Application of the ant colony algorithm for a truck delivery distributed system

Fabrizio Corriera

30/11/2023

Contents

1 Introduction

In this work we explore the application of the ant colony algorithm to propose a vehicle routing model based on a dynamic pick-up and delivery problem (DPDP) inside of a distributed system.

The premise of this work is an experimentation about the application of a routing algorithm based on a heuristic model.

A heuristic model is a problem-solving or decision-making approach that relies on practical rules, intuition, and experience rather than strictly adhering to an optimal algorithmic solution. Heuristic models are employed when finding an exact solution is computationally expensive or impractical. Instead of exhaustively exploring all possible solutions, heuristics aim to quickly identify a satisfactory solution.

In the context of routing algorithms, heuristics play a crucial role in optimizing the pathfinding process. Routing algorithms determine the most efficient path from a source to a destination in a network or graph. Heuristic models in routing algorithms prioritize speed and practicality over guaranteeing the absolute shortest path. The purpose is to rapidly find a reasonably good solution without the need for exhaustive computation, which is particularly important in real-time applications.

The idea behind using heuristics in routing algorithms is to strike a balance between efficiency and accuracy. By leveraging informed estimates and rules of thumb, heuristic models can significantly reduce the computational overhead associated with finding optimal routes, making them well-suited for dynamic and resource-constrained environments. While the solutions provided by heuristic models may not always be optimal, they often offer a satisfactory trade-off between computational complexity and solution quality, making them practical for a wide range of applications.

2 Functional requirements

The project functional requirements are:

- The main routing algorithm of the system must be a heuristic algorithm.
- The heuristic algorithm must be able to find a path for the vehicle in a reasonable amount of time.
- The tests must be run keeping in mind a real world scenario.
- The system is composed by many nodes that correspond to the vehicles.

3 Tools used

The tools used in this project are:

- **Python:** Python is a high-level, interpreted programming language known for its simplicity and readability. It emphasizes code readability and allows developers to express concepts in fewer lines of code.
- **Ant colony algorithm:** The ant colony optimization algorithm is a nature-inspired optimization technique that mimics the foraging behavior of ants. It involves a population of artificial ants exploring and marking paths with pheromones to find the optimal solution to a given problem. The algorithm utilizes the principle of collective intelligence and pheromone-based communication to iteratively converge towards an efficient solution.
- **Open Street Map:** OpenStreetMap (OSM) is a collaborative, open-source mapping platform that allows users to create, edit, and share geographic data globally. It provides a free and editable map of the world, relying on contributions from a diverse community of mappers. OSM is widely used for applications such as navigation, geolocation, and urban planning.
- **OSMnx:** OSMnx is a Python package built on top of NetworkX and GeoPandas, and interacts with OpenStreetMap APIs that allows us to download and model street networks or other infrastructure anywhere in the world with geospatial features.
- **JSON:** JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. It uses key-value pairs and supports nested structures, providing a widely adopted standard for data exchange between web servers and clients.

4 Project structure

During the designing phase of the project we try to work keeping in mind a few key concepts:

- Modularity: we want to develop this project in a way that we can easily modify it or add features with relative ease.
- Efficiency: knowing that this application should run on mobile devices, we try to develop a code that keeps computational costs at minimum.
- Decentralization: as far as possible we try to keep this system as decentralized as possible, minimizing failure points and keeping communications between devices as light as possible.

So, we will present the project structure keeping in mind two different representations:

1. The **Vehicle** as its own environment.

2. The vehicle as a node in the **Network**.

4.1 The vehicle

The vehicle in this project is represented as the set of all its properties, specifically:

- Position
- Storage
- Energy
- Scheduled path

So, we create the **Truck** class for keeping track of all these internal variables regarding the vehicle status.

From this point on we start implementing in the class some methods that will be useful in the general schema of the project: mainly we focus on the path of the vehicle, expanding it in many ways, until we realize that we need dedicated data structures for it, and that's why we proceed to implement new classes for the management of the vehicle path.

Then another interesting consideration was how to deal with the algorithms and computations involving how to find the ideal path for the vehicle. Considering that these kind of computations would require the management of a lot of resources not directly related to the vehicle itself, we decided to implement another class for the pathfinding.

4.2 Route management

During the development of the project we realize the need for a class, or more classes, that might help us with the management of the vehicle's path. These classes must answer to these needs:

- We need a class that helps us abstract the route list of the vehicle.
- The route list must be sort of a middle point between human representation and easy to compute data.
- We need to micromanage different aspects of this data structure.
- We must also think of a way to make the possible communication of this data easy.

From all these considerations we realized that what we needed was to create a hierarchical structure of classes:

- **RouteList**: This is the object with which the vehicle class interacts directly, using its methods and its structure to interact and manage the list of stops that are scheduled.

  - **Job**: This is kind of a middle object, the one containing information about a full job (withdrawal+delivery). This object is useful mainly for communication between nodes and server, giving us a useful and common way to communicate new jobs.
    * **Stop**: Stop is the brick class, basically a RouteList is a list of Stop objects, with some high level methods that allow us to manage the sigle stops inside of it. The Stop object should be very transparent, and almost invisible to the other classes, allowing to deal with it just through the RouteList object.

4.3 Pathfinding

For all the pathfinding related issues we implemented a class called RouteCalculator, that, from the road graph of the town, and the access to the Truck object can do all the computations we need for solving the already mentioned pathfinding related problems.

The 3 key features we want to implement are:

1. Resolving the smaller pathfinding problems related to the paths from point to point.
2. Being able to compute the ant colony algorithm locally.
3. Using a greedy algorithm to compute the smaller deviations to add new nodes in an already existing path.

So, the way we imagined this class is using certain methods to obtain the information regarding the city streets, their length, their speed limits and average speeds and travel times, and then we use those information as support to the two main computation algorithms (greedy algorithm and ant colony algorithm), this way we don't need to know about the complexity of every computation behind it.

4.4 Networking

The networking design for this project is very simple; considering the following principles:

- We want our system to be as decentralized as possible
- We still need a central point to dispatch requests
- We want the system to be able to work in case of failure of any $n-1$ nodes
- We want to minimize clients-server communication

A **P2P network** with a slightly modified **discovery server** seems like the optimal solution.

The nodes will communicate with the server just when joining the network (during initialization), obtaining the list of already connected nodes, and then they will be able to communicate with each other.

The server will not be a pure discovery server, as intended in a normal P2P network, but it will also be able to send job requests to the nodes (in broadcast). Since we want to minimize server-nodes traffic, we will let the nodes decide in that case which node will take the job and then just contact the server to inform it of the decision.

5 Development

We will now show the code implementation for the project as we described it in the previous section. We will explain the code and our design choices class by class:

1. Stop (plus StopEncoder and StopDecoder)
2. Job (plus JobEncoder and JobDecoder)
3. RouteList
4. Truck
5. RoutesCalculator
6. Plotter
7. Node
8. Server

5.1 Stop

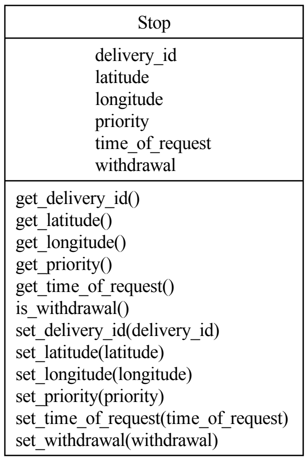

Figure 1: Attributes and methods of the Stop class

- **delivery_id**: This is a unique value used to identify the correlation between a withdrawal and a delivery.
- **latitude**: Geographic coordinate for the stop relative to the distance form the equator.
- **longitude**: Geographic coordinate for the stop relative to the distance form the reference meridian.
- **priority**: A numerical value that gives us the priority of the stop (default=1, high priority>1, low priority $\in [0,1[$).
- **time_of_request**: This datetime variable gives us the time at which the request was forwarded (this might be useful in order to boost the priority of old requests).
- **withdrawal**: Boolean flag value that informs us if the stop is actually a withdrawal (True) or a delivery (False).

The methods for this class are just setters and getters of the above mentioned attributes. Basically this class is our data structure for all the useful informations regarding the stops.

5.2 Job

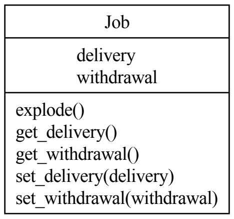

Figure 2: Attributes and methods of the Job class

- **delivery**: The stop object containing information about the withdrawal operation of this job.
- **withdrawal**: The stop object containing information about the delivery operation of this job.

This class exist mainly as a tool for communication between server and nodes (when the server has to inform the nodes of a new job request). One main drawback regarding this implementation may be about the use of just one withdrawal object and one delivery object, but actually the code is implemented in a way that we could easily adapt this class using more than one delivery (or even more than one withdrawal); it could be easily adapted.

The only method worth mentioning is **explode**, that just returns the single stops.

5.3 RouteList

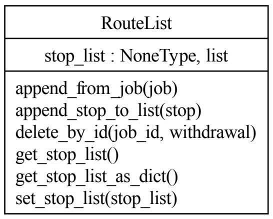

Figure 3: Attributes and methods of the RouteList class

- **stop_list**: This is a list object containing multiple Stop objects.

