Universidad del País Vasco Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL

TRABAJO FIN DE MÁSTER

Review of the assembly instructions, analysis of the amount of copper used for brazing, automation of the change of format of a set of files, data analysis, and optimization of the shuttle service provided by a company for the pick-up and delivery of its workers.

Estudiante *Lázaro, Vega, Daniel*
Director/Directora *Herrero, Villalibre, Saioa*
Departamento
Curso académico *2020-2021*

Abstract:

Spanish:

El primer bloque reúne las tareas y responsabilidades que he tenido como ingeniero junior en *Franklin Brazing & Metal Treating* durante mis prácticas. Desde la revisión y actualización o corrección de antiguas instrucciones de ensamble, hasta el análisis de datos de producción y mano de trabajo para la optimización de procesos. También fui responsable de crear una tabla de cálculo para el proceso de Brazing y la automatización del cambio de formato de las instrucciones de trabajo de la empresa. Tanto Franklin como yo hemos terminado esta experiencia muy satisfechos. Agradezco a la Universidad del País Vasco y a la Universidad de Cincinnati la preparación y oportunidades que me han dado.

English:

This paper brings together the tasks and responsibilities I have had as a junior engineer at Franklin Brazing & Metal Treating during my internship. From reviewing and updating or correcting old assembly instructions, to analyzing production and labor data for process optimization. I was also responsible for creating a calculation template for the Brazing process and automating the changeover of the company's work instruction format. Both Franklin and I have finished this experience very satisfied. I thank the University of the Basque Country and the University of Cincinnati for the technical preparation and opportunities they have given me.

Basque:

Lan honek praktiketan Franklin Brazing & Metal Treating enpresako ingeniari junior gisa izan ditudan zereginak eta erantzukizunak biltzen ditu. Muntatzeko argibide zaharrak berrikustea eta eguneratzea edo zuzentzea, ekoizpenaren datuak eta eskulanak aztertzea prozesuak optimizatzeko. Soldadura prozesurako kalkulu taula sortzeaz eta konpainiaren lan argibideen birformatizazioa automatizatzeaz ere arduratu nintzen. Franklinek eta biok oso pozik amaitu dugu esperientzia hau. Eskerrak ematen dizkiet Euskal Herriko Unibertsitateari eta Cincinnatiko Unibertsitateari eman didaten prestaketa eta aukerengatik.

Contents

1. Intro: 6
2. Objectives: 6
3. Duties and tasks accomplished: 7
3.1. Update and review the Work Instructions. 7
3.2. Calculate the correct amount of copper needed for an appropriate brazing. 7
3.3. Automation of the process of changing the sorting format of the company's files to switch from an old data base system to a new SQL one. 8
3.4. Cleaning, visualization, and analysis of the data. 9
3.5. A live route planning algorithm for the company's shuttle service. 9
3.5.1. Background. 10
3.5.2. Methods. 10
Customer allocation: 11
Ant Colony Optimization: 11
Parameters: 13
3.5.3. Results. 13
3.5.4. Conclusions of the task. 16
4. References 16
Annex 1. Python Code for the Automation of the process of changing the sorting format of the company's files to switch from an old data base system to a new SQL one. 17
Annex 2. Python Code for the live route planning algorithm for the company's shuttle service. 21

1. Intro:

This internship has been my first job experience as an engineer, and I am very grateful for the opportunity that Franklin Brazing & Metal Treating has given me depositing trust in myself. What I have learned during this internship has not been limited to technical know-how, they really taught me how a work environment is supposed to be. This company cares about each worker there, asking them for feedback in a daily basis to improve their working conditions. Thus, the aftermath of this good practice is a welcoming and friendly workspace where you feel like a part of it since the first day. I have learned a lot about how to behave professionally in a work environment and how to work as team, the latter if I am being honest, has always been one of my flaws.

This was an Engineering Internship and as I expected, I have been able to apply different ideas learned during my major (Industrial Engineering) and during my masters (Artificial Intelligence).

To bring a little bit of context, Franklin Brazing & Metal Treating is the only company with the PuroBrite™ system to braze, normalize, temper, stress relieve, and anneal stainless steel, carbon steel and other ferrous alloy parts. The process was developed for the strict requirements of stainless-steel brazing and annealing. Finished stainless steel parts must always be clean and bright, and have precise micro structure requirements for strength and corrosion resistance properties. Average brazing and annealing processes cannot reliably meet these requirements. The company has an engineering team and a manufacturing team.

The engineering team supervises the manufacturing process, the instructions for each part, and quality of the products. Some parts go through the furnace directly, others go through a more tedious process involving a pre-assembly, assembly, and furnace time. The brazing of the parts occurs on the furnace, and there are two furnace configurations depending on the specs of the part, the high flow configuration, and the low flow configuration. High flow configuration uses a different atmosphere inside the furnace achieving higher temperatures and avoiding oxidation.

The manufacturing team handles the workforce, they work on the pre-assembly, assembly and furnace stages of the manufacturing processes. Also, they work on the quality check of the products, and communicate the defects found. That information is crucial and valuable for the engineering team, to evaluate the sanity of the furnace, and investigate what changes should be made to fulfill the quality specs demanded by the client. They work under shift schedule, since the furnaces work nonstop 24h a day, 365 days a year unless something goes wrong. This is because the fancy process of turning on a furnace involves a great cost for the company.

The company is based in Lebanon, and the workers live all nearby the company. Not all of them have vehicles, so the company offers a shuttle service for its workers. The service consists of two vehicles that pick-up and deliver the workers to the desired destination.

2. Objectives:

The tasks and the objectives go hand in hand. The overall objective of the internship was to contribute as much as I could to the company and apply to the extend possible the knowledge acquired during both master's degrees.

3. Duties and tasks accomplished:

3.1. Update and review the Work Instructions.

Through this company a great number of different pieces are manufactured every day, each of those has its specific Work Instructions. Workers follow these instructions, so those must be updated, otherwise mistakes will be made during the operations.

Every time that a process change was needed due to a change of the part or a new requirement of the client, I had to update the instructions of that part. In addition, to make the instructions as clear as possible for the worker images of those steps attached, indicating all the details of the process. In this company a great amount of the workers are hispanic, so every written instruction had its translation right below. After updating an instruction, it needs to be saved as a PDF and returned to the data base system of the company, archiving the old one and specifying the reason of the change.

3.2. Calculate the correct amount of copper needed for an appropriate brazing.

I made an Excel template to calculate the volume of filling metal needed, depending on the geometry of de different parts that make up the joint. Brazing is a welding process for metal parts, where a filling metal (Copper or Aluminum alloys) is used to fill the gap of the joint between both parts. This filling metal usually comes as a preform (ring or washer), and it is assembled in the joint before the brazing process. There are a lot of preforms with different geometries (inner and outer diameter, diameter of the wire, thickness depending on if the preform is a washer or a ring). After the assembly of the components and the preform, the part goes into the furnace. Once it is in the furnace the part is heated to the brazing temperature, the filling material melts and by capillarity action process fills the gap of the joint.

How does this template work? As it has been said previously, this template takes a couple measures as input, for which the engineer needs to take a look to the print of the part. Once you have those values the excel will calculate the amount of copper needed, and gives you back the parameters of the theoretical preform (ring or washer) that would have that exact amount of copper. Usually the company uses standardized preforms (the piece of filling material), so you will compare those theoretical values with the standardized preform that will be used. This template is usually used in two cases:

- When a defect that does not meet a client's requirements and specifications is detected.

The old preform, the new one and the theoretical one should be compared to check if there is enough copper.

- When a new part arrives, to decide the new preform that is going to be used.

