited

**Return** solution

**End** ant solution construction

**Ant colony optimization**

**Start** ant colony optimization

**Input** : instance, greedy solution, alpha, beta, rho, nb of ants, time (or nb of iterations)

best solution 🡨 greedy solution, (customers list, vehicle capacity, max number of vehicle) from instance

compute distance matrix and inverse distance matrix

initialize the pheromone matrix according to greedy solution

**Repeat** :

Constructed solutions 🡨 empty list

**Repeat** :

constructed solution 🡨 ant solution construction (params)

Store constructed solution in constructed solutions list

**Until** nb of ants is reached

Update the pheromones (pheromone matrix[i,j] = tau\_ij) :

* evaporate the pheromones : tau\_ij 🡨 tau\_ij \* (1-rho) for all i,j
* reinforce the pheromones for the k best solutions :

quality based approach :

for all (i,j) belonging to k best solutions : tau\_ij 🡨 tau\_ij + delta, delta = 1/fitness(solution\_k)

Sort constructed solutions by best fitness (primarly : nb of trucks , secondly : distance travelled)

Compare best constructed solution with current best solution :

if fitness(best constructed solution) < fitness(best solution) : best solution 🡨 best constructed solution

**Until** time / nb of iterations has been reached

**Return** best solution

**End** ant colony optimization

## **Description**

Motivation for ACO :

Before diving into the project, we studied the literature around CVRPWTW. We found that several known algorithms can be implemented with good solutions. Namely, we read about ACO, ACS (Ant colony system, a variant of ACO), Clark Wright algorithm… Considering that we just studied ACO in class and that it is a well suited algorithm for “spatial” problems such as vehicle routing, we decided to focus on this one.

Heuristic customer selection :

To construct a solution (ant\_search algorithm), a truck lists all valid customers of the instance, assigns a probability to visit it based on its pheromone value, its heuristic distance and its waiting time. It then chooses randomly a customer based on these probabilities, which is added to the trucks route. We added the waiting time factor to optimize trucks so they don’t start their route with a late opening customer and then have to go to the depot having served only a few customers.

## **Analysis**

Performance : iteration size / instance size

As we can very clearly see in the table on page three, the number of iterations our algorithm is able to perform very dramatically decreases with increasing instance size. Starting already at instance three, we only get a maximum of one iteration per minute, which results in the algorithm not being able to properly take advantage of some of the upsides of Ant Colony Optimization. Nevertheless, since some of the solutions we found are pretty close the theoretical best, which didn’t take any constraints into account, we still went with this solution.

Parameters influence :

**Alpha** (pheromone influence) : In the smaller instances we can observe a very clear correlation between alpha size and our fitness functions, which are number of trucks found and distance, for the solutions, it is even almost linear in for the three instances. In general we can very confidently say that a higher alpha value results in better solutions.

**Beta** (distance influence) : We can also observe a trend of higher value leading to better solutions. The trend is clearer to see in the plots, although for some instances, the solutions are worse than comparable solutions where we tuned alpha, see for example the two graphs for Instance three.

**Rho** (evaporation rate) : It is much harder to read a trend from the plots and our solutions, although for the smaller instances one would be inclined to also suggest that bigger values mean better solutions. For instance 6 however, the complete inverse is true. More importantly though, when looking at the number of trucks, one can very clearly see that for most instances there is actually not that big of a gap between the best and worst solution when tuning rho. This suggests that the changing of this parameter is not as important as the other two before it and a value of 0.1 might be the best-suited.

**Number of Ants**: In regards to the number of ants used in the search, there really isn’t any which we can make out in the solutions. This is probably because our code is suboptimal in regards to efficiency, which we will talk more about in the next part, and therefore a greater number of ants is not really able to impact the solutions on a big scale, since our algorithm then needs more time per iteration and therefore is not able to execute as many iterations.

In conclusion, we can with a certain level of confidence state that a good mix of alpha and beta values should lead to good solutions, while the rho parameter doesn’t have too much impact on it, at least in our case, and we can’t really conclude about number of ants.

For further information on the parameters see the **plots attached at the end of this report**.