Though this might seem just a container for a list, it allows us to implement as many methods for the management of the stop list as we need to:

- **append_from_job**: This method simply appends a new stop to the list, but also might allow us to add many adjustments as we would see fit (for example we might add a control over some values of the Stop object and act differently as needed).
- **append_stop_to_list**: This method allows us to add to the list new Stop objects passing the Job object without any need for any other operations.
- **delete_by_id**: We also want to simplify the eventual need to delete objects from the list, and we implemented this method to do so querying the ids of the Stop objects in list. The second parameter, withdrawal, is a flag used to specify what element with the specified id we would like to delete:
  - 0: just the withdrawal
  - 1: just the delivery
  - 2: both
- **get_stop_list_as_dict**: This method may be useful for the specific need to have a dict instead of having a list of Stop items. We implemented it mainly for the ant colony algorithm, but it may be useful for future updates.

5.4 Truck

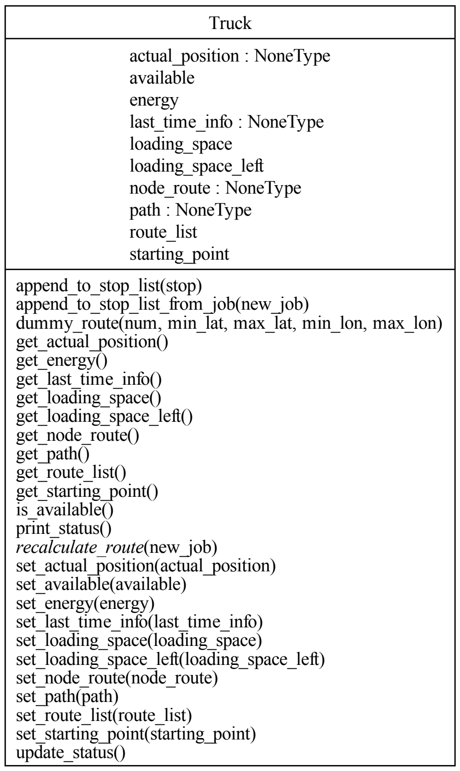
Truck
actual_position : NoneType
available
energy
last_time_info : NoneType
loading_space
loading_space_left
node_route : NoneType
path : NoneType
route_list
starting_point
append_to_stop_list(stop)
append_to_stop_list_from_job(new_job)
dummy_route(num, min_lat, max_lat, min_lon, max_lon)
get_actual_position()
get_energy()
get_last_time_info()
get_loading_space()
get_loading_space_left()
get_node_route()
get_path()
get_route_list()
get_starting_point()
is_available()
print_status()
recalculate_route(new_job)
set_actual_position(actual_position)
set_available(available)
set_energy(energy)
set_last_time_info(last_time_info)
set_loading_space(loading_space)
set_loading_space_left(loading_space_left)
set_node_route(node_route)
set_path(path)
set_route_list(route_list)
set_starting_point(starting_point)
update_status()

Figure 4: Attributes and methods of the Truck class

- **actual_position**: A tuple containing the coordinates of the actual position of the vehicle.
- **available**: A boolean value that informs us of the status of the vehicle (True = available, False = not available).
- **energy**: The percentage of energy left to the vehicle. This might be useful in order to establish if the vehicle may take new orders or not.
- **last_time_info**: A timestamp needed to save the last time the vehicle updated its stats (mainly useful for debugging).
- **loading_space**: The total loading space available to the vehicle.
- **loading_space_left**: The remaining loading space available to the vehicle.
- **node_route**: This list of lists contains the information regarding the sorted path of the vehicle in a format readable by the OSMnx library.

- **path**: The sorted path of the vehicle.
- **route_list**: The RouteList object of the vehicle.
- **starting_point**: The coordinates of the starting point of the vehicle path. This would correspond to the warehouse or the depot where the vehicles are kept when not in use.

Aside from the setters and getters methods of the class we also implemented other methods useful for an abstract management of our resources, without need to access directly to the other classes.

- **append_to_stop_list**: This method directly calls the RouteList method append_stop_to_list, to add to the vehicle route_list object the new stop.
- **append_to_stop_list_from_job**: This method directly calls the RouteList method append_from_job, to add to the vehicle route_list object two new stops from a Job object.
- **dummy_route**: This is a method useful for debugging. It returns us a specified number of fake Jobs, created from random coordinates in the format (latitude, longitude) in the interval [min_lat, max_lat] and [min_lon, max_lon].
- **print_status**: In case we may need to print the stas of the vehicle in a human readable way (if the truck is running correctly, the actual timestamp, the energy left, the loading space left and the actual position of the vehicle).
- **recalculate_route**: This is actually an unused method, but the idea would be to be able to call the methods needed to recalculate the route of the vehicle directly from the Truck object instead of calling them explicitly from RoutesCalculator.
- **update_status**: This method currently just works for debugging: it reads the stats of the vehicle from a file, but ideally it would be a method that would work in a while loop reading the data from sensors on the vehicle.

5.5 RoutesCalculator

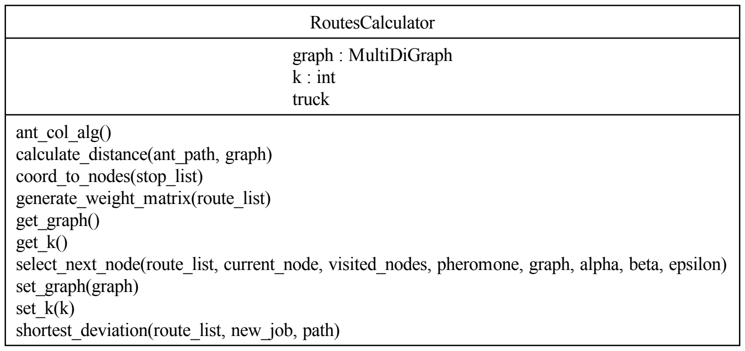
RoutesCalculator
graph : MultiDiGraph
k : int
truck
ant_col_alg()
calculate_distance(ant_path, graph)
coord_to_nodes(stop_list)
generate_weight_matrix(route_list)
get_graph()
get_k()
select_next_node(route_list, current_node, visited_nodes, pheromone, graph, alpha, beta, epsilon)
set_graph(graph)
set_k(k)
shortest_deviation(route_list, new_job, path)

Figure 5: Attributes and methods of the RoutesCalculator class

- **graph**: This is the OSMnx graph object, that is actually a NetworkX directed graph object. We obtain it from the OSMnx library passing directly the name of the place which we need the map of, specifyingwhat kind o graph we want, and then we call different methods to add to nodes and edges the values we need, like the edge weight based on the geographical distance between nodes.
- **k**: This value is currently unused, but it would be used in an hypothetical version of the RoutesCalculator object that allow us to calculate the best path considering the k shortest paths between each node. The reason why we did not implement it is because we need to keep performance in mind, and the ant colony algorithm in this implementation is already per se an algorithm with exponential computational complexity, and adding this variable to the algorithm would increase the exponential growth of the algorithm's complexity.
- **truck**: The Truck opject to which the calculator refers to. Thanks to this reference to the Truck object, the RoutesCalculator may read the attributes of the vehicle needed to compute the paths efficiently. We preferred to keep two separate objects as a modularity oriented design choice.

The following methods of this class are two main algorithms of the project, the ant colony algorithm and the greedy algorithm (shortest_deviation), along with the other methods used to allow their use.

- **ant_col_alg**: This is the main core of the project. It does not return a value, but directly sets the path and the node_route attributes of the Truck object it refers to. This algorithm will be best explained in a further section.

- **calculate_distance**: Subordinate method of ant_col_alg, it will also be best explained in a further section.
- **coord_to_nodes**: This method allow us to convert the human readable coordinates of the stops objects to node values, readable by the OSMnx library and the OSMnx graph object. It uses the nearest_nodes OSMnx method.
- **generate_weight_matrix**: This method returns the weight matrix of the subgraph whose nodes are only our stop objects. It is basically an adjacency matrix where all the edges are characterized by the weight of the edge. Actually the specified weight measure for this is the distance between nodes, but the OSMnx method shortest_path can take any weight as the cost measure for this optimization problem. The shortest_path method uses the Dijkstra algorithm to find the optimal path between nodes; obviously this method's computational cost increases with the graph's dimension.
- **select_next_node**: Subordinate method of ant_col_alg, it will also be best explained in a further section.
- **shortest_dev**: This method is a greedy algorithm that, having an already existing path and its cost, it adds a new withdrawal point in the path, just inserting it in the path, following the node closest to it. Then it does the same with the delivery point, searching the nearest node to it whose position in the path is set after the relative withdrawal. Then it computes the new weight of the path and returns it along the updated path (this method does not directly update the Truck object attributes because ideally it would first need to be elected by the network as the less costly path).

5.6 Plotter

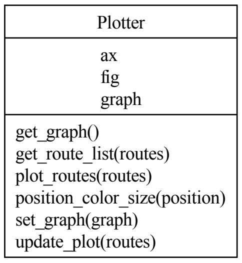

Figure 6: Attributes and methods of the Plotter class

This class is an helper for the plotting of the graph and the routes, it has no actual utility beside being a graphical tool for the project.

- **ax, fig**: This are items used by the matplolib library to plot and update existing plots without having to delete them and then plot again.
- **graph**: This is the same OSMnx graph object used by RoutesCalculator.