The scope of this template is to know how much copper you are using and make sure that there is an appropriate exceed of filling material. This analysis will be always folloId be a trial to verify that the preform works. The template is just one more tool to interpret the results of the trials, brazing is a very complex type of Ilding where too many factors must be taken in mind. The most important part to decide if a preform works or not are the trials, the empirical proof is the final decision factor.

3.3. Automation of the process of changing the sorting format of the company's files to switch from an old data base system to a new SQL one.

Franklin Brazing is preparing to change from their old data base system (Access) to a new one (SQL). For that they designed a different way to manage the work instructions, by operation or process. Thus, the work instructions are being split into individual PDFs for each operation. For example, if part A goes through a pre-assembly, knurling, assembly, load to furnace, off-load inspection processes its work instruction would have to be split into a PDF for each process. After the split there will be 5 PDFs: pre_assembly.pdf, assembly.pdf, knurling.pdf…

There are some clients that have more than 80 different parts, if the splits are done one by one, the work would take days of work, maybe even weeks. To save this time and resources to the company I decided writing a program to automate the process. I used Python, a language that I learned during the masters of Artificial Intelligence, a lot of subjects required python to develop their projects. The clients have different parts, but those parts usually have their work instructions written using the same criterion. Consequently, the program will split all the documents of one client at a time. For every client there is the need to update a few values of two variables that are very important for the program to work. One of these variables is a list of the keywords that the program will search for in the work instructions. Those key words are the name of the operations or the sections that can be found in the work instructions of the parts that belong to one client. The other variable is a list of strings too but contains the titles that will be used to sabe the split files. Every file will be saved using the same norm. The first part of the name of the file will be the part number, the second part will be the operation that is explained in that PDF (That is the title contained in that important variable).

I wrote two scripts, one of them has all the functions that the program required, and the other one has the main program. I separated the code in two scripts because probably this program will be used by people that do not know a lot about Python. Hence, I tried to leave the main program script very clean and with comments to explain what the main variables that affect the code are for. Additionally, I wrote a tutorial with detailed instructions that follow all the steps. Before running the code, a folder needs to be created in the same folder where all the work instructions that are going to be split are. That is the directory where all the new individual instructions will be saved.

The Python skills that I acquired during this master have allowed me to come up with this project and make it work.

3.4. Cleaning, visualization, and analysis of the data.

I gathered raw data from the Access database of Franklin to get valuable information aiming to bring light upon important inquiries. Studies like if there is capacity for more the production of more parts or during which shifts is more scrap produced, which workers are more efficient, which parts take the most capacity of the machinery and resources… To carry out this task I learned the basics of Tableau, a very used tool in the field of Data Science to clean and visualize data. I imported the datasets from the Access database of the company, there they have all the information about the production. Once in Tableau I cleaned the data removing the Nan values and the outliers.

In the Applied AI and ML tools I learned the pre-processing basic methods and ideas that I needed to develop this task.

3.5. A live route planning algorithm for the company's shuttle service.

The Vehicle Routing Problem is a classical problem in discrete optimization. The interest of this problem lies in the multiple variants and applications it has in the real world for the transportation of goods and people. In this work I solve the dynamic version of this problem for a specific application: the shuttle service provided by Franklin. Using a combination of smart clustering and Ant Colony Optimization, I have solved the problem and demonstrated the algorithm can outperform the performance of the route planning done by a human operator.

The service uses several vehicles for the purpose. The users can request point to point trips within a 2-mile radius around the company location. So far, each vehicle needs two workers to operate: the driver and a copilot that decides the routes based on incoming requests. There is no communication between vehicles, other than a general view of all the pending requests. These facts result in a sub-optimal service, that often leads to long waiting times when demand for the service is high. Therefore, it is of general interest to come up with a system that can optimize and coordinate the routes of each vehicle to improve the quality of the service.

I define the problem to solve as follows. I consider as input several customer requests, each with a pick-up and a drop-off point in a metric space. The set of requests is not fixed, and new requests

appear during the simulation. I also consider the number of vehicles available to provide the service, and the capacity of each of them k. The goal is to calculate a route that progressively serves all the incoming requests, minimizing its length and the waiting time of the customers.

To bound the scope of this work and fit it to the allotted time frame, I have considered several simplifications and assumptions. Each request will have a single person riding, as opposed to the real service where groups of up to 5 customers can request a ride. The vehicles can have multiple customers with different routes riding at the same time, provided the maximum capacity is not exceeded. I consider the capacity of a vehicle to be k = 5. I also limit the number of vehicles to N = 2.

All the incoming requests go to a common customer pool shared by all the vehicles in service. However, it is worth noting that each customer requires its pick-up and drop-off locations to be included into the same route. This is not only a constraint, but also an added difficulty since both waypoints have influence in the length of the route.

I have divided the resolution of the problem into two tasks. First, based on the two waypoints and waiting time of each customer, a clustering algorithm allocates a set of $n \leq k$ customers from the pool to a vehicle. Note that with this limitation the capacity of the vehicles can never be exceeded. An Ant Colony Optimization (ACO) algorithm takes the 2n waypoints of the selected customers and finds a near-optimal route to serve them efficiently.

3.5.1. Background.

The Vehicle Routing Problem (VRP) is a well- known discrete optimization problem. The VRP involves a set of customers that must be served once, and a fleet of vehicles that depart from a depot, serve the requests, and go back to the depot. The VRP and its many possible variations have been widely studied in the past. [1] shows an extensive literature review on the topic. Another interesting review is presented in [2], with a novel method to solve the multi-depot VRP. One possible variation of the VRP closer to reality is the Dynamic Vehicle Routing Problem (DVRP), where the customers change their demands gradually with time. One way to solve this problem is to divide it into multiple static VRP problems [3]. In the specific problem we are solving, the vehicles do not go back to a depot after serving several requests, and every request has two waypoints, both for pickup and delivery. These conditions are shared with the Dial a Ride Problem (DARP) when the capacity of the vehicle k is not equal to 1 [4]. However, to the authors' best knowledge, the problem that combines DVRP and DARP has not been solved yet.

3.5.2. Methods.

In this section we will introduce the algorithms used to solve the two tasks mentioned in the previous section.

In the dynamic problem, customer requests change gradually with time. Directly using an ACO algorithm to solve the problem would entail several issues. If the ACO is executed every time there is a new customer in the pool, the planned route changes every few simulation steps. This means that the vehicles never complete all the computed routes, and they are not exploiting the benefits of route planning with the ACO. This approach is also computationally expensive. That is why in our methodology we divide the dynamic problem in multiple static problems [3]. In turn, the static problem is divided into two tasks: customer allocation and

route planning with ACO. A new static problem is solved every time a vehicle finishes serving its current route.

Customer allocation:

This algorithm takes as input the information from all the customers in the pool and allocates n ≤ k to a vehicle. The information contains the location of the pickup and the drop-off in a metric space. To account for the waiting time, we add the time that customers have been in queue to this information. These parameters are fitted into a vector of the form:

$$\boldsymbol{x}_{customer_i} = \begin{bmatrix} x_0 & y_0 & x_f & y_f & t_{wait} \end{bmatrix}^T \qquad (1)$$

where x_0 and y_0 denote the coordinates of the pickup point, x_f and y_f the drop-off point, and t_{wait} *represents the waiting time. The vector of each of the vehicles is built as follows,*

$$\boldsymbol{x}_{vehicle_j} = \begin{bmatrix} x_v & y_v & x_v & y_v & t_{longest} \end{bmatrix}^T \qquad (2)$$

where x_v and y_v denote the coordinates of the vehicle when this algorithm is executed. $T_{longest}$ is the highest waiting time of all the customers in the pool. The choice for the drop-off point of 2 the vehicle is arbitrary, and helps to select customers closer to the vehicle. Setting the waiting time of the vehicle to the highest one of the customers in the pool makes the algorithm give more weight to the customers that have been in queue the longest. The algorithm calculates the Euclidean distance between $x_{vehicle\,j}$ and all the customers in the pool, $x_{customer\,i}$. Then, selects the *n* lowest distances, allocates those customers to the vehicle, and removes them from the pool.