# Discussion

## **Efficiency / Performance**

One aspect we noticed when running our implementation on bigger instances like 4,5 and 6 is that it needs a longer running time to take advantage of the ACO algorithm. Indeed, for 1min or 3min runs, we only get a few iterations and therefore don’t fully use the pheromone enriched routes. Instances 1, 2 and 3 seemed to work as intended and we are rather confident in the solutions obtained for them.

A small sidenote on the “unlimited” runs for the instances: We saw that the solutions tended to converge after some time and therefore stopped those runs after a given time, which is why some instances have a higher amount of iterations than others and vice versa. For example, for instance three we let the algorithm run for 25 minutes and for instance four for 30 minutes.

In chapter 5.3 we look a bit into the code and discuss potential root causes of our performance problems.

## **Local Search**

When researching similar problems online we found that many implementations would use, just like mentioned in the lecture, Local Search at the end of an iteration to try and improve the best found solution(s). However, we opted not to implement that in our solution mainly because of performance issues. Moreover, we weren’t too sure how to even construct a neighbourhood given the constraints we had to work with, especially the time windows. Since we also didn’t see how the Local Search could improve a solution by more than one truck at a time, we agreed to focus on other areas and not take time away from possible iterations.

## **Problems**

Let’s start this chapter with some concrete problems where we suspect we lost some performance. When looking at the ACO two areas with potential limitations stood out to us.   
  
First, the pheromone update functions. In essence, we suspect that there could be a way in which one would combine the evaporation and reinforcement into one function and therefore save some time. Because the matrix gets rather big quickly with an increasing number of customers, for instance six it is already 1000x1000, which is 1.000.000 edges which have to be considered. Like we can see in the code, we even tried to find a solution to this, but ultimately failed and since we weren’t 100% sure there was even a way to save time, agreed on concentrating on other areas of this project, since we were far from finished at that point.  
  
The other area which we suspect to have impacted our implementation’s performance, and this time pretty confidently, is in the ant\_search function around the lines where we call the is\_valid function. In essence, we check every time we are at a customer for all the remaining customers whether they are valid next stops on the route and do some calculations if that’s the case. Especially in the beginning of a truck’s route this is very computationally expensive, since almost all or all, depending on where the truck currently is and the time, of the remaining customers, which is at the beginning of a solution construct also a big number of customers, is indeed valid. For example, for the first truck in a solution for instance six, starting at the depot, there are 999 possible remaining customers for which the is\_valid check has to be done and then the calculations executed. Then one customer is selected and there are still 998 customers left which will be checked with is\_valid and a high number of them are positive, meaning we have to execute the calculations.   
Because of this we are fairly certain that this is a huge time sink, but couldn’t make out how we could forego this. The general idea would most likely include calculating the probabilities only once per iteration and excluding our is\_valid check. The problem was just that we couldn’t make out how to do this, which we will illustrate in the following example: Let us assume we are in the middle of a route of a truck at say time 300. If we calculated the probabilities beforehand and now choose the next customer based on them, chances are we could get a customer whose time window already has passed. Especially at the end of a route of a truck this would apply to most of the customers. So therefore we couldn’t figure out how we could only look at the customers which are at that point in time relevant as possible next customers, without doing the is\_valid test inside of ant\_search and calculating and adding up the probabilities inside.  
But like we said, this is a huge time sink and it is probably somehow possible. Unfortunately, we just couldn’t figure out a solution to it.

However, we were told before the winter break by the teacher that students commonly go with ACO with this project, and that this way is computationally demanding. Therefore, it seems normal to face this sorts of problems.