5.7 Node

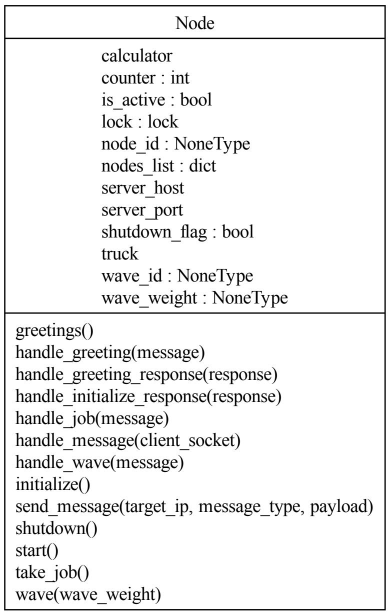

Figure 7: Attributes and methods of the Node class

This is the networking interface object for the vehicle used to communicate with the server and the other nodes; developed just for debugging purposes, it uses socket objects for communication.

- **calculator**: This is a reference to the RoutesCalculator object used in the main program.
- **counter**: This is a counter for the wave algorithm implemented for the election of the node that will take a job request submitted by the server. When the counter reaches a value equal to the number of nodes in the network, it knows that it has been elected.
- **is_active**: Flag value used to determine if the node is actually considerable active or not. Ideally this would be linked to certain vehicle's attributes like energy or storage left.

- **lock**: This object is used to allow threading in a safe way, avoiding race conditions to certain memory areas.
- **node_id**: THe id of the node, given by the server, after joining the network.
- **nodes_list**: The list of all the other nodes in the network, containing also information about the other nodes status.
- **server_host**: The host of the server.
- **server_port**: The port of the server.
- **shutdown_flag**: This flag is used to gracefully shutdown the node functions and running threads. The threads while in their running loop always check if this flag is set to True, if so, they will stop working.
- **truck**: The reference to the Truck object.
- **wave_id**: This is the id adopted by the node in the wave algorithm. When the algorithm starts, the wave_id is equal to the node_id, but if it adopts a new node election, then it will change its id to that node id.
- **wave_weight**: This is the weight adopted by the node in the wave algorithm. When the algorithm starts, the wave_weight is equal to the one computed by the node with the RoutesCalculator's shortest_deviation method, but if it receives a message with a weight lower than its own, it will adopt that node election, changing its own wave_weight to the that of that node.

The Node methods can be divided in two categories: the initialization methods and the wave methods.

The first category consists of communications between nodes and server to get initialized and the greetings messages to the other nodes.

The second category includes all the methods regarding the handling of a job request from the server and the subsequent wave algorithm with the other nodes.

- **greetings**: The node, in this method, sends a greeting message to all the nodes it has in its node_list. If after a timeout it does not receive an answer, it will mark the node as inactive.
- **handle_greeting**: Upon receiving a greeting message, the node save the new node information in its node_list and marks it as active. After that it responds with a greeting_response message.
- **handle_greeting_response**: Upon receiving a greeting response, the node will mark the corresponding sender id in node_list as active.
- **handle_initialize_response**: Upon receiving an initialize response, the node will save the received node list and its own new id.

- **handle_job**: Upon receiving a job request from the server, the node will call the shortest_deviation method from the RoutesCalculator object. After calculating the new weight, it will enter the wave algorithm.
- **handle_message**: This is the method that gets called whenever the node receives a message. Depending on the type of message it will call a different handler.
- **handle_wave**: Upon receiving a wave message, one of the following cases will occur:
  - The received wave_id is equal to the node id: discard the message and increment the counter.
  - The received wave_id is equal to the node wave_id: discard the message.
  - The received wave_weight is grater than the node wave_weight: discard the message.
  - The received wave_weight is lower than the node wave_weight: adopt the received wave_id and wave_weight, then broadcast a wave message with the new wave information.

  When the counter of one node will reach the number of nodes in the network minus one, this means that it has been elected and it will now call the take_job method. It is safe that it will never happen that more than one node will be elected in the same wave algorithm.
- **initialize**: This method sends to the server an initialize message that starts our three-way handshake. After sending it, it waits for the response from the server, and once it receives it, it will call the handle_initialize_response.
- **send_message**: This is the method used by the node when it needs to send a message. We simply pass the target address, the type of message and the payload.
- **shutdown**: This method simply sets the shutdown flag to True, initializing the shutdown sequence of the node.
- **start**: This method is called when the node is initialized. It will start the initialize thread and the greetings thread after that. It also binds the socket and starts listening for messages.
- **take_job**: This method gets called when a node gets elected in the wave algorithm. It will send a take job message to the server.
- **wave**: The wave algorithm starts initializing the counter at 0, initializing the wave_id with the node_id, and initializing the wave_weight with the result of shortest_deviation passing the new Job object. After that it sends a message with its wave_id and wave_weight to all other nodes.

5.8 Server

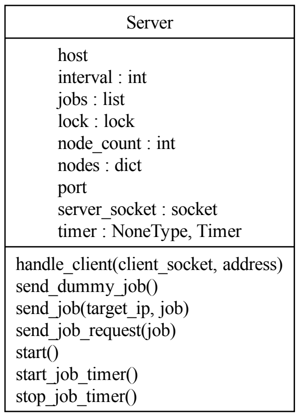

Figure 8: Attributes and methods of the Server class

The server implementation is pretty simple, we just used it for debugging purposes.

- **host**: The server host.
- **interval**: The timeout limit.
- **jobs**: The server uses this attribute to keep track of the jobs currently being executed by the nodes.
- **lock**: This object is used to allow threading in a safe way, avoiding race conditions to certain memory areas.
- **node_count**: The number of nodes in the network.
- **nodes**: The dictionary containing the nodes_id and their relative ips.
- **port**: The server port.
- **server_socket**: The socket that gets initialized in the constructor with the passed host and port.
- **timer**: The timer object needed for the timeout waiting a response.

The server implements the following methods that mainly regards the handling of the node initialization and the handling of the job requests.

- **handle_client**: This method handle the messages from the client:

  - If the message is an initialization message, the server checks if the ip of the node is already in its nodes list, if not it increases the node_count, adds the node ip and id to its list, and then it sends a response containing the assigned id and the nodes list. If the node is already in the nodes list, the server will respond with an error.
  - If the message is a take_job request, it will respond with an ack.

- **send_dummy_job**: This method is for debugging purposes only. It creates a fake Job object with random coordinates and calls the send_job method.
- **send_job**: The server sends the Job object in a job type message in broadcast to the network.
- **send_job_request**: The server calls the send_job method in a for loop for every node in the nodes list.
- **start**: The initialization method for the server, that puts it listening for messages from the clients.
- **start_job_timer**: The method to start the timer.
- **stop_job_timer**: The method to stop the timer.

5.9 Github

The project files can be found at this GitHub link.

6 Centralized vs decentralized approach

Aside from our implementation, we should make a few considerations about the differences between the centralized and decentralized approach.

What we implemented is sorta of an hybrid approach, with a first computation of the vehicle paths, using the ant colony algorithm on the server, and then the computation of a possible deviation on the vehicle themselves. Aside from the academic point of view, in a real life scenario we should investigate the advantages and downsides of a pure centralized approach and a pure decentralized approach.

The two main criteria that might come to mind are fault tolerance and real time computations.

Of course having a fully distributed and decentralized system would mean having no central server and that the network created by the vehicles would be the full network: we would have to implement a mechanism to elect a node as leader in our network. In this case probably we would use a consensus algorithm like raft, and then we would let every node make the pathfinding computations locally, and then inform the leader, that would register the events and then inform the nodes of the new log. The advantages are pretty clear: we would have a system that would be extremely resilient to faults: if a node would fail, the leader node would discover it and inform the others, letting them take the jobs of the other node, computing new routes. If it would be the leader node to fail, it would just take another round of election and we would have a new leader node, and then do the same, splitting the jobs with the other nodes. The real problem would be related to the computational weight of the pathfinding operations: on a server machine these would take seconds at most (without considering that if we would have also to query the geocoder and recompute the weight matrix the problem would scale exponentially), but on a smaller device it would probably take a lot than that, making us consider a new plethora of problems like computing with respect to the future position of the vehicle, estimating the computation times, wasting computation resources on a mobile machine, wasting also energy that could be used for something else.

Instead, in a fully centralized approach we would have a central server doing all the pathfinding computations. When a new job request would arrive it would compute all the deviations for every active vehicle and then notify the chosen vehicle of the new stop. The vehicle could will be able to refuse (low battery or no storing space), and then the server would still be able to notify the next lowest weight deviation vehicle. The greatest advantage of this approach is obvious: the heavy computations would run exclusively on a powerful server machine, probably taking still less time than parallelizing the N computations on N extremely less powerful machines. The compromise would be losing all of the fault tolerance from the decentralized approach, having one critical fault point on the server. This of course can be bypassed with various strategies (back-up machines for example), making the overall system more resilient.

In general probably the best approach in a real life implementation would still be the fully centralized one: we would have to lose a bit of fault tolerance in order to have a more robust (computationally-wise) system, that would also severely simplify the networking approach, eliminating all the need for the nodes to communicate between themselves.

7 The ant colony algorithm

The main purpose of this project is to test the performance of the **ant colony algorithm** in the already presented scenario of a fleet of vehicles that must divide their load of work in a functional way. This means that we want that the algorithm is able to minimize the weight we are considering for this optimization problem, and that eventually the split weight can have a variance as low as possible.

As we already explained, the ant colony algorithm draws inspiration from the foraging behavior of ants. In nature, ants communicate through pheromones, leaving a trail as they find food. This trail attracts other ants, reinforcing the path to the food source. The algorithm translates this concept into an optimization algorithm where artificial ants navigate a solution space, leaving digital "pheromones" on the paths they explore.