Ant Colony Optimization:

This algorithm considers a set of $*m*$ ants, each of which represents a vehicle. All of them start at the node corresponding to the initial position of the vehicle and visit the way-points assigned to the vehicle until all of them have been visited once. Being n the number of customers assigned to each vehicle, the route consists of $*2n+1*$ waypoints: one pick-up and drop-off location per customer and the initial position of the vehicle. For a given number of iterations $n_{iteration}$ every ant generates a route over $*2n*$ steps by choosing a new waypoint at each step.

In this DVRP, it must be considered that drop-off locations cannot be visited before pick-up locations. Therefore, to overcome this constraint, we have defined an “illegal” list for each ant, which contains the drop-off waypoints whose respective pick-up locations have not been visited yet. At each step, any arc defined by the actual position of the ant and a point in its “illegal” list will be considered an “illegal” arc. Note that “illegal” lists depend on which nodes have been visited so far and can be different for each ant. This means that at the same time an arc may be illegal for one ant but not for another. Consequently, an ant could choose a no longer illegal arc for itself, and thus deposit pheromone on it, which in the next step could still be illegal for another ant.

Since next waypoints are chosen based on pheromone and distance information, to avoid selecting “illegal” paths, before choosing any next way-point, $*m*$ copies of the current pheromone information are done. Each copy is used for one ant, and in it the ph≥eromones corresponding to all the “illegal” arcs of that ant are set to zero. This way, if each ant uses its pheromone copy for the decision, “illegal” arcs will have zero probability of being chosen.

The equations to calculate the probability of an ant taking an arc I are extracted from [9]. Two probabilistic rules are defined to select the next destination. A random number $*q*$ between [0,1] is generated to decide which one to apply. If q ≥ q_0 the ant will use

$$p_{ij}^k = \begin{cases} 1 & \text{if } j = \arg\max a_{ij} \\ 0 & \text{otherwise} \end{cases} \qquad (3)$$

Otherwise it will use

$$p_{ij}^k(t) = \frac{a_{ij}(t)}{\sum_{l \in \mathcal{N}_i^k} a_{il}(t)} \qquad (4)$$

where

$$a_{ij}(t) = \frac{[\tau_{ij}(t)]\,[\eta_{ij}]^\beta}{\sum_{l \in \mathcal{N}_i} [\tau_{il}]\,[\eta_{il}]^\beta} \qquad (5)$$

p_{ij}^k represents the probability of an ant *k* choosing a path (i,j), $\mathcal{N}_{i^k}$ the neighbors of node *i* not visited by ant *k* and all the $\mathcal{N}_{i^k}$ neighbors of node *i*, τ_{ij} the pheromone concentration on arc (i,j), and η_{ij} the inverse of the length of arc (i,j).

Once the next waypoint is chosen, if it was a pick-up location, its respective drop-off location is removed from the list of “illegal”. On the other hand, the pheromone of the chosen arc is updated using

$$\tau_{ij} = (1-\rho)\tau_{ij} + \rho\tau_0 \qquad (6)$$

Where ρ represents the evaporation rate and$\tau_0$is the fixed pheromone increment. Finally, after all ants have constructed their routes, a “daemon” adds pheromone over the arcs included in the shortest route found according to

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \rho\Delta\tau_{ij}(t) \qquad (7)$$

Where

$$\Delta\tau_{ij}(t) = \begin{cases} 1/L^+(t) & (i,j) \in T^+(t) \\ 0 & (i,j) \notin T^+(t) \end{cases} \qquad (8)$$

and L^+ represents the length of the shortest route, T^+.

Parameters:

Equations (3) through (8) show the large number of parameters that this algorithm has. These parameters greatly affect the behavior and performance of the algorithm. Using [10,11] as reference, we have done several test runs to find the parameters that suit our problem the best. The final values are $n_{iterations} = 10,\ m = 20,\ q_0 = 0.2,\ \rho = 0.2,\ \beta = 6\ and\ \tau_0 = 1/65.$

3.5.3. Results.

The objective of this work is to improve the performance of shuttle service with respect to the current state, in which a human operator in each of the vehicles plans the customers to serve next. Therefore, the results shown here compare the performance of our algorithm with the performance of a planning done by a human over the same list of customers.

The results have been obtained in a simulation environment specifically created for this work. The environment is formed by a 20×20 bi-dimensional space where customers are located and vehicles can move (Figure 1). In each step, the vehicles move a fixed distance v towards the next waypoint in their current route. If they reach a waypoint, they pick up or drop off a customer and proceed to the next waypoint. If the waypoint they reach is the end of their current route, they execute the customer allocation and ACO algorithms to calculate the next route. Every $customer\_frequency$ steps a new random customer appears and is added to the pool. The pseudo-code of the environment is shown in Algorithm 1.

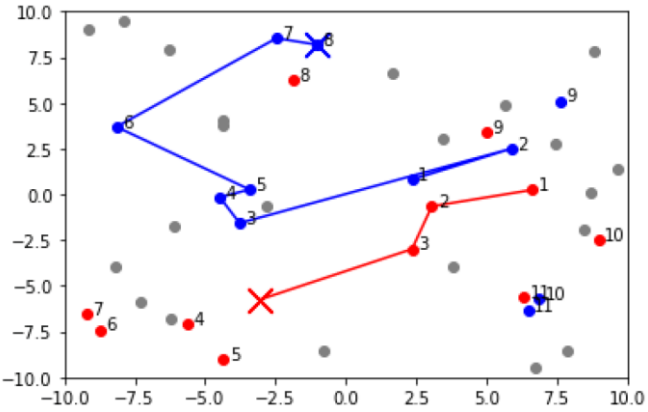

Figure 1. Snapshot of the simulation environment. Each vehicle (X), the route it is following, and the waypoints allocated to it is represented with a different color. The grey points are the rest of the customers' waypoints in the pool.

```
initialize vehicles;
initialize customer pool;
for t in T iterations do
    if mod(t,customer_frequency) = 0
     then
        add customer to pool;
    end
    for j vehicles do
        move 1 step towards current
         destination;
        if destination reached then
            set new destination;
            if end of current route then
                new customers ← customer
                 allocation algorithm;
                new route ← ACO;
            end
        end
    end
    for i customers do
        t_wait ← t_wait + 1;
    end
end
```

Figure 2. Algorithm 1: Simulation environment. The values of customer frequency and the speed of the vehicles have to be balanced to keep a steady state of customers and avoid an empty pool.

Table 1 (in the next page) shows the comparison of various metrics when the route planning is done by the algorithm and manually. The human planners designing manual routes were given a set of $k = 5$ customers previously chosen by the customer allocation algorithm. Therefore, this table compares only the route planning, not the customer allocation. The algorithm outperforms the manual planning in 8 out of 10 simulation runs. The algorithm can reduce the mean waiting time of customers (t_{mean}) up to a -21.5% with respect to the manual planning. The maximum difference in the cases in which the algorithm increases the mean waiting time is $+4.1\%$ .There are no significant differences in the total distance covered by the vehicles. This can be explained with the fact that the vehicles move at a constant speed and the total simulation time-steps is a fixed parameter. The simulation environment is ideal, so no delays are produced by traffic or customer pick-up or drop-off times. Therefore, the vehicles are expected to cover a similar distance in all the simulation runs. There are very little differences in the number of customers served during the simulations. This indicates there is not a big improvement in the route planning, since more efficient routes would allow the vehicles to serve more customers in the same simulation time.