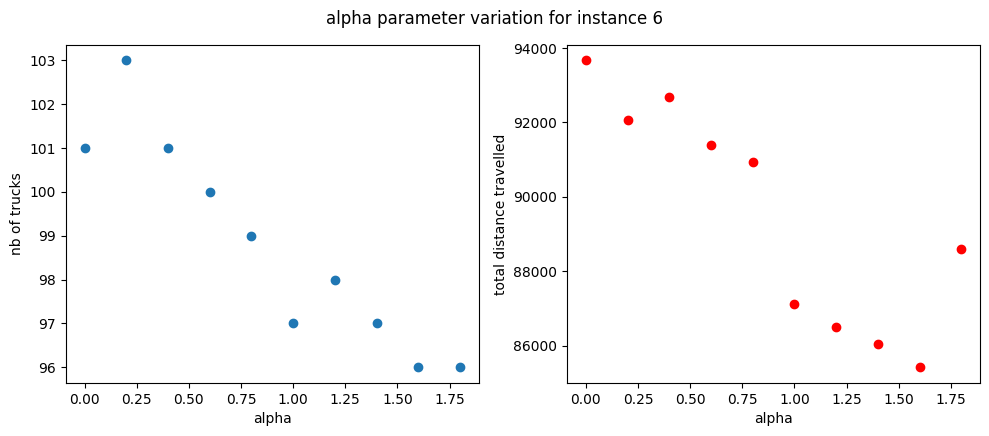
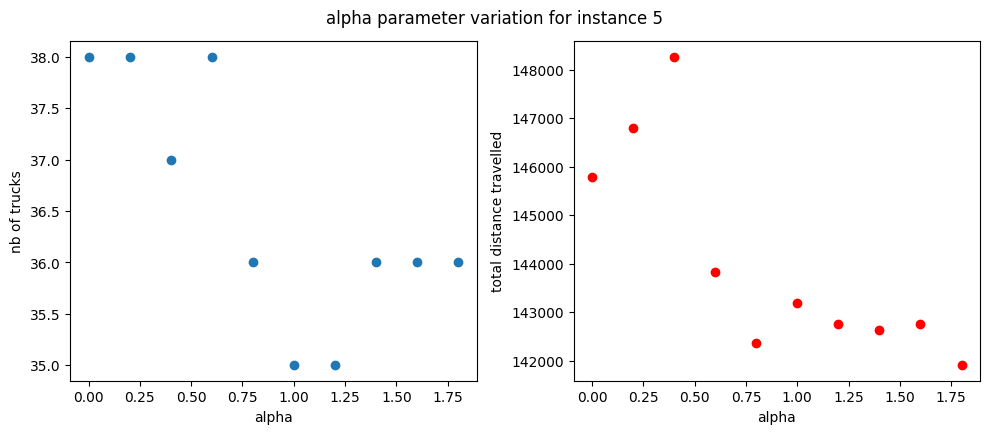
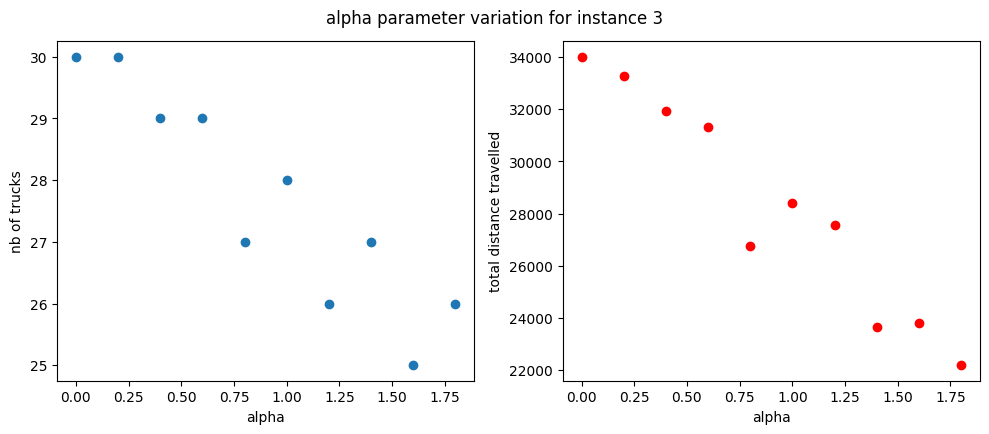