So basically we are deploying artificial ants searching for the optimal solution to a problem. Initially, the artificial ants are scattered randomly across the solution space. Pheromone levels are initialized uniformly on all paths. Each ant then starts to construct a solution by iteratively choosing the next component based on a combination of pheromone levels and heuristic information.

The pheromone levels represent the desirability of a particular path, analogous to the attractiveness of a trail in the natural world. Heuristic information guides the ants toward promising regions in the solution space, akin to ants being drawn to areas where they are likely to find food.

As the ants construct solutions, they deposit pheromones on the paths they traverse. The better the solution, the more intense the pheromone deposition. This imitates the reinforcement of successful paths in nature. Over time, paths leading to superior solutions accumulate more pheromones, making them more appealing to subsequent ants.

To prevent the algorithm from converging prematurely, pheromones are subjected to evaporation. This simulates the natural decay of pheromone trails and ensures adaptability in the exploration process. The balance between exploration and exploitation is crucial in the ant colony algorithm. Initially, there's a phase of exploration where ants discover diverse paths. As the algorithm progresses, the influence of pheromones causes a shift towards exploitation, where ants favor paths with higher pheromone levels.

The convergence to a solution is a dynamic interplay between the ants' exploration and exploitation tendencies. The collective intelligence emerges as ants communicate indirectly through the pheromone-laden paths. Paths that consistently lead to better solutions intensify in attractiveness, guiding subsequent ants to explore similar routes.

The implementation that we will show in this section is not the same of the previous section, but a modified version used for the case study presented in the following section.

7.1 Prelude

```python
import osmnx as ox
import numpy as np
import geocoder
import pandas as pd
import copy
import datetime
import matplotlib.pyplot as plt

import itertools
```

These are all the libraries we are importing for the experiment:

- **osmnx**: as we said before this is a library that uses Open Street Map data and manages it using the NetworkX library.

- **numpy**: python library that allows to manage scientific data and scientific computing with ease.
- **geocoder**: geocoding library for python, it allows us to get coordinates from addresses.
- **pandas**: python library used to manage dataframes.
- **copy**: this python library is extremely useful with its deepcopy function to set a variable to a specific value of another object without passing the reference of said object.
- **datetime**: the standard python library to get times and manage time variables.
- **matplotlib**: python library for plotting data and manage plots.
- **itertools**: this library has many functions used to create iterable items, we use it mainly for the cartesian product of the ant colony algorithm's parameters for the grid search of the optimal parameters.

```
jobs_dataframe = pd.read_excel('complete_original_task_list.
    xlsx')
```

We use pandas function to pass the excel data to a dataframe, so we can read it and feed it to the algorithm after some management. We also had to correct some typos making researches on the web, since some of these addresses did not gave us a result on the geocoding queries, but we will see it in detail in the study case section.

```
G = ox.graph_from_point((45.464664, 9.188540), dist=20000,
    network_type='drive', simplify=False)

G = ox.distance.add_edge_lengths(G)
G = ox.speed.add_edge_speeds(G)
G = ox.speed.add_edge_travel_times(G)
```

These lines of code will give us the graph with all the information we need for our ant colony algorithm:

- **graph_from_point** is a function that returns us a graph starting from a certain set of coordinates for a certain km range. We also specify the type of network we are interested in (if we want just drive roads, pedestrian roads, both etc) and if we want to simplify the edges removing some of the less important ones.
- **add_edge_lengths**: adds the length of the road stat to the edges.
- **add_edge_speeds**: adds the speed limit stat to the edges.
- **add_edge_travel_times**: adds the mean time travel stat to the edges.

7.2 Ant Colony Algorithm

We will now proceed to show all the code related to the ant colony algorithm.

```python
def get_jobs(df = jobs_dataframe, graph = G, starting_point = 272714832):
    job_list = []
    ref_map = {'Via Olona, 2, 20123 Milano MI, Italy': {'id': 0, 'has_pick_up': True}}
    map_value = 1

    job_list.append({'id': 0, 'pick_up': True, 'node': starting_point})

    for i, row in df.iterrows():
        job = {}
        if row['Reference'] not in ref_map:
            ref_map[row['Reference']] = {}
            ref_map[row['Reference']]['id'] = map_value
            ref_map[row['Reference']]['has_pick_up'] = False
            map_value += 1
        job['id'] = ref_map[row['Reference']]['id']
        if row['Category']=='drop_off':
            job['pick_up'] = False
        else:
            job['pick_up'] = True
            ref_map[row['Reference']]['has_pick_up'] = True
        g = geocoder.osm(row['Address']).json
        job['node'] = ox.distance.nearest_nodes(graph, g['lng'], g['lat'])
        job_list.append(job)

    for k in ref_map:
        if ref_map[k]['has_pick_up']:
            continue
        else:
            job_list.append({'id': ref_map[k]['id'], 'pick_up': True, 'node': 272714832})

    return job_list, ref_map
```

This first function returns us a list of dictionary items starting from the dataframe we obtained earlier. Basically the way it works is, fixed a starting point, that will be the node 0 of our matrix, the function maps the dataframe reference values and translates them in int values, then queries the geocoder to obtain the coordinates of the address, and uses those coordinates to get the corresponding nodes on the graph. In the end, it checks if there are no ”orphan” delivery orders, and in that case we will assume that the shipment relative to that delivery has been picked up before and is waiting for delivery in the warehouse, so we will add a dummy job placed in the starting point/warehouse to fill in the gap of the missing pick up.

```python
def get_weight_matrix(G, job_list, weight_string='travel_time')
    :
    l = len(job_list) #The number of jobs
    paths_dict = {}
    weight_matrix = np.zeros((l, l)) #Initialize the matrix
    for i in range(l): #For every job (rows)
        paths_dict[i] = {}
        for j in range(l): #For every job (columns)
            if i != j: #row not equal column, no self loops in
                the adjacency matrix
                el = ox.routing.shortest_path(G, job_list[i]['
                    node'], job_list[j]['node'], weight=
                    weight_string, cpus=None) #Open Street Map
                    uses Dijkstra's algorithm in order to find
                    the path between two nodes that minimize the
                     chosen weight
                paths_dict[i][j] = el
                for u, v in zip(el, el[1:]): #Use the zip
                    function to create an iterable object for
                    every step of the path
                    weight_matrix[i][j] += G.edges[u,v,0][
                        weight_string] #For every step of the
                        path add the weight to the matrix
    return weight_matrix, paths_dict #Return the weight matrix
```

This is without a dubt the most computastional heavy function of the whole project, but it is necessary to obtain the weights of the travel between nodes. It is of course heavy because we will have to run the Dijkstra's algorithm (as we can read in the comments of the code) N^2 times, for N equal to the number of nodes, for a non-negligible amount of edges. Luckily we can parallelize it (cpus=None).

From this point on all the functions we will see are used directly in the ant colony algorithm, describing each fundamental part of it. Since all the functions are heavily commented, we will be very brief on the comments.

```python
def get_allowed_nodes(job_list, path, visited_nodes):
    allowed_nodes = [] #Initialize the allowed nodes list
    for i in range(len(job_list)): #For every possible job
        if i in visited_nodes: #If it has already been visited
            continue #Skip iteration
        if job_list[i]['pick_up']: #If it is a pick up
            allowed_nodes.append(i) #Append the node to the
```

```
allowed nodes
        else: #Otherwise, if it is not a pick up
            for node in path: #For every node already visited
                if job_list[node]['id'] == job_list[i]['id']: #If the node already visited the relative withdrawal
                    allowed_nodes.append(i) #Append the node to the allowed nodes
    return allowed_nodes #Return the allowed nodes list
```

This function simply returns a list of all the nodes that an ant, given its path up to that point and given the other's concurrent ants path, can travel to.

```
def get_next_node(weight_matrix, job_list, pheromone, path, visited_nodes, epsilon, alpha, beta):
    current_node = path[-1] #The node we are currently in is the last visited
    allowed_nodes = get_allowed_nodes(job_list, path, visited_nodes) #Get the list of allowed nodes
    if len(allowed_nodes) == 0: #If the allowed nodes list is empty
        return False #Return false
    pheromone_values = pheromone[current_node, allowed_nodes] #Extract the pheromone values of the outgoing edges of this node
    heuristic_values = 1 / (weight_matrix[current_node, allowed_nodes] + epsilon) #Compute the heuristic values as the inverse of the weights
    #heuristic_values *= priorities #If there were some priorities to slide in the equation, they would need to be computed here
    probability_values = (pheromone_values ** alpha) * (heuristic_values ** beta) #Initialize the probability values as the pheromones at power alpha times heuristic values at power beta
    probability_values /= np.sum(probability_values) #Normalize the probability values
    next_node = np.random.choice(allowed_nodes, p=probability_values) #Based on probability values choose the next node
    return next_node #Return the next node
```

This function, given the allowed nodes an ant can travel to, does a random choice of the next node in its path. The choice, although being random, it has a weighted probability, given by the following formula:

$$p_n = pher^{\alpha} h^{\beta} \tag{1}$$

where p_n is the non-normalized probability, $pher$ is the pheromone value, h is the inverse of the weight and α and β are their respective exponents.

```python
def compute_path_weight(weight_matrix, ant_path):
    tot_weight = 0 #Initialize weight
    n_weights = {}
    for n in ant_path: #For every agent's path
        weight = 0
        for i in range(len(ant_path[n]) - 1): #For every edge in agent's path
            weight += weight_matrix[ant_path[n][i], ant_path[n][i + 1]] #Add the edge weight
        n_weights[n] = weight
        tot_weight += weight
    return tot_weight, n_weights #Return the total sum of paths' weight
```