Table 1. Comparison of results of the algorithm and solutions obtained manually over 400 simulation time-steps. t_{mean} is the mean waiting time of the customers (in simulation time-steps), and D_1 and D_2 the total distances traveled by the vehicles 1 and 2, respectively. n_{total} represents the number of customers served during the simulation time. Each Algorithm-Manual simulation pair has been obtained fixing the random generation of customers. The manual solutions have been designed by the authors of this work and two independent, external, test subjects.

Method	t_{mean}	D_1	D_2	n_{total}
Algorithm	84.73	348.8	327.6	55
Manual	87.50	352.8	368.0	60
Algorithm	75.58	334.3	366.8	60
Manual	81.67	361.2	399.5	60
Algorithm	65.67	361.1	343.0	60
Manual	85.67	352.9	402.1	60
Algorithm	66.31	351.7	365.9	65
Manual	84.50	356.0	365.9	60
Algorithm	79.17	371.2	366.3	60
Manual	76.08	352.2	364.9	60
Algorithm	73.83	369.8	355.7	60
Manual	77.75	360.2	399.4	60
Algorithm	60.00	385.0	365.3	65
Manual	70.08	364.8	376.3	60
Algorithm	62.92	333.1	352.7	60
Manual	68.92	337.3	354.5	60
Algorithm	81.17	368.5	378.6	60
Manual	84.64	337.2	345.4	55
Algorithm	74.58	352.6	373.9	60
Manual	71.25	375.0	365.6	60

Table 2. Comparison of results when using the clustering algorithm and the FIFO criterion for customer allocation in the simulation. Each simulation environment has been run 300 times and results have been averaged. t_{mean} is the average of the mean waiting

Method	t_{mean}	t_{std}	L_{mean}	L_{std}
Clustering	59.80	2.21	55.84	1.31
FIFO	76.86	1.77	61.03	1.35

To see to what extent the classifier is helping in the route planning, we have also compared the clustering algorithm and the human choosing criterion. The latter has been assumed to follow a *<First in, First Out> (FIFO)* criterion serving the oldest $k = 5$ customers in the pool. Table 2 shows the comparison in the mean waiting times and mean route lengths when both customer allocating strategies are used.

At first glance, one might think that picking up the longest waiting customers would reduce the average waiting time. However, by not considering their pick-up and drop-off locations, routes become more inefficient, which ultimately results in longer average waiting times. A

9.29% increase in average route length converts to a difference in wait time of $+28.53\%$. This demonstrates the effectiveness of the proposed clustering algorithm.

3.5.4. Conclusions of the task.

- Results have shown that the algorithm outperforms a human by hand's planning.
- The main difficulty of the problem is the allocation of the customers.
- It is "easy" for an individual to find a near optimal route with only 11 waypoints.
- If higher capacity vehicles were considered, the difficulty of the task would increase exponentially for a human. ACO would make bigger differences.
- The simplifications made for this first approach are far from reality. Future development of the project should consider road network, non-individual requests, and traffic conditions.

4. References

[1] Cagri Koc, G. Laporte, and Ilknur Tukenmez, "A review of vehicle routing with simultaneous pickup and delivery", *Computers & Operations Research*, vol.122, 2020.

[2] Y. Li, H. Solemani, and M. Zohal, "An improved and colony optimization algorithm for the multi-depot green vehicle routing problem with multiple objectives", *Journal of Cleaner Production*, vol.227, pp. 1161-1172, 2019.

[3] H. Xu, P. Pu, and F. Duan, "Dynamic vehicle routing problems with enhanced ant colony optimization", Discrete Dynamics in Nature and Society, vol. 2018, pp. 137-172, 2018.

[4] M. Charikar and B. Raghavachanari, "The finite capacity dial-a-ride problem", *in Proceedings 39th Annual Sympossium on Foundations of Computer Science* (Cat. No.98CB36280), pp. 458-467, Nov 1998.

[5] J.E. Bell and P.R. McMukllen, "Ant colony optimization techniques for the vehicle routing problem", Advanced Engineering Informatics, vol. 18, no. 1, pp. 41-48, 2004.

[6] B. Bullnheimer, R. Hartl, and C. Strauss, "An improved ant system algorithm for the vehicle routing problem", *Annals of OperationsResearch*, vol. 89, pp. 319-328, 1999.

[7] B. Yu, Z.-Z. Yang, and B. Yao, "An improved ant colony optimization for vehicle routing problem", European Journal of Operational Research, vol. 196, no. 1, pp. 171-176, 2009.

[8] R.Goel and R. Maini, "A hybrid of ant colony and firefly algorithms (hafa) for solving vehicle routing problems", *Journal of Computational Science*, vol. 25, pp. 28-37, 2018.

[9] M. Dorigo, G. Di Caro, and L. M. Gambardella, "Ant algorithms for discrete optimization", *Artificial Life*, vol. 5, pp. 137-172, 04, 1999.

[10] J. C. Molina, J. L. Salmeron, and I. Eguia, "An acs-based Memetic algorithm for the heterogeneous vehicle routing problem with time windows", *Expert Systems with Applications*, vol. 157, p. 113379, 2020.

[11] D. Gaertner and K. Clark, "On optimal parameters for ant colony optimization algorithms", *Proceedings of the 2005 International Conference on Artificial Intelligence*, vol. 1, pp. 83-89, 01 2005.

Annex 1. Python Code for the Automation of the process of changing the sorting format of the company's files to switch from an old data base system to a new SQL one.

```
'''FRANKLIN BRAZING & METAL TREATING'''
#Developed by Daniel Lazaro - 7/13/2021


from splitting_functions import file_paths, split_files



'''Specify File Directory and Extension'''
file_dir = r"C:\Users\Dani\Desktop\Franklin Brazing\Yokohama-20210706T171928Z-001"
file_ext = r".pdf"


'''Get the Paths and the Names of all the Files that are going to be Split'''
path_list, file_names = file_paths(file_dir, file_ext)


'''Specify the titles for the individual split files, and the key words that the program has
to search for in order to split the file correctly into the different operations/sheets that
form the WI'''
split_file_titles = ['Data-Sheet', 'Pre-Assembly', 'Knurl',
                     'Spot-Weld', 'Assembly', 'Furnace-Load-Sheet',
                     'Furnace-Off-Load-Insp', 'Furnace-Off-Line-Insp',
                     'Revision-Date', 'Print']

key_words = ['PART DATA-SHEET', 'Pre-Assembly', 'Knurling Sheet',
             'Spot-Weld', 'Assembly Sheet', 'Furnace Load Sheet',
             'Furnace Off-Load Inspection', 'Furnace Off-Line Inspection',
             'Revision Date', 'Print']


'''This Function splits the Files and saves them inside the folder called Split Files'''
split_files(file_dir, file_ext, path_list, file_names, key_words, split_file_titles)
```

```
from PyPDF2 import PdfFileWriter, PdfFileReader
import os
import slate3k as slate

#Funcion para saber en que pagina esta cada titulo
```

```
def listed_text(path_pdf):

    with open(path_pdf,'rb') as f:
        extracted_text = slate.PDF(f)
    pdftext = str(extracted_text)
    pdftext.replace('[', '')
    pdftext.replace(']', '')
    #pdftext.replace('\n', '')

    listedtext = pdftext.split('x0c', -1)
    listedtext.pop(-1)
    #clean not important text
    for i in range(len(listedtext)):
        listedtext[i] = listedtext[i][0:100]
    return listedtext


def pages_sheets_dic(listedtext, possible_areas):
    dic = {}

    for i in range(len(listedtext)):

        text = listedtext[i]
        no_area = 0
        for area in possible_areas:
            if area in text:
                dic[i] = area
            else: no_area += 1

        if no_area == len(possible_areas):
            print(i-1)
            dic[i] = dic[i-1]

        for j in listedtext:
            if len(j) == 0:
                print('Encontrado 0 len')
                dic[listedtext.index(j)] = 'Print'
                print(dic)
        page_print = get_key('Print', dic)

    return dic


# function to return key for any value
def get_key(val, dic):
    keys =[]
    for key, value in dic.items():
         if val == value:
             keys.append(key)
```

```
    return keys


def file_name(key, file_names, possible_areas):
    index = possible_areas.index(key)
    filename = file_names[index]
    return filename


def file_paths(file_dir, file_ext):

    filenames = [_ for _ in os.listdir(file_dir) if _.endswith(file_ext)]
    pathlist = [os.path.join(file_dir, name) for name in filenames]

    return pathlist, filenames


def split_files(file_dir, file_ext, path_list, file_names, possible_areas, split_file_titles):

    counter =0
    for path in path_list:
        print(path)

        inputpdf = PdfFileReader(open(path, "rb"))
        name = file_names[counter]
        n_name = len(name)
        print(name)
        listedtext = listed_text(path)
        dic = pages_sheets_dic(listedtext, possible_areas)

        sheets = list(set(dic.values()))

        for sheet in sheets:
            pages = get_key(sheet, dic)

            output = PdfFileWriter()
            if sheet != 'PART DATA-SHEET' :
                output.addPage(inputpdf.getPage(0))

            for i in pages:
                output.addPage(inputpdf.getPage(i))

            sheet1 = file_name(sheet, split_file_titles, possible_areas)
            namepdf = f'Split Files\{name[:n_name-4]}-{sheet1}.pdf'
            print(name)
            savepath = os.path.join(file_dir, namepdf)
```

```
            print(savepath)
            with open(savepath, mode ="wb") as outputStream:
                output.write(outputStream)

        counter += 1
```