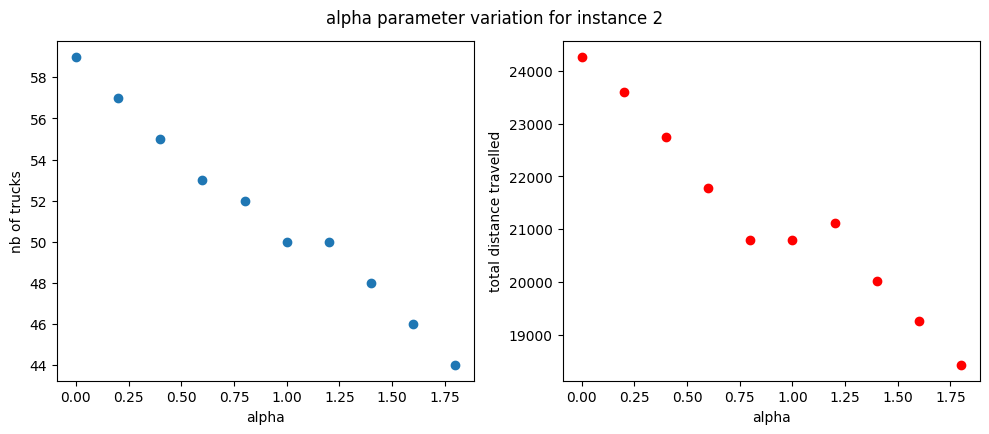
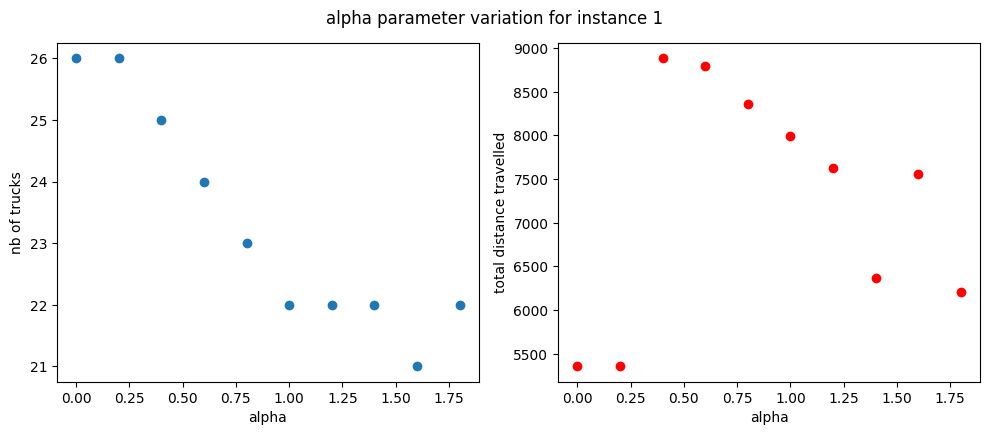
## **Thoughts and general work process**