After all the concurrent ants terminate their travel we need to compute their path weight. We made sure to have the algorithm returns both the total weight of the concurrent ants and their individual scores.

```python
def update_pheromone(pheromone, ant_paths, ant_weights, ant_n_weights, rho, use_sum_weight=True):
    new_pheromone = pheromone #Initialize new pheromone levels
    new_pheromone *= (1 - rho) #Apply decay rate
    for i in range(len(ant_paths)): #For each ant
        for n in ant_paths[i]: #For each agent in ant_path
            for u, v in zip(ant_paths[i][n], ant_paths[i][n][1:]): #Use zip in order to create an iterable of the path
                if use_sum_weight:
                    new_pheromone[u, v] += 1 / ant_weights[i] #Add pheromone values for each edge traveled inverse to the total weight of the path traveled
                else:
                    new_pheromone[u, v] += 1 / ant_n_weights[i][n] #If we would prefer to update the pheromones using the individual weights we would need to do it this way
    return new_pheromone
```

The update pheromone function is pretty straightforward too. We apply the decay we defined $(1 - \rho)$, then we just add new values to the existing ones based on the inverse of the weights of the road traveled. We implemented two different methods to update the pheromones, based on the idea that updating the pheromone with the sum of the weights of the concurrent ants might give us a different result then updating them with the singular weights of every ant. We expect the algorithm to converge more efficiently towards a global minimum if we update the pheromones with the singular weight values of each ant.

```python
def ant_colony(G, job_list, weight_matrix, n_agents=4, n_ants
    =10, n_iterations=50, alpha=1, beta=2, rho=0.5,
    pher_sum_weight=True):
    '''
    G = the osm graph
    n = number of agents for the algorithm
    n_ants = number of ants running for iteration
    n_iteration = number of iterations to let the algorithm
        converge
    alpha = exponent of the pheromone (higher values give
        pheromone more weight)
    beta = exponent of the cost measure (higher values give
        more weight to the cost)
    rho = pheromone decay rate ([0,1] where 0 lefts the
        pheromone values unchanged)
    '''
    epsilon = 1e-10 #This value is added to the denominator in
        the divisions to avoid divisions by zero
    #weight_matrix = get_weight_matrix(G, job_list) #The weight
         matrix obtained from the function
    #Since the computation for the weight matrix is very heavy
        we prefer, for debugging purposes, to compute it outside
         the main algorithm use the one passed as parameter
    pheromone = np.ones_like(weight_matrix) * 0.1 #Initialize
        the pheromones matrix
    # Define the main algorithm loop
    results = {"path": [], "weight": [], "n_weights": []} #
        Initialize results dictionary
    for iteration in range(n_iterations): #For every iteration
        ant_paths = [] #Initialize the list of paths
        ant_weights = [] #Initialize the list of weights
        ant_n_weights = [] #Initialize the list of weights for
            every agent
        for ant in range(n_ants): #For every ant
            ant_path = {} #Initialize path
            for n in range(n_agents): #For every agent
                ant_path[n]=[0] #Start the path from the
                    starting node
            n = -1 #Initialize the agents iterator
            visited_nodes = [0] #Initialize the visited nodes
                list
            end_path = [False] * n_agents #List to check if an
```

```
                agent has already ended its path
            while len(visited_nodes) < len(job_list): #Until
                all nodes are visited
                n = (n + 1) % n_agents #Changing agent at each
                    iteration
                if end_path[n]: #If the ant has already ended
                    its path
                    continue #Skip the loop iteration
                next_node = get_next_node(weight_matrix,
                    job_list, pheromone, ant_path[n],
                    visited_nodes, epsilon, alpha, beta) #Select
                     the next node
                if next_node == False: #If the next_node
                    function returns False
                    end_path[n] = True #Set the end_path for
                        this agent to True
                    ant_path[n].append(0) #End the path going
                        back to the starting point
                    continue #Skip the loop iteration
                ant_path[n].append(next_node) #Append the node
                    to the path of the agent
                visited_nodes.append(next_node) #Update the
                    list of visited nodes
            for e in range(len(end_path)): #For every agent
                if end_path[e] == False: #Check if the path
                    ended at the warehouse
                    ant_path[e].append(0) #If not, append the
                        end point of the path
            ant_paths.append(ant_path) #Save the path
            weight, n_weights = compute_path_weight(
                weight_matrix, ant_path)
            ant_weights.append(weight) #Save the weight of the
                path
            ant_n_weights.append(n_weights) #Save the
                individual weights of the agents
        pheromone = update_pheromone(pheromone, ant_paths,
            ant_weights, ant_n_weights, rho, use_sum_weight=
            pher_sum_weight) #Update pheromones on the edges of
            the graph
        best_ant_index = np.argmin(ant_weights) #Find the best
            ants
        best_ant_path = ant_paths[best_ant_index] #Find the
            best path
        best_ant_weight = ant_weights[best_ant_index] #Find the
             weight of the best ant
        best_ant_n_weights = ant_n_weights[best_ant_index] #
```

```
            Finde the individual weights of the best ant
        results["path"].append(best_ant_path) #Append best ant
            path to results
        results["weight"].append(best_ant_weight) #Append best
            ant weight to results
        results["n_weights"].append(best_ant_n_weights) #Append
             best individual weight to results
    return results
```

The main function of the algorithm basically just puts everything together: for every iteration, for every ant we travel through our subgraph until we traveled it all, than we compute our weights, update the pheromones and find the best ant of the iteration, saving its result. The result this function returns is a list of the best ant for each iteration. Normally from this we simply get the best solution based on our criteria.

7.3 Experiment

All of the following code will present also the instructions to calculate its computing time.

First of all, since, as we said, the computation of the weight matrix is very computationally heavy, we do it before the ant colony algorithm, instead of doing this at run time.

```
start = datetime.datetime.now()

weight_matrix, shortest_paths = get_weight_matrix(G, job_list)

end = datetime.datetime.now()
weight_matrix_time = end-start
```

After that we can begin our grid search, searching for the optimal set of parameters for our problem:

```
n_ants_list=[10, 20, 30, 40, 50]
n_iterations_list=[50, 100, 150, 200, 250]
alpha_list=[0.0, 0.2, 0.5, 1.0, 2.0, 5.0]
beta_list=[0.0, 0.2, 0.5, 1.0, 2.0, 5.0]
rho_list=[0.0, 0.001, 0.01, 0.1, 0.5, 0.9]
pher_sum_weight_list=[False, True]

grid_search = itertools.product(n_ants_list, n_iterations_list,
     alpha_list, beta_list, rho_list, pher_sum_weight_list)

grid_search_num = len(list(itertools.product(n_ants_list,
    n_iterations_list, alpha_list, beta_list, rho_list,
    pher_sum_weight_list)))
```

```

ant_colony_time = {}

results = {}

grid_iteration = 1

for params in grid_search:

    n_ants, n_iterations, alpha, beta, rho, pher_sum_weight = params

    print(f"{grid_iteration}/{grid_search_num}: n_ants = {n_ants}, n_iterations = {n_iterations}, alpha = {alpha}, beta = {beta}, rho = {rho}, pher_sum_weight = {pher_sum_weight}")

    start = datetime.datetime.now()

    res = ant_colony(G, job_list, weight_matrix, 4, n_ants, n_iterations, alpha, beta, rho, pher_sum_weight)

    end = datetime.datetime.now()

    ant_colony_time[params] = end-start
    results[params] = res

    grid_iteration +=1
```

Basically we initialize the algorithm parameters list, with all the values we want to try, then we compute a cartesian product of those parameters (itertools.product), and for every combination of those parameters we run the ant colony algorithm, searching for the set of parameters that will give us the optimal solution.

8 Study case

Our study case involves a dataset from a real business located in Milan, Italy.