Annex 2. Python Code for the live route planning algorithm for the company's shuttle service.

```
#

#This script contains the ACO algorithm (function name:ACO) and all the function
s it needs.
#It is Dorigo's proposed algorithm.
#It is call by the main_loop, where the needed parameters are specified in the lines 34-39.

import numpy as np
from random import random
from itertools import permutations

def calculate_distances(A,B):
    dx = A[0] - B[0]
    dy = A[1] - B[1]
    dist = np.array([dx,dy])
    return np.linalg.norm(dist)


def create_nodes(customers,car_position,k):
    nodes=[]
    for i in range(2*k):
        if i<k:
            node=[customers[i][0],customers[i][1]]
            nodes.append(node)
        else:
            node=[customers[i-k][2],customers[i-k][3]]
            nodes.append(node)
    nodes.append(car_position)
    return nodes


def ACO(k,customers,car_position,n_ants,alpha,beta,q0,ph_increment,evaporation_rate):

    customers=customers
    k=k
    alpha=alpha
    beta=beta
    car_position=car_position
    n_ants= n_ants
    n_iterations=10
    evaporation_rate=evaporation_rate
    ph_increment= ph_increment
    q0=q0
```

```
    pheromone_matrix,distance_matrix,nodes=initialize_ACO(k,customers,car_position,ph_increment)
    waypoints=list(range(len(nodes)))
    length_best=100000
    for i in range(n_iterations):
        routes, pheromone_matrix=generate_route(waypoints,k,pheromone_matrix,distance_matrix,alpha,beta
,evaporation_rate,ph_increment,n_ants,q0)
        new_best_route,new_length_best= best_route_and_length(routes,distance_matrix)
        pheromone_matrix=daemon_update(pheromone_matrix,new_best_route,new_length_best,evaporation_rate
)

        if new_length_best < length_best:
            length_best=new_length_best
            best_route=new_best_route

        route_nodes=order_nodes(best_route,nodes)

    return best_route,pheromone_matrix,distance_matrix,waypoints,route_nodes,length_best


def initialize_ACO(k,customers,car_position,ph_increment):
    nodes=create_nodes(customers,car_position,k)
    distance_matrix=np.zeros((2*k+1,2*k+1))
    pheromone_matrix=ph_increment*np.ones_like(distance_matrix) #Initial value of pheromone in every pa
th
    for i in range (2*k+1):
        for j in range (2*k+1):
            distance_matrix[i,j]=calculate_distances(nodes[i],nodes[j])
            if i==j: #or j+k==i:
                pheromone_matrix[i,j]=0   #Paths going from destination to origin
                distance_matrix[i,j]=10**5
    return pheromone_matrix,distance_matrix,nodes


def generate_route (waypoints,k,pheromone_matrix,distance_matrix,alpha,beta,evaporation_rate,ph_increme
nt,n_ants,q0):

    illegal= [waypoints[k:2*k] for i in range(n_ants)]
    position=[[waypoints[-1]] for i in range(n_ants)]
    routes=position


    for i in range(len(waypoints)-1):
        arcs=[]
        for j in range(n_ants):
            next_wpt= next_waypoint(routes[j],illegal[j],position[j][i],pheromone_matrix,distance_matri
x,alpha,beta,q0)
            routes[j].append(next_wpt)
            if next_wpt<k:
                illegal[j].remove(next_wpt+k)
```

```
            arcs.append([position[j][i],next_wpt])
            position[j][i+1]=next_wpt

        #Once every ant takes an arc, pheromone is updated
        increment_matrix=np.zeros_like(pheromone_matrix)
        pheromone_mat_copy=np.copy(pheromone_matrix)
        for arc in arcs:
            increment=pheromone_update(pheromone_mat_copy[arc[0],arc[1]],evaporation_rate,ph_increment)
            increment_matrix[arc[0],arc[1]]+=increment
            pheromone_matrix[arc[0],arc[1]]=0
        pheromone_matrix+=increment_matrix
    return routes, pheromone_matrix


def next_waypoint(route,illegal,position,pheromone_matrix,distance_matrix,alpha,beta,q0):
    pheromone=np.copy(pheromone_matrix[position])  #takes the row corresponding to the actual position
    pheromone[route]=0
    pheromone[illegal]=0
    attract_numerator=pheromone**alpha * (1/distance_matrix[position])**beta
    distance_matrix[position]
    attractiveness= attract_numerator/np.sum(attract_numerator)
    if random() <= q0:
        next_wpt = np.argmax(attractiveness)
    else:
        probabilities = attractiveness/ np.sum(attractiveness)
        next_wpt = np.random.choice(range(len(probabilities)),1, p=probabilities)
    return int(next_wpt)

def pheromone_update(pheromone_arc,evaporation_rate,increment):
    pheromone_arc = (1-evaporation_rate)*pheromone_arc + evaporation_rate*increment
    return pheromone_arc

def best_route_and_length(routes,distance_matrix):
    all_lengths=[]
    for i in range(len(routes)):
        route_length=0
        for j in range (len(routes[0])-1):
            arc_length=distance_matrix[routes[i][j],routes[i][j+1]]
            route_length += arc_length
        all_lengths.append(route_length)
    best_route=np.argmin(all_lengths)

    return routes[best_route],all_lengths[best_route]

def daemon_update(pheromone_matrix,best_route,length_best,evaporation_rate):
```

```
    pheromone_matrix=(1-evaporation_rate)*pheromone_matrix
    for i in range (len(best_route)-1):
       pheromone_matrix[best_route[i],best_route[i+1]]+=evaporation_rate*1/length_best
    return pheromone_matrix

def order_nodes(route,nodes):
    route_nodes=[]
    for waypoint in route:
        route_nodes.append(nodes[waypoint])
    return route_nodes

def route_length(route,distance_matrix):
    route_length=0
    for i in range (len(route)-1):
        arc_length=distance_matrix[route[i],route[i+1]]
        route_length += arc_length
    return route_length

def possible_routes(k):
    '''Generates all the combinations of nodes to create routes and filters the illegal routes'''
    #Not necessary in the ACO, but for making some tests
    perm = permutations(range(k*2))
    per=[]

    for i in perm:
        per.append((6,)+i)
    all_permutations = np.array(per)
    origin = [i for i in range(0,k)]
    dest = [i for i in range(k, 2*k)]
    zipped = zip(origin, dest)
    origin_dest_matrix = np.array(list(zipped))
    i = 0
    todelete = []
    for perm in all_permutations:
        for origindest  in origin_dest_matrix:
            if np.where(perm == origindest[0]) > np.where(perm == origindest[1]) :
                todelete.append(i)
        i+=1
    todelete = np.unique(todelete)
    all_permutations = np.delete(all_permutations, todelete, 0)

    return all_permutations


#%% TEST CODE
# if __name__ == "__main__":
#     customers=[[1,2,3,4,5],[6,7,8,9,0],[12,0,13,9,1]]
#     car_position=[2,1]
#     k=3
```

```
#     b,p,d,w,rn=ACO(k,customers,car_position,3,1,1)



#     all_routes=possible_routes(k)
#     all_lengths=[]
#     for route in all_routes:
#         length=route_length(route,d)
#         all_lengths.append(length)
#     idx=np.argmin(all_lengths) #No creo que vaya a haber dos rutas con la misma longitud y que sea ju
sto la minima
#     print('Shortest route:', all_routes[idx])
#     print('Best found route:', b)
```

```
#Clustering algorithm for customer allocation
#The vehicles are the centers of the clusters
#Setting waiting time of the vehicle to the highest one of the customers in the pool...
  #... gives more weight to the ones that have been in queue the longest
import numpy as np

def get_customers(vehicle_position, customer_list, k):
    """
    get_customers selects the k customers closer to a vehicle from the
    customer pool

    Parameters
    ----------
    vehicle_position : LIST
        [x,y] position of the vehicle.
    customer_list : LIST
        [customer_1, customer_2, ...].
    k : INT
        Number of customers to pick from customer_list.

    Returns
    -------
    selected_customers : LIST