In the beginning we worked on separate implementations of ACO for this problem since we thought it would be a good idea to get a start like that and then compare the two approaches, maybe also in regards to performance. Since one implementation was up and running pretty fast we then abandoned the other one, which is still included as prototype.py in this solution folder. Although in the end we couldn’t test both of them head to head against each other, we still feel that this was a good choice, because this way both of us got a very good understanding of the code and the solution as a whole, even if one wrote some of the parts and the other one other parts.

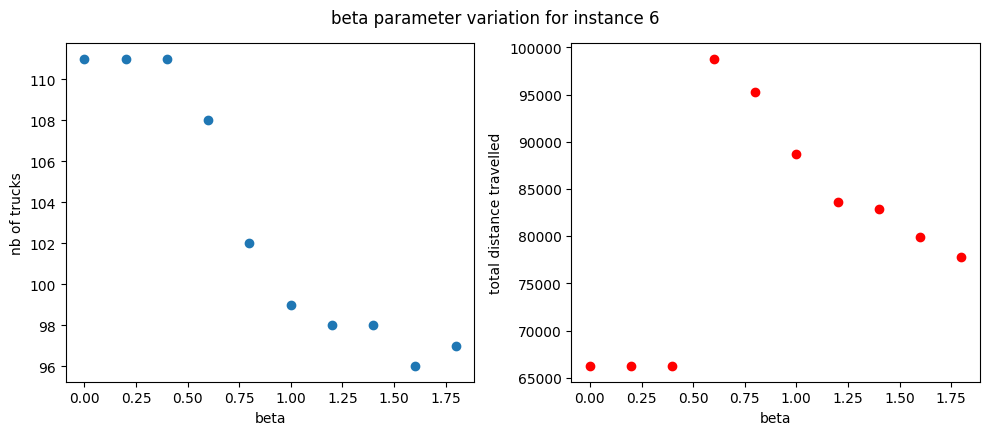
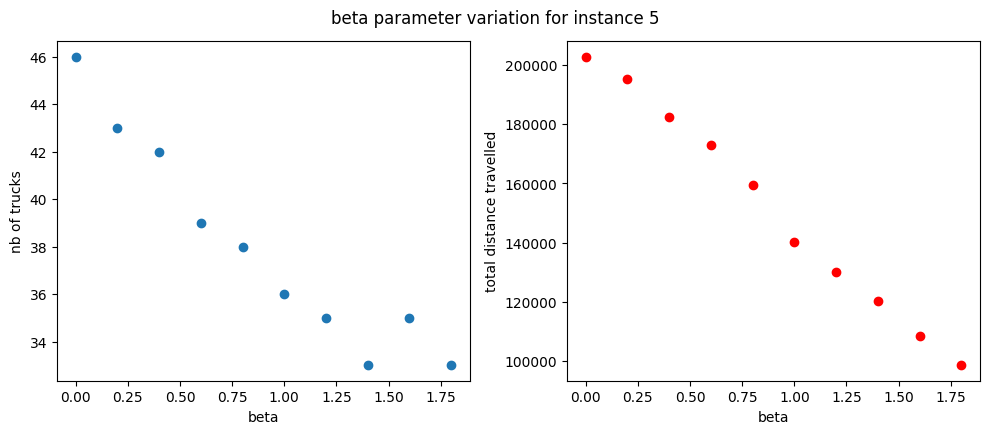
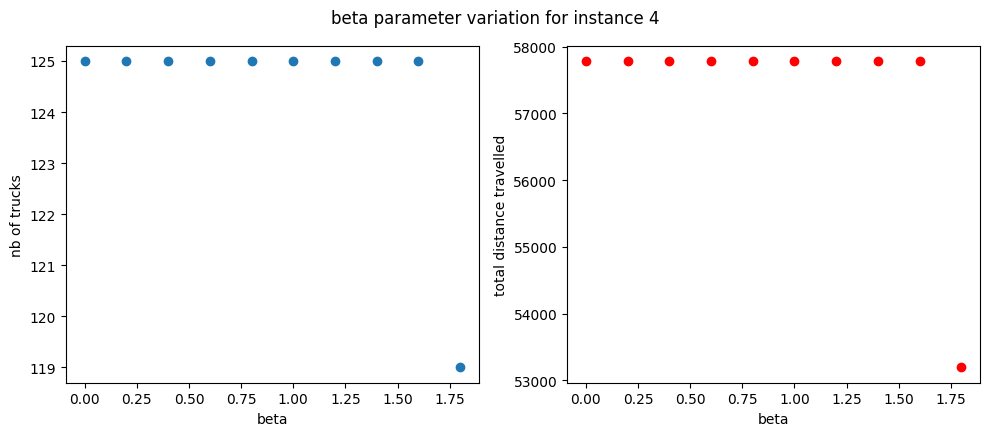
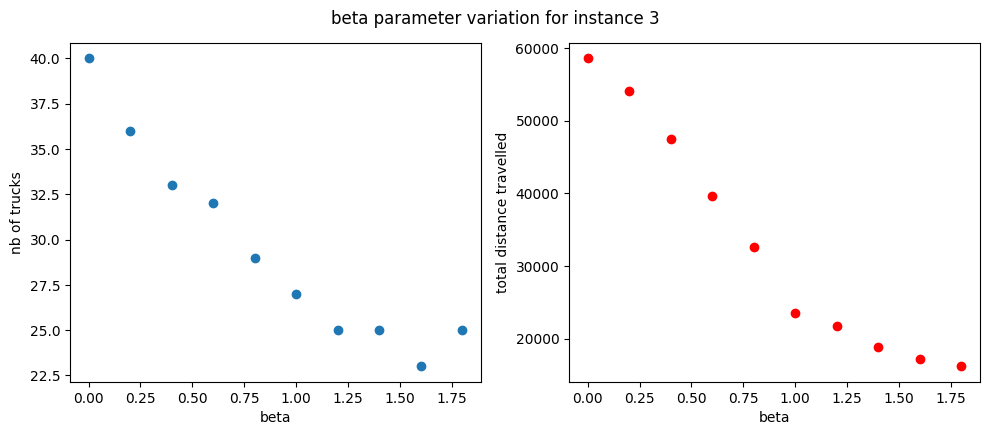
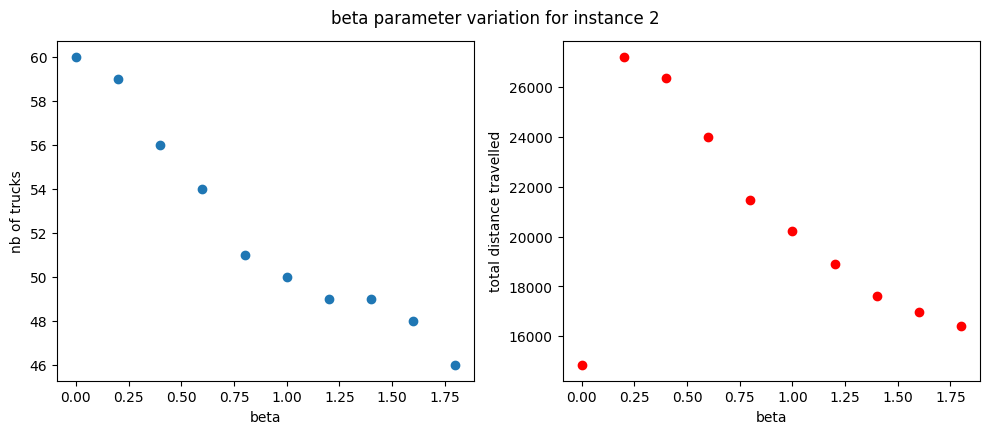
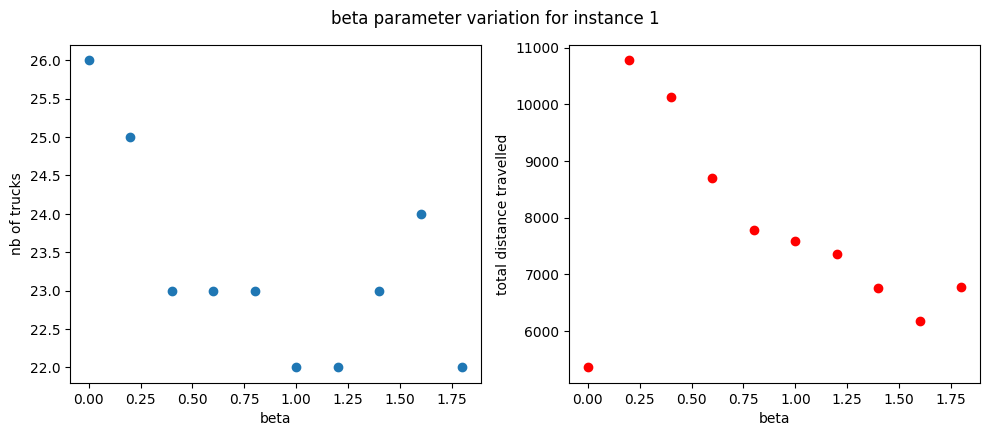
All in all we are very happy with our final result, since we’ve obtained some pretty good results despite the performance problems mentioned above. Furthermore, through this project we could get a glimpse of what might go on in the background of companies like Amazon or DHL and were able to see the fast growing complexity of such problems.