Id	Reference	Address	Start travel	End travel	Start task	End task	Duration	Category	Start large	End large	Complete range
0	Frida's Corso Genova02_2023-11-21	Via Ettore Ponti, 58, Milano, MI, Italia	0	0	2023-11-21 15:00:00	2023-11-21 17:00:00	0.003472	drop_off	2023-11-21 15:00:00	2023-11-21 17:00:00	0.083333
1	Frida's Corso Genova02_2023-11-21	Corso Genova, 6, 20123 Milano MI, Italy	0	0	2023-11-21 12:15:00	2023-11-21 15:00:00	0.003472	pick_up	2023-11-21 12:15:00	2023-11-21 15:00:00	0.114583
2	Fiori Fumagalli03_2023-11-21	Foro Buonaparte, 22, Milano, MI, Italia	0	0	2023-11-21 14:15:00	2023-11-21 17:00:00	0.003472	drop_off	2023-11-21 14:15:00	2023-11-21 17:00:00	0.114583
3	Cafezal02_2023-11-21	Via Larga, 13, Milano, MI, Italia	0	0	2023-11-21 12:15:00	2023-11-21 17:00:00	0.003472	drop_off	2023-11-21 12:15:00	2023-11-21 17:00:00	0.197917
4	Clori03_2023-11-21	Via Larga, 19, Milan, Metropolitan City of Milan, Italy	0	0	2023-11-21 12:00:00	2023-11-21 17:00:00	0.003472	drop_off	2023-11-21 12:00:00	2023-11-21 17:00:00	0.208333
5	Fiori Fumagalli03_2023-11-21	Via Messina, 47, Milano, MI, Italia	0	0	2023-11-21 17:30:00	2023-11-21 18:00:00	0.003472	drop_off	2023-11-21 17:30:00	2023-11-21 18:00:00	0.020833
6	Fiori Fumagalli03_2023-11-21	Piazza Guglielmo Oberdan, 2, Milano, MI, Italia	0	0	2023-11-21 14:00:00	2023-11-21 16:15:00	0.003472	pick_up	2023-11-21 14:00:00	2023-11-21 16:15:00	0.093750
7	La teiera Eclettica01_2023-11-21	Via Piero della Francesca, 74, Milano, MI, Italia	0	0	2023-11-21 14:00:00	2023-11-21 17:00:00	0.003472	drop_off	2023-11-21 14:00:00	2023-11-21 17:00:00	0.125000
8	La teiera Eclettica01_2023-11-21	La Teiera Eclettica, Via Melzo, 30, Milano, MI, Italia	0	0	2023-11-21 11:00:00	2023-11-21 16:00:00	0.003472	pick_up	2023-11-21 11:00:00	2023-11-21 16:00:00	0.208333
9	Dolcemente01_2023-11-20	Via Mario Pichi, 8, Milano, MI, Italia	0	0	2023-11-21 11:00:00	2023-11-21 17:00:00	0.003472	drop_off	2023-11-21 11:00:00	2023-11-21 17:00:00	0.250000
10	Florarredo01_2023-11-21	Via Gaetano de Castillia, 6A, Milano, MI, Italia	0	0	2023-11-21 11:00:00	2023-11-21 17:00:00	0.003472	drop_off	2023-11-21 11:00:00	2023-11-21 17:00:00	0.250000
11	Florarredo01_2023-11-21	Piazza della Repubblica, 32, 20124 Milano MI, Italy	0	0	2023-11-21 10:45:00	2023-11-21 16:00:00	0.003472	pick_up	2023-11-21 10:45:00	2023-11-21 16:00:00	0.218750
12	Pasticceria Grecchi Luigi01_2023-11-21	Via Felice Casati, 45, 20124 Milano MI, Italy	0	0	2023-11-21 17:00:00	2023-11-21 18:30:00	0.003472	drop_off	2023-11-21 17:00:00	2023-11-21 18:30:00	0.062500
13	Pasticceria Grecchi Luigi01_2023-11-21	Via Piero della Francesca, 7, 20154 Milano MI, Italy	0	0	2023-11-21 16:30:00	2023-11-21 17:00:00	0.003472	pick_up	2023-11-21 16:30:00	2023-11-21 17:00:00	0.020833
14	Au Nome de la Rose01_2023-11-21	Via Moneta, 1, Milano, MI, Italia	0	0	2023-11-21 12:15:00	2023-11-21 16:30:00	0.003472	drop_off	2023-11-21 12:15:00	2023-11-21 16:30:00	0.177083
15	Au Nome de la Rose01_2023-11-21	Piazza Riccardo Wagner, 20145 Milano MI, Italy	0	0	2023-11-21 12:00:00	2023-11-21 15:30:00	0.003472	pick_up	2023-11-21 12:00:00	2023-11-21 15:30:00	0.145833
16	Cosaporto_603825	Via Giovanni Cimabue, 41, Milano MI, Italia	0	0	2023-11-21 17:00:00	2023-11-21 19:00:00	0.003472	drop_off	2023-11-21 17:00:00	2023-11-21 19:00:00	0.083333
17	Cosaporto_603825	Via Felice Casati, 27, 20124 Milano, Milano MI, Italia	0	0	2023-11-21 17:00:00	2023-11-21 18:00:00	0.003472	pick_up	2023-11-21 17:00:00	2023-11-21 18:00:00	0.041667
18	Pasticceria Spezia01_2023-11-20	Via Amedei, 8, 20123 Milano MI, Italy	0	0	2023-11-21 17:00:00	2023-11-21 18:30:00	0.003472	drop_off	2023-11-21 17:00:00	2023-11-21 18:30:00	0.062500
19	Pasticceria Spezia01_2023-11-20	Via la Spezia, 26, Milano, MI, Italia	0	0	2023-11-21 16:30:00	2023-11-21 17:30:00	0.003472	pick_up	2023-11-21 16:30:00	2023-11-21 17:30:00	0.041667
20	Bibilab02_2023-11-20	Via Seprio, 2, Milano, MI, Italia	0	0	2023-11-21 15:15:00	2023-11-21 17:00:00	0.003472	drop_off	2023-11-21 15:15:00	2023-11-21 17:00:00	0.072917
21	Bibilab02_2023-11-20	Via Caminadella, 18, 20123 Milano MI, Italy	0	0	2023-11-21 15:15:00	2023-11-21 17:00:00	0.003472	drop_off	2023-11-21 15:15:00	2023-11-21 17:00:00	0.072917
22	Bibilab02_2023-11-20	Via Mario Pannunzio, 4, 20156 Milano MI, Italy	0	0	2023-11-21 15:00:00	2023-11-21 16:00:00	0.003472	pick_up	2023-11-21 15:00:00	2023-11-21 16:00:00	0.041667
23	Dolcemente01_2023-11-20	Via Privata Giovanni Ventura, 5, Milano, MI, Italia	0	0	2023-11-21 09:15:00	2023-11-21 17:00:00	0.003472	drop_off	2023-11-21 09:15:00	2023-11-21 17:00:00	0.322917
24	Walter Calzature01_2023-11-17	via Val di Fiemme, 25, 20128 Milano MI, Italy	0	0	2023-11-21 08:00:00	2023-11-21 20:00:00	0.003472	drop_off	2023-11-21 08:00:00	2023-11-21 20:00:00	0.500000
25	1952_452_92	Via Santa Sofia, 27, 20122 Milano MI, Italy	0	0	2023-11-21 05:00:00	2023-11-21 20:00:00	0.003472	drop_off	2023-11-21 05:00:00	2023-11-21 20:00:00	0.625000
26	Cosaporto_603053	Via Morozzo della Rocca, 8, Milano, MI, Italia	0	0	2023-11-21 14:00:00	2023-11-21 16:00:00	0.003472	drop_off	2023-11-21 14:00:00	2023-11-21 16:00:00	0.083333
27	Cosaporto_603053	Corso Lodi, 15, 20135 Milano MI, Italy	0	0	2023-11-21 14:00:00	2023-11-21 15:00:00	0.003472	pick_up	2023-11-21 14:00:00	2023-11-21 15:00:00	0.041667
28	PastaFra 01_2023-11-20	Via Giuseppe Tartini, 14, 20158 Milano MI, Italy	0	0	2023-11-21 15:00:00	2023-11-21 18:00:00	0.003472	drop_off	2023-11-21 15:00:00	2023-11-21 18:00:00	0.125000
29	PastaFra 01_2023-11-20	Via Enrico Annibale Butti, 18, 20158 Milano MI, Italia	0	0	2023-11-21 13:45:00	2023-11-21 17:00:00	0.003472	drop_off	2023-11-21 13:45:00	2023-11-21 17:00:00	0.135417
30	PastaFra 01_2023-11-20	Via Filippo Argelati, 7, 20143 Milano MI, Italy	0	0	2023-11-21 13:00:00	2023-11-21 14:30:00	0.003472	pick_up	2023-11-21 13:00:00	2023-11-21 14:30:00	0.062500
31	Walter Calzature03_2023-11-20	Via Forze Armate, 50, Milano, MI, Italia	0	0	2023-11-21 11:30:00	2023-11-21 20:00:00	0.003472	drop_off	2023-11-21 11:30:00	2023-11-21 20:00:00	0.354167
32	Walter Calzature03_2023-11-20	Via Monte Rosa, 17, Milano, MI, Italia	0	0	2023-11-21 11:30:00	2023-11-21 20:00:00	0.003472	drop_off	2023-11-21 11:30:00	2023-11-21 20:00:00	0.354167
33	1950_450_92	Giuseppe, Via Giuseppe Ripamonti, 27, 20135 Milano MI, Italy	0	0	2023-11-21 05:00:00	2023-11-21 20:00:00	0.003472	drop_off	2023-11-21 05:00:00	2023-11-21 20:00:00	0.625000
34	Alveare Basiglio01_2023-11-20	Via Pietro Nenni, 40, 20085 Locate di Triulzi MI, Italia	0	0	2023-11-21 17:45:00	2023-11-21 20:00:00	0.003472	drop_off	2023-11-21 17:45:00	2023-11-21 20:00:00	0.093750
35	Alveare Basiglio01_2023-11-20	Vicolo Torretta, 18, 27010 Siziano PV, Italia	0	0	2023-11-21 17:45:00	2023-11-21 20:00:00	0.003472	drop_off	2023-11-21 17:45:00	2023-11-21 20:00:00	0.093750
36	Alveare Basiglio01_2023-11-20	Via dei Pini, 2l, 20072 Pieve Emanuele MI, Italy	0	0	2023-11-21 17:45:00	2023-11-21 20:00:00	0.003472	drop_off	2023-11-21 17:45:00	2023-11-21 20:00:00	0.093750
37	Alveare Basiglio01_2023-11-20	Via dei Pini, 4, 20090 Pieve Emanuele MI, Italia	0	0	2023-11-21 17:45:00	2023-11-21 20:00:00	0.003472	drop_off	2023-11-21 17:45:00	2023-11-21 20:00:00	0.093750
38	Alveare Basiglio01_2023-11-20	Via Borgo del Majno, 1a, 20080 Basiglio, MI, Italy	0	0	2023-11-21 17:30:00	2023-11-21 19:00:00	0.003472	pick_up	2023-11-21 17:30:00	2023-11-21 19:00:00	0.062500
39	Alveare del Naviglio01_2023-11-20	Via Nicola Romeo, 5, 20142 Milano MI, Italia	0	0	2023-11-21 17:15:00	2023-11-21 20:00:00	0.003472	drop_off	2023-11-21 17:15:00	2023-11-21 20:00:00	0.114583
40	Alveare del Naviglio01_2023-11-20	Via Don Rodrigo, 3, 20142 Milano MI, Italy	0	0	2023-11-21 17:00:00	2023-11-21 18:30:00	0.003472	pick_up	2023-11-21 17:00:00	2023-11-21 18:30:00	0.062500
41	Alveare Terzo Tempo01_2023-11-20	Via Giuseppe Frua, 26, Milano, MI, Italia	0	0	2023-11-21 17:15:00	2023-11-21 20:00:00	0.003472	drop_off	2023-11-21 17:15:00	2023-11-21 20:00:00	0.114583