        List with selected customers' vectors.
    new_pool : LIST
        list of customers from customer_list that have not been selected by
        get_customers.

    """

    ## find customer with maximum waiting time
```

```
    max_time = 0
    for customer in customer_list:
        time = customer[-1]
        if time > max_time:
            max_time = time

    ## define vehicle's vector
    vehicle = []
    vehicle.extend(vehicle_position)
    vehicle.extend(vehicle_position)
    vehicle.append(max_time)

    ## calculate distances from vehicle to each customer
    distances = []
    vehicle_np = np.array(vehicle)
    for customer in customer_list:
        customer_np = np.array(customer)
        distances.append(np.linalg.norm(vehicle_np - customer_np))

    idx = np.argsort(distances)
    idx = idx[0:k]

    ## list with customers
    selected_customers = []
    for i in range(k):
        selected_customers.append(customer_list[idx[i]])

    ## customer pool without the selected customers
    new_pool = list(customer_list)
    for j in sorted(idx, reverse=True):
        del new_pool[j]

    return selected_customers, new_pool




if __name__ == "__main__":

    customer_pool = [[3,4,5,6,0],
                     [1,2,7,8,3],
                     [-1,1,5,5,4],
                     [-5,-4,3,3,3],
                     [0,0,9,9,8],
                     [1,-5,5,-1,7]]

    get_customers([1,2], customer_pool, 3)
```

```python
#script use for selecting the 5 first customers in the pool when the FIFO criterion is used

import numpy as np

def get_customers(vehicle_position, customer_list, k):

    selected_customers = []
    for i in range(k):
        selected_customers.append(customer_list[i])

    ## customer pool without the selected customers
    new_pool = list(customer_list)

    del new_pool[0:k]

    return selected_customers, new_pool
```

```python
#Simulation environment for the live route planning algorithm

#Parameters of the environment are described in lines 28-31
#Parameters used in the ACO algorithm are described in lines 34-39

#The program outputs:
    #Mean waiting time of customers
    #Total distances traveled by each car
    #Customers served by each car

import numpy as np
import matplotlib.pyplot as plt
import random
from vehicle import Vehicle
from classifier import get_customers
from ACO_function import ACO
from plot_routes import plot_routes
from obtain_figures import plot_route
#from first_five import get_customers            #for FIFO criterion to be used in the selection of cus
tomers
                                                 #Deactivate line 17 if used
time=[]
length=[]

for i in range(1):             # This loop was used with 300 runs for the comparison (clustering VS FIF
O)
```

```
## PARAMETERS
simulation_time = 400      # time-steps for the simulation
customer_freq = 8          # simulation time-steps of interval between customer appearances
k = 5                      # customers to pick from pool
n_initial_customers = 20   # customers at t=0

## ACO PARAMETERS
ACO_alpha = 1              # controls the type of choice heuristic
ACO_beta = 6               # controls the type of choice heuristic
ACO_ants = 20              # number of ants
q0=0.2                     # probability of greedy choice
ph_increment=1/65          # fixed pheromone increment over an arc every time an ant takes the arc
evaporation_rate=0.2       # evaporation rate

random.seed(97)

routes=[]
def create_customer():
    new_customer = []
    ## add 4 random parameters for initial and final X and Y position
    for parameter in range(4):
        new_customer.append(random.uniform(-10,10))
    new_customer.append(0)  # set the waiting time
    return new_customer

def add_waiting_time(customers):
    delta = 0
    for customer in customers:
        delta += customer.pop(-1)
    return delta

if __name__ == "__main__":
    ###################  INITIALIZATION  ###################
    ## create vehicles
    vehicle1 = Vehicle()
    vehicle2 = Vehicle()
    vehicles = list([vehicle1, vehicle2])

    ## create initial pool of customers
    customer_pool = []
    for i in range(n_initial_customers):
        customer_pool.append(create_customer())
    customer_pool_global = list(customer_pool)

    ## Set initial routes for vehicles
    total_distance = [0,0]
    customers_list=[]
    for i in [0,1]:
        vehicle = vehicles[i]
```

```
            customers, customer_pool = get_customers(vehicle.position, customer_pool, k)
            customers_list.append(customers)
            # best_route,pheromone_matrix,distance_matrix,waypoints,route_nodes
            best_route,_,_,_,nodes,route_length = ACO(k,customers,vehicle.position.copy(),ACO_ants,ACO_
alpha,ACO_beta,q0,ph_increment,evaporation_rate)
            routes.append(route_length)
            total_distance[i] += route_length
            # plot_route(customers, vehicle.position, best_route)
            # plot_route(customers, vehicle.position, None)
            # print(best_route)
            vehicle.route=nodes.copy()
            nodes.pop(0)  # remove the current vehicle position
            vehicle.update_route(nodes)
            vehicles[i] = vehicle

        ## initialize metrics
        mean_t = 0
        mean_t_samples = 10
        customers_picked = [5,5]



        ###################  MAIN LOOP  ###################
        xx = []
        customer_pool_size = []
        for t in list(range(simulation_time)):

            ## add new random customer every customer_freq timesteps
            if t % customer_freq == 0:
                cust = create_customer()
                customer_pool.append(cust)
                customer_pool_global.append(cust)

            ## update vehicles
            for i in [0,1]:
                vehicle = vehicles[i]

                ## 1. move
                waypoint_flag = vehicle.move()

                ## If waypoint reached, change destination
                list_flag = False
                if waypoint_flag:
                    list_flag = vehicle.update_destination()

                ## If end of current customer list, find new customers
```

```
                if list_flag:
                    # get customers
                    # print('Vehicle %d end of route. Updating...' % i)
                    customers, customer_pool = get_customers(vehicle.position, customer_pool, k)
                    customers_list.append(customers)
                    customers_picked[i] += 5
                    mean_t += add_waiting_time(customers)
                    mean_t_samples += 5
                    # get new customers and route and update
                    best_route,_,_,_,nodes,route_length = ACO(k,customers,vehicle.position.copy(),ACO_a
nts,ACO_alpha,ACO_beta,q0,ph_increment,evaporation_rate)
                    routes.append(route_length)
                    total_distance[i] += route_length
                    #plot_route(customers, vehicle.position, best_route)
                    vehicle.route=nodes.copy()
                    nodes.pop(0)  # remove the current vehicle position
                    vehicle.update_route(nodes)

                vehicles[i] = vehicle

            # if t<=120:
            #     if t%2==0:
            #         plot_routes(vehicles,t,customer_pool)
            #increase waiting time of customers in pool
            for i in range(len(customer_pool)):
                customer_pool[i][-1] += 1

            if t % 10 == 0:
                xx.append(t)
                customer_pool_size.append(len(customer_pool))
                pass

        # fig, ax = plt.subplots()
        # plt.plot(xx, customer_pool_size)

        print('Mean waiting time: %.2f u' % (mean_t*1.0/mean_t_samples))