# Plots

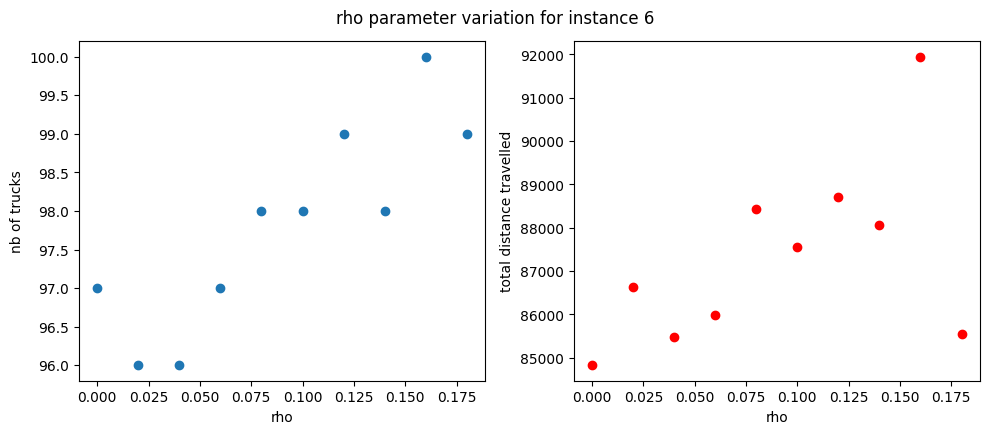
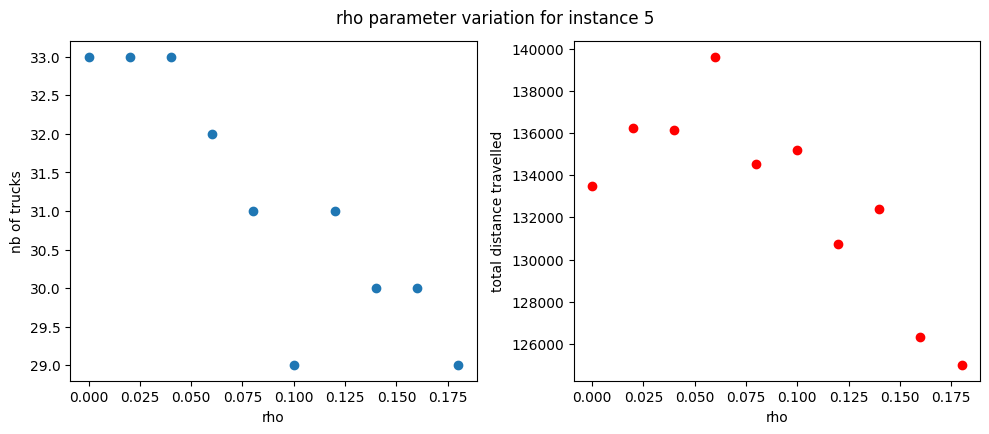
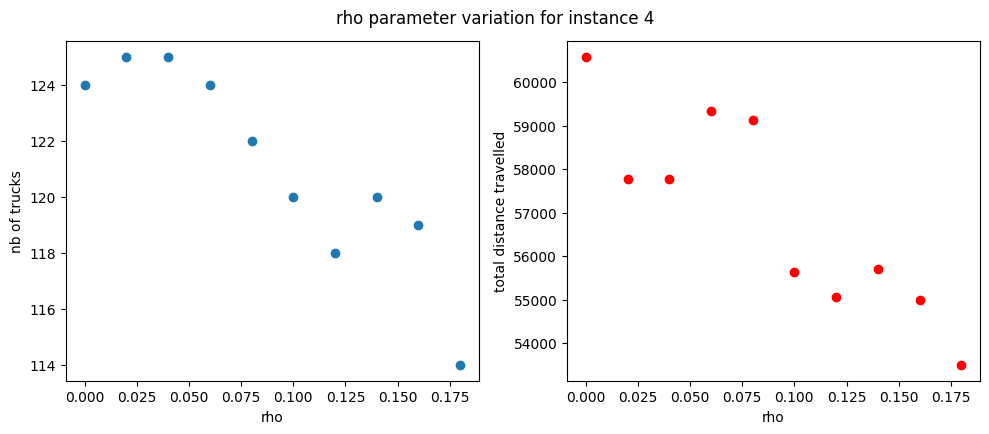
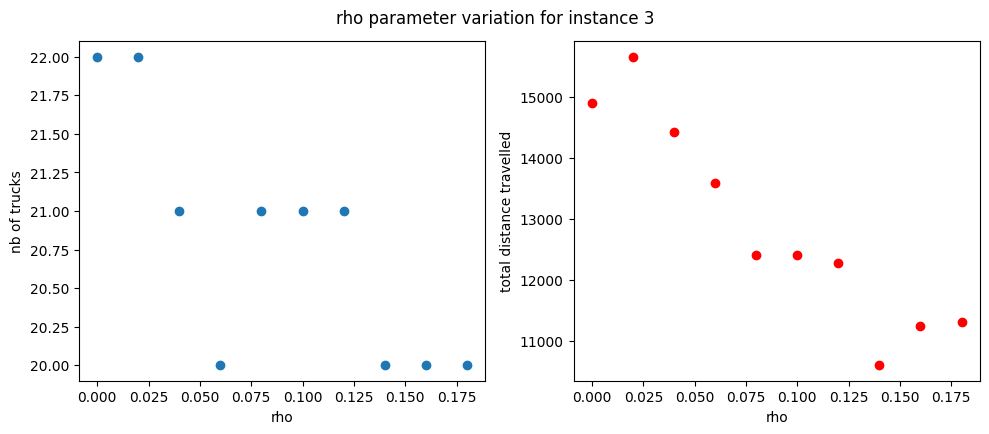