Table 1: The dataset

As we can see the dataset is composed by 42 elements, with the following fields:

- **Id**: a unique identifier for the task.
- **Reference**: presumably the customer that required the job.
- **Address**: the location of the task.
- **Start travel**: this should be related to the path given by the elaboration of the original system, so we will ignore it.
- **End travel**: this should be related to the path given by the elaboration of the original system, so we will ignore it.
- **Start task**: this should be part of the result of the elaboration of the original system, so we will ignore it.
- **End task**: this should be part of the result of the elaboration of the original system, so we will ignore it.
- **Duration**: this is not an information useful to our implementation.
- **Category**: the type of task: withdrawal(pick_up) or delivery (drop_off).
- **Start large**: the start of the time period where the task can be performed.
- **End large**: the end of the time period where the task can be performed.
- **Complete range**: this is not an information useful to our implementation.

Analyzing the dataset we can rapidly detect that there are some "orphan" drop_off tasks. This means that there is not a pick_up task with the same reference. We assume that this means that the object of the drop_off is already in the warehouse, and so we add some dummy pick_up tasks located in the warehouse. This is the list of the Ids of the "orphan" tasks:

- 3
- 4
- 9
- 23
- 24
- 25
- 31
- 32
- 33
- 41

Also, during the geocoding of some addresses, the geocoder library returned some None values, so we had to rename some of the addresses to correct some typos or some problems related to the geocoder:

- 19: "Via la Spezia" does not exist in Milan, it's called "Via Spezia".
- 31: "Via Forze Armate" does not exist in Milan, it's called "Via delle Forze Armate".
- 33: There was a typo: "Giuseppe, Via Giuseppe Ripamonti" is actually "Via Giuseppe Ripamonti".
- 38: There actually is a typo in the Open Street Map database: "Via Borgo del Majno" is wrongly written as "Via Borgo del Maino" on Open Street Map, so we had to rename it in order to have the geocoder work correctly.

9 Time management

We also implemented a modified version of the ant colony algorithm that includes mechanisms of time management for the ants, bringing us closer to a realistic situation, but in some way we are drifting further away from the metaheuristic algorithm and going towards more of a discrete times simulation.

In our new implementation we need to modify 4 things maily:

1. The list of tasks that we extract from the excel file must include the time periods for the task (Start large and End large in the excel file).
2. We can't allow ants to travel to nodes where the task would not be executable once they arrive.
3. We must change our pheromone updating criteria to meet the new requirements.

4. We must insert in the main loop some synchronization and idling mechanisms.

9.1 Job list

```
1  def get_jobs_t(df = jobs_dataframe, graph = G, starting_point = 272714832):
2      ...
3          g = geocoder.osm(row['Address']).json
4          job['node'] = ox.distance.nearest_nodes(graph, g['lng'], g['lat'])
5          job['time'] = (row['Start large'].to_pydatetime(), row['End large'].to_pydatetime())
6          job_list.append(job)
7      ...
```

This was probably the most straightforward modification of them all. We simply added the code that is displayed above on the 5th row to the already existing get_jobs function. This way our dictionary will have one more field called 'time' with a tuple (start task period, end task period).

9.2 Allowed nodes

```
def get_next_node_t(weight_matrix, job_list, pheromone, path, unvisited_nodes, epsilon, alpha, beta, time, wait_time):
    current_node = path[-1] #The node we are currently in is the last visited
    current_time = time[-1][1] + datetime.timedelta(seconds=wait_time)

    allowed_nodes = [] #Initialize the allowed nodes list
    for i in unvisited_nodes: #For every possible job
        if 'time' in job_list[i]: #If the task has a time constraint
            if current_time < (job_list[i]['time'][0] + datetime.timedelta(seconds=weight_matrix[current_node, i])) or current_time > (job_list[i]['time'][1] + datetime.timedelta(seconds=weight_matrix[current_node, i])): #If we are out of the time period for the task
                continue #Skip the iteration
        if job_list[i]['pick_up']: #If it is a pick up
            allowed_nodes.append(i) #Append the node to the allowed nodes
    ...
```

Here the modifications were a bit more in depth. First of all we do not use the get_allowed_nodes function in this version of the get_next_node function. This is because in order to find out the nodes that we can travel to, we need the weight matrix, and since we already pass it to the get_next_node function, we can simply make our computations here instead of passing it to another function. Then what we simply did is add one more condition to the check of the visitable nodes, adding the condition that they must be available for the task at the time at which the agent will be there.

Two observations need to be made:

1. As we can see, in this new version we don't have the visited_nodes list that will increment, but we use an unvisited_nodes list. This was done in order to simplify the allowed nodes check loop, and also for an easier main loop check in the main algorithm (we will see it better in the next sections).

2. The check for the time constraints is actually very simple, and the reason for this design choice came after trying many more convoluted check structures. Those structures not only made the computations much more heavy, but they were also not as effective as this simpler one. This way the algorithm has some kind of "best effort" style, but along with the considerations made in the main algorithm and the pheromone update it works.

9.3 Pheromone updating

```python
def update_pheromone_t(pheromone, ant_paths, ant_weights, ant_n_weights, rho, use_sum_weight=True):
    new_pheromone = pheromone #Initialize new pheromone levels
    new_pheromone *= (1 - rho) #Apply decay rate
    for i in range(len(ant_paths)): #For each ant
        n_nodes = 0 #Counter needed to see if the ant traveled all the graph
        penalty = 1 #Multiplicative penalty for the ants that did not traverse the whole graph
        for k in ant_paths[i]: #For every agent
            n_nodes += len(ant_paths[i][k])-2 #Add the length of the path of the agent to a cumulative sum (minus starting and ending point)
        if (n_nodes+1) != len(job_list): #If the sum of all the nodes traversed by every agent is not equal to every node in the graph
            penalty = (n_nodes)/len(job_list) #Set the penalty as the ratio between the number of nodes and the nodes traversed
        for n in ant_paths[i]: #For each agent in ant_path
```

```python
            for u, v in zip(ant_paths[i][n], ant_paths[i][n][1:]): #Use zip in order to create an iterable of the path
                if use_sum_weight:
                    new_pheromone[u, v] += penalty / ant_weights[i] #Add pheromone values for each edge traveled inverse to the total weight of the path traveled
                else:
                    new_pheromone[u, v] += penalty / (ant_weights[i]/ant_n_weights[i][n]) #If we would prefer to update the pheromones using the individual weights we would need to do it this way
    return new_pheromone
```

The decay operation as we can see is identical to the one implemented before, but after that as we can see, we inserted a new concept: **penalty** for not terminating the whole path.

The idea of inserting the penalty came from seeing that, since the optimization problem was focused on getting the lowest weight possible, the agents started choosing nodes in a way that would not allow them to take as many other tasks as possible, keeping the weight very low. Observing this behaviour we decided to borrow the penalty concept from the reinforced learning approaches: we introduce a parameter that discurages certain unwanted behaviours. In our case, specifically, we used the ratio between traversed nodes and the total number of nodes:

$$P = \frac{\sum_{i=0}^{n_a-1} k_i}{N} \tag{2}$$

where P is the penalty, n_a is the number of agents, k_i is the number of nodes traversed by the agent i and N is the total number of nodes in the graph.

As we can see this is a quantity that will always be in range $[0, 1]$, so we use it as a multiplicative factor in the new pheromone computation. This way we inflict the penalty reducing the pheromone values in a way that is proportional to the quantity of edges not traveled.

We also added another penalty (less sophisticated) to the denominator of the pheromone computation for the spitted weights, trying to reinforce a behaviour that discourages a single agent to choose its path in a way that would not allow it to take many jobs, hence reducing its weight.