        print('Total distance traveled by the vehicles: %r' % total_distance)
        print('Customers picked by vehicles: %r' % customers_picked)
        #print('Mean best route length:', sum(routes)/len(routes))
        # time.append(mean_t*1.0/mean_t_samples)
        # length.append(sum(routes)/len(routes))
# mean_time=sum(time)/len(time)
# mean_length= sum(length)/len(length)
# print('Mean waiting time:', mean_time)
# print('Mean best route length:',mean_length)
# variance_t = sum([((x - mean_time) ** 2) for x in time]) / len(time)
# res_t = variance_t ** 0.5
# variance_l = sum([((x - mean_length) ** 2) for x in length]) / len(length)
# res_l = variance_l ** 0.5
```

```
# print('Mean std time:', res_t)
# print('Mean best route std:',res_l)
```

```
#This script is used for getting the humand by hand routes.
#The random seed in line 26 must be the same as the seed in line 36 in the main_loop.
#When run, the user will be asked to complete several routes.
#A plot with numbered nodes is displayed for each new case:
    #Triangles represent pick-up locations.
    #Stop signs represent drop-off locations.
    #The vehicle (starting point) is represented with a diamond.
#User has to introduce manually the sequence of nodes that make the route, each number separated by a b
lank space
#Finally, the program outputs:
    #Mean waiting time of customers
    #Total distances traveled by each car
    #Customers served by each car

import numpy as np
import matplotlib.pyplot as plt
import random
from vehicle import Vehicle
from classifier import get_customers
from ACO_function import ACO, route_length, initialize_ACO, order_nodes
from plot_routes import plot_routes
from obtain_figures import plot_route

## PARAMETERS
simulation_time = 400     # time-steps for the simulation
customer_freq = 8         # simulation time-steps of interval between customer appearances
k = 5                      # customers to pick from pool
n_initial_customers = 20   # customers at t=0

## ACO PARAMETERS
ACO_alpha = 1
ACO_beta = 6
ACO_ants = 20

random.seed(97)

def create_customer():
    new_customer = []
    ## add 4 random parameters for initial and final X and Y position
    for parameter in range(4):
        new_customer.append(random.uniform(-10,10))
    new_customer.append(0)  # set the waiting time
```

```
    return new_customer

def add_waiting_time(customers):
    delta = 0
    for customer in customers:
        delta += customer.pop(-1)
    return delta

def get_user_route():
    error_flag = True
    while error_flag:
        raw_input_string = input('Answer: ')
        ls = raw_input_string.split()
        for i in range(len(ls)):
            ls[i] = int(ls[i])

        # Check if length is correct
        error_flag = False
        if len(ls) != 10:
            error_flag = True
            print('Error! The length of the list is incorrect.')
            continue

        # Check for illegal routes
        len_half = int(len(ls)/2)
        for j in range(len_half):
            dif = ls.index(j+len_half) - ls.index(j)
            if dif < 0:
                error_flag = True
                print('Error! Illegal route.')
                break
    return ls


if __name__ == "__main__":
    ###################  INITIALIZATION  ###################
    ## create vehicles
    vehicle1 = Vehicle()
    vehicle2 = Vehicle()
    vehicles = list([vehicle1, vehicle2])

    ## create initial pool of customers
    customer_pool = []
    for i in range(n_initial_customers):
        customer_pool.append(create_customer())
    customer_pool_global = list(customer_pool)

    ## Set initial routes for vehicles
    total_distance = [0,0]
```

```
    customers_list=[]
    for i in [0,1]:
        vehicle = vehicles[i]
        customers, customer_pool = get_customers(vehicle.position, customer_pool, k)
        customers_list.append(customers)
        # best_route,pheromone_matrix,distance_matrix,waypoints,route_nodes
        plot_route(customers, vehicle.position, None)  # give the user the points
        best_route = get_user_route()  # the user chooses the route
        best_route.insert(0,len(best_route))  # add the vehicle as the first stop in the route
        _,distance_matrix,nodes = initialize_ACO(k, customers, vehicle.position.copy(),1/65)
        route_dist = route_length(best_route, distance_matrix)
        # best_route,_,_,_,nodes,route_length = ACO(k,customers,vehicle.position.copy(),ACO_ants,ACO_al
pha,ACO_beta)
        total_distance[i] += route_dist
        plot_route(customers, vehicle.position, best_route)

        nodes = order_nodes(best_route, nodes)
        vehicle.route=nodes.copy()
        nodes.pop(0)  # remove the current vehicle position
        vehicle.update_route(nodes)
        vehicles[i] = vehicle

    ## initialize metrics
    mean_t = 0
    mean_t_samples = 10
    customers_picked = [5,5]



    ###################  MAIN LOOP  ###################
    xx = []
    customer_pool_size = []
    for t in list(range(simulation_time)):

        ## add new random customer every customer_freq timesteps
        if t % customer_freq == 0:
            cust = create_customer()
            customer_pool.append(cust)
            customer_pool_global.append(cust)

        ## update vehicles
        for i in [0,1]:
            vehicle = vehicles[i]

            ## 1. move
            waypoint_flag = vehicle.move()
```

```
            ## If waypoint reached, change destination
            list_flag = False
            if waypoint_flag:
                list_flag = vehicle.update_destination()

            ## If end of current customer list, find new customers
            if list_flag:
                # get customers
                customers, customer_pool = get_customers(vehicle.position, customer_pool, k)
                customers_list.append(customers)
                mean_t += add_waiting_time(customers)
                mean_t_samples += 5
                customers_picked[i] += 5
                # best_route,pheromone_matrix,distance_matrix,waypoints,route_nodes
                print('Iteration %d/%d' % (t, simulation_time))
                plot_route(customers, vehicle.position, None)  # give the user the points
                best_route = get_user_route()  # the user chooses the route
                best_route.insert(0,len(best_route))  # add the vehicle as the first stop in the route
                _,distance_matrix,nodes = initialize_ACO(k, customers, vehicle.position.copy(),1/65)
                route_dist = route_length(best_route, distance_matrix)
                # best_route,_,_,_,nodes,route_length = ACO(k,customers,vehicle.position.copy(),ACO_ant
s,ACO_alpha,ACO_beta)
                total_distance[i] += route_dist
                plot_route(customers, vehicle.position, best_route)

                nodes = order_nodes(best_route, nodes)
                vehicle.route=nodes.copy()
                nodes.pop(0)  # remove the current vehicle position
                vehicle.update_route(nodes)
                vehicles[i] = vehicle



                # print('Vehicle %d end of route. Updating...' % i)
                # customers, customer_pool = get_customers(vehicle.position, customer_pool, k)
                # mean_t += add_waiting_time(customers)
                # mean_t_samples += 3
                # # get new customers and route and update
                # best_route,_,_,_,nodes,route_length = ACO(k,customers,vehicle.position.copy(),ACO_ant
s,ACO_alpha,ACO_beta)
                # total_distance[i] += route_length
                # #plot_route(customers, vehicle.position, best_route)
                # vehicle.route=nodes.copy()
                # nodes.pop(0)  # remove the current vehicle position