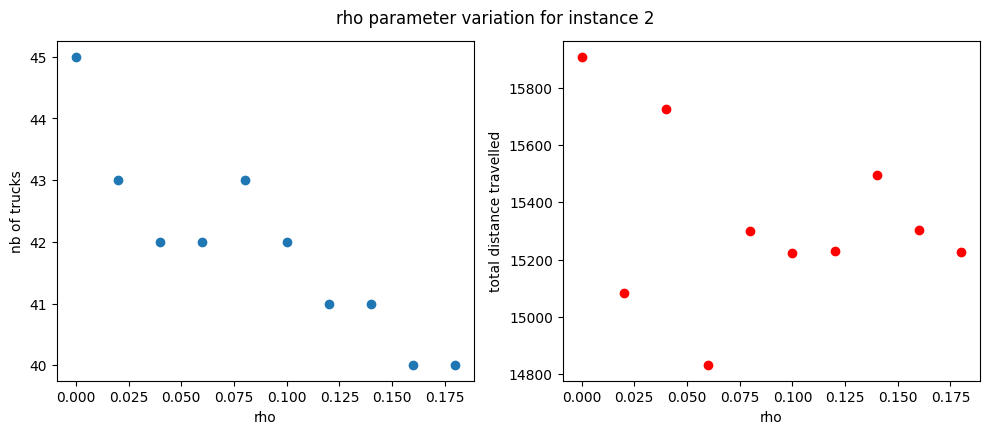
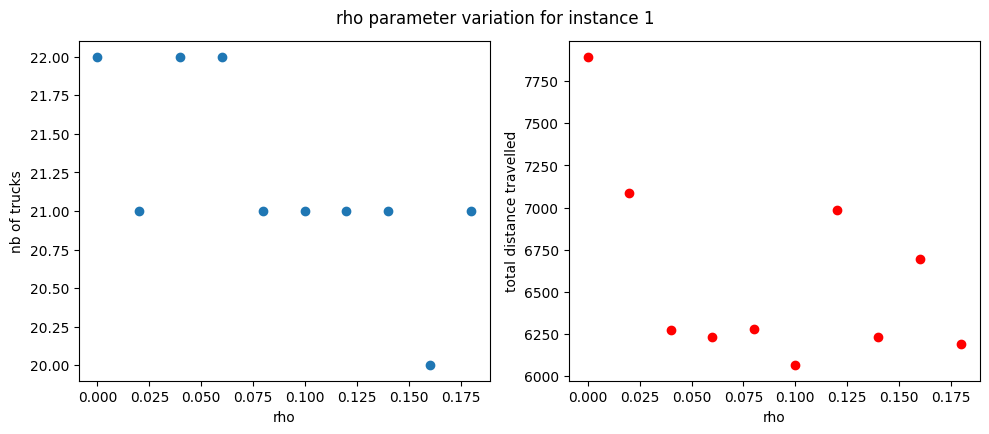
Alpha : pheromone influence :



Beta : distance influence :



Rho : evaporation rate :



Number of ants per iteration of ACO :