                # vehicle.update_route(nodes)

            vehicles[i] = vehicle

        # if t<=120:
```

```
        #     if t%2==0:
        #         plot_routes(vehicles,t,customer_pool)
        # increase waiting time of customers in pool
        for i in range(len(customer_pool)):
            customer_pool[i][-1] += 1

        if t % 10 == 0:
            xx.append(t)
            customer_pool_size.append(len(customer_pool))
            pass

    # fig, ax = plt.subplots()
    # plt.plot(xx, customer_pool_size)

    print('Mean waiting time: %.2f u' % (mean_t*1.0/mean_t_samples))
    print('Total distance traveled by the vehicles: %r' % total_distance)
    print('Customers picked by vehicles: %r' % customers_picked)
```

```
#script use for plotting the routes obtained by the ACO
#A plot with numbered nodes is displayed:
    #Triangles represent pick-up locations.
    #Stop signs represent drop-off locations.
    #The vehicle (starting point) is represented with a diamond.

import matplotlib.pyplot as plt

def plot_route(customers, car_position, route):

    ## Extract initial and final points from customers
    initial_nodes = []  # [x,y]
    final_nodes = []   #[x,y]
    for customer in customers:
        initial_nodes.append(customer[0:2])
        final_nodes.append(customer[2:4])

    ## Create list with all nodes for route and rearrange
    if route != None:
        nodes = []
        nodes.extend(initial_nodes)
        nodes.extend(final_nodes)
        nodes.append(car_position)
        nodes_arranged = []
        for node_idx in route:
            nodes_arranged.append(nodes[node_idx])
```

```
        nodes_arranged_plot = [[],[]]
        for i in range(2*len(final_nodes)+1):
            nodes_arranged_plot[0].append(nodes_arranged[i][0])
            nodes_arranged_plot[1].append(nodes_arranged[i][1])

    fig, ax = plt.subplots()

    ## plot nodes
    plt.gca().set_prop_cycle(None)  # reset color cycle
    for i in range(len(initial_nodes)):
        plt.scatter([initial_nodes[i][0]], [initial_nodes[i][1]],
                    marker='^',s=105)
        if route == None:
            plt.text(initial_nodes[i][0], initial_nodes[i][1], s=str(i))
    plt.gca().set_prop_cycle(None)  # reset color cycle
    for i in range(len(final_nodes)):
        plt.scatter([final_nodes[i][0]], [final_nodes[i][1]],
                marker='8',s=105)
        if route == None:
            plt.text(final_nodes[i][0], final_nodes[i][1], s=str(i+5))



    ## plot car
    plt.scatter([car_position[0]], [car_position[1]],
                marker='d', s=105, c=0)

    ## plot route
    if route != None:
        plt.plot(nodes_arranged_plot[0], nodes_arranged_plot[1],
                 '--', c='chocolate', zorder=-5)

    plt.xlim([-10,10])
    plt.ylim([-10,10])
    plt.show()



if __name__ == "__main__":
    customers=[[1,2,3,4,5],[6,7,8,9,0],[12,0,13,9,1]]
    route = [6,0,3,1,4,2,5]

    plot_route(customers, [5.5, -1], route)
```

```
#script use for plotting the routes and customer pool for making the simulation time-lapse
    #Customers in the pool are represented with grey dots
    #Vehicle position is represented with an X and its allocated customers and traveled route is...
       #...represented in the same color (red/blue)
```

```python
import matplotlib.pyplot as plt


def plot_routes(vehicles,t,customer_pool):

    for cust in customer_pool:
        plt.scatter(cust[0], cust[1],color='gray')
        plt.scatter(cust[2], cust[3],color='gray')
    for (i,c) in zip([0,1],['b','r']):

        j=1
        for coord in vehicles[i].route:
                plt.scatter(coord[0], coord[1],color=c)
                plt.scatter(vehicles[i].position[0],vehicles[i].position[1], marker='x',color=c,s=200)
                plt.text(coord[0]+0.2, coord[1], '{}'.format(j))
                j+=1

        missing=len(vehicles[i].current_route)+1 #current route doesn't consider current_destination
        completed=vehicles[i].route[0:len(vehicles[i].route)-missing]+[vehicles[i].position]

        x=[]
        y=[]

        for pos in completed:

            x.append(pos[0])
            y.append(pos[1])
        plt.plot(x,y,c=c)
    plt.title('t={}'.format(t))
    plt.xlim([-10,10])
    plt.ylim([-10,10])
    #plt.savefig(f'routes/{t}.png', dpi = 1000)
    plt.show()
```

```python
#The class vehicle is defined:
    #ATRIBUTES:
        # position
        # current destination
        # current route: route already traveled
        # route: route assigned
    #METHODS:
        # move
        # update destination
```

```
        # update route

import numpy as np

class Vehicle(object):

    ## Vehicle parameters
    v = 0.87  # vehicle speed, we need to find a reasonable value

    def __init__(self):
        self.position = [0, 0]  # vector with vehicle position
        self.current_destination = [0, 0]
        self.current_route = []  # [[wpt1_x,wpt1_y], [wpt2_x,wpt2_y], ...]
        self.route=[]

    def move(self):
        """
        'move' moves the vehicle one step towards the current destination.

        Returns
        -------
        reached_destination_flag : BOOLEAN
            The flag will be True when the vehicle reaches the destination.
            Otherwise, it will be False.

        """
        destination = self.current_destination

        ## calculate direction
        dx = destination[0] - self.position[0]
        dy = destination[1] - self.position[1]
        dest_vector = np.array([dx,dy])
        dist = np.linalg.norm(dest_vector)  # distance to the next waypoint
        u = dest_vector / dist  # u is the unit direction vector

        ## avoid overshooting
        # if distance is larger than speed, advance a distance equal to speed
        # otherwise, reduce the speed to the distance
        if dist > self.v:
            vel = self.v * u
            reached_destination_flag = False
        else:
            vel = dist * u
            reached_destination_flag = True

        ## update position in x and y
        for i in [0,1]:
            self.position[i] = self.position[i] + vel[i]
```

```
        # check algorithm
        # print('New position: %r' % self.position)
        dest = np.array(destination)
        # print('New distance: %.5f' % np.linalg.norm(self.position - dest))

        return reached_destination_flag

    def update_destination(self):
        """
        update_destination sets a new current destination for the vehicle

        Returns
        -------
        end_of_route_flag : BOOLEAN
            Returns True if the end of the route is reached. Returns
            False if there are more destinations in the current route.

        """
        if self.current_route == []:
            end_of_route_flag = True
        else:
            self.current_destination = self.current_route.pop(0)
            end_of_route_flag = False
            # print('Destination reached. Moving to next waypoint: %r' % self.current_destination)
        return end_of_route_flag

    def update_route(self, new_route):
        """
        update_route sets a new route for the vehicle

        Parameters
        ----------
        new_route : LIST
            New route for the vehicle.
            The format must be
            [[waypoint1_x,waypoint1_y], [waypoint2_x,waypoint2_y], ...]

        """
        self.current_route = new_route
        self.update_destination()




if __name__ == "__main__":
```

```
import numpy as np

car = Vehicle()
route = [[4,5], [3,1], [3,8]]
car.update_route(route)
route_flag = False

while(1):
    flag = car.move()
    if flag:
        route_flag = car.update_destination()
    if route_flag:
        print('Route finished')
        break
```