9.4 Main algorithm

```python
def timed_ant_colony(G, job_list, weight_matrix, n_agents=4, n_ants=20, n_iterations=100, alpha=3.0, beta=1.0, rho=0.3, pher_sum_weight=False, starting_time = datetime.datetime(2023, 11, 21, 10, 00)):
```

```
...
results = {"path": [], "weight": [], "n_weights": [], "time": []} #Initialize results dictionary
for iteration in range(n_iterations): #For every iteration
    ...
    ant_times = []
    for ant in range(n_ants): #For every ant
        ...
        time[n] = [(starting_time, starting_time)]
        wait_time = {} #Initialize waiting times
        for n in range(n_agents): #For every agent
            ...
            time[n] = [(starting_time, starting_time)]
            wait_time[n] = 0
        n = -1 #Initialize the agents iterator
        unvisited_nodes = [i for i in range(len(job_list))][1:] #Initialize the unvisited nodes list
        counter = 0
        while len(unvisited_nodes) != 0: #Until all nodes are visited
            n = (n + 1) % n_agents #Changing agent at each iteration
            next_node = get_next_node_t(weight_matrix, job_list, pheromone, ant_path[n], unvisited_nodes, epsilon, alpha, beta, time[n], wait_time[n]) #Select the next node
            if next_node == False: #If the next_node function returns False
                wait_time[n] += (900*random.random()) #Wait 15 minutes or less
                counter+=1
                if counter == 1000:
                    break
                continue
            ant_path[n].append(next_node) #Append the node to the path of the agent
            unvisited_nodes.remove(next_node) #Update the list of visited nodes
            time[n].append((time[n][-1][1] + datetime.timedelta(seconds=wait_time[n]), time[n][-1][1] + datetime.timedelta(seconds=300) + datetime.timedelta(seconds=wait_time[n]) + datetime.timedelta(seconds=weight_matrix[ant_path[n][-2], ant_path[n][-1]])))
            wait_time[n] = 0
        for n in range(len(ant_path)): #For every agent
```

```
                    wait_time[n] = 0
                    ant_path[n].append(0) #If not, append the end point of the path
                    time[n].append((time[n][-1][1], time[n][-1][1] + datetime.timedelta(seconds=300) + datetime.timedelta(seconds=weight_matrix[ant_path[n][-2], ant_path[n][-1]])))
            ...
            ant_times.append(time)
        pheromone = update_pheromone_t(pheromone, ant_paths, ant_weights, ant_n_weights, rho, use_sum_weight=pher_sum_weight) #Update pheromones on the edges of the graph
        ...
```

As we can see, first of all we added a new field to the results ('time'), we added a new time list for every iteration, and a time list for every ant. We also added a dictionary called wait_time, that is needed to manage the time constraints of the algorithm (when an agent does not have nothing to do it starts idling).

The times are defined as tuple that specify us the starting task time and the ending task time (comprised of 5 minutes for picking/dropping the load).

As we said before we now use an unvisited_nodes list rather than a visited_nodes list. This helps as to change the while loop condition from continuing until the visited nodes has as many nodes as the job list to continuing unitl the unvisited nodes list is empty.

Now, instead of ending the path of an agent that does not have a next node to go to, we put in idling, adding a random waiting time to its time variable (the random time is never higher than a quarter of an hour). We then increment a shared counter. If the agents bring the counter to a value of 1000 (at maximum 62 hours of wait), we exit the while loop.

Each time we add a new node to a path we must now update the times of the agents, adding the eventual wait time to the start of the task, and adding the wait time, the operation time and the trip time to the end of the task. We also need to do this at the end of the path, adding no waiting times this time (the agent remained idling waiting for a new job that it never took).

10 Results and analysis

The results we observed experimenting with our study case have been encouraging, but not optimal.

The ant colony algorithm indeed proved to be a powerful tool to obtain in rapid times near optimal solutions, but we surely can't isolate the performances of the algorithm to just its computation. In fact, we need to analyse the times to fetch the data also. Even if we would consider that we have the graph already in memory, we would still need to geocode the addresses and find the path weight between nodes (in our implementation we considered just the lowest weight path, but in a more realistic implementation might be useful to use and compare k different paths for each couple of nodes, and that would have our computational complexity exponentially rise).

Mean times	
Geocoding	0:01:35.132227
Weigth matrix computation	0:54:28.160136
Ant colony	0:00:02.171035

Table 2: The mean times for the ant colony algorithm without time management

As we can see, the time performances for the ant colony algorithm alone are not bad, but we have to consider all the operations as a whole, so in truth we have an algorithm that takes about an hour to find us the best path between 50 nodes (the number is inflated with respect to the original 41 from the excel file because we added the dummy tasks for the orphan deliveries).

Given the original premises of the project, that is to say that the ant colony algorithm would be used with the system at rest, and not in a real time calculation scenario these performances might still be satisfying.

For what regards the actual results, the algorithm returns paths that have a mean cumulative weight that is always about an hour and half, with a low variance, so the load is split in an almost equal way.

Regarding the ant colony algorithm with time management the results are a bit more complex to analyze, because we managed to modify the algorithm in a functional way, adding time constraints and control, but yet we see some undesirable behaviours.

Since the key concept of the algorithm is to minimize the weight of the path, and the algorithm was never meant to have more than one agent concurring with others, each agent will try to find the best solution for itself, where the best solution is the one that minimize the weight. We tried many different approaches in order to minimize this load unbalancing but it seems like it is not possible to avoid it completely.

Anyway we can compare our results with the results from another implementation of the same problem.

Id	Reference	Address	Start travel	End travel	Start task	End task	Duration	Category	Start large	End large	Complete range
30	PastaFra 01_2023-11-20	Via Filippo Argelati, 7, 20143 Milano MI, Italy	2023-11-21 13:23:43.698000	2023-11-21 13:35:12.738000	2023-11-21 13:35:12.738000	2023-11-21 13:40:12.738000	0.003472	pick_up	2023-11-21 13:00:00	2023-11-21 14:30:00	0.062500
1	Frida's Corso Genova02_2023-11-21	Corso Genova, 6, 20123 Milano MI, Italy	2023-11-21 13:40:12.738000	2023-11-21 13:47:34.692000	2023-11-21 13:47:34.692000	2023-11-21 13:52:34.692000	0.003472	pick_up	2023-11-21 12:15:00	2023-11-21 15:00:00	0.114583
8	La teiera Eclettica01_2023-11-21	La Teiera Eclettica, Via Melzo, 30, Milano, MI, Italia	2023-11-21 13:52:34.692000	2023-11-21 14:04:38.706000	2023-11-21 14:04:38.706000	2023-11-21 14:09:38.706000	0.003472	pick_up	2023-11-21 11:00:00	2023-11-21 16:00:00	0.208333
6	Fiori Fumagalli03_2023-11-21	Piazza Guglielmo Oberdan, 2, Milano, MI, Italia	2023-11-21 14:09:38.706000	2023-11-21 14:11:13.440000	2023-11-21 14:11:13.440000	2023-11-21 14:16:13.440000	0.003472	pick_up	2023-11-21 14:00:00	2023-11-21 16:15:00	0.093750
23	Dolcemente01_2023-11-20	Via Privata Giovanni Ventura, 5, Milano, MI, Italia	2023-11-21 14:16:13.440000	2023-11-21 14:29:36.672000	2023-11-21 14:29:36.672000	2023-11-21 14:34:36.672000	0.003472	drop_off	2023-11-21 09:15:00	2023-11-21 17:00:00	0.322917
9	Dolcemente01_2023-11-20	Via Mario Pichi, 8, Milano, MI, Italia	2023-11-21 14:34:36.672000	2023-11-21 15:05:06.120000	2023-11-21 15:05:06.120000	2023-11-21 15:10:06.120000	0.003472	drop_off	2023-11-21 11:00:00	2023-11-21 17:00:00	0.250000
4	Clori03_2023-11-21	Via Larga, 19, Milan, Metropolitan City of Milan, Italy	2023-11-21 15:10:06.120000	2023-11-21 15:20:34.014000	2023-11-21 15:20:34.014000	2023-11-21 15:25:34.014000	0.003472	drop_off	2023-11-21 12:00:00	2023-11-21 17:00:00	0.208333
3	Cafezal02_2023-11-21	Via Larga, 13, Milano, MI, Italia	2023-11-21 15:25:34.014000	2023-11-21 15:25:44.292000	2023-11-21 15:25:44.292000	2023-11-21 15:30:44.292000	0.003472	drop_off	2023-11-21 12:15:00	2023-11-21 17:00:00	0.197917
29	PastaFra 01_2023-11-20	Via Enrico Annibale Butti, 18, 20158 Milano MI, Italia	2023-11-21 15:30:44.292000	2023-11-21 15:50:39.924000	2023-11-21 15:50:39.924000	2023-11-21 15:55:39.924000	0.003472	drop_off	2023-11-21 13:45:00	2023-11-21 17:00:00	0.135417
7	La teiera Eclettica01_2023-11-21	Via Piero della Francesca, 74, Milano, MI, Italia	2023-11-21 15:55:39.924000	2023-11-21 16:05:21.882000	2023-11-21 16:05:21.882000	2023-11-21 16:10:21.882000	0.003472	drop_off	2023-11-21 14:00:00	2023-11-21 17:00:00	0.125000
2	Fiori Fumagalli03_2023-11-21	Foro Buonaparte, 22, Milano, MI, Italia	2023-11-21 16:10:21.882000	2023-11-21 16:21:43.326000	2023-11-21 16:21:43.326000	2023-11-21 16:26:43.326000	0.003472	drop_off	2023-11-21 14:15:00	2023-11-21 17:00:00	0.114583
0	Frida's Corso Genova02_2023-11-21	Via Ettore Ponti, 58, Milano, MI, Italia	2023-11-21 16:26:43.326000	2023-11-21 16:42:28.704000	2023-11-21 16:42:28.704000	2023-11-21 16:47:28.704000	0.003472	drop_off	2023-11-21 15:00:00	2023-11-21 17:00:00	0.083333
19	Pasticceria Spezia01_2023-11-20	Via la Spezia, 26, Milano, MI, Italia	2023-11-21 16:47:28.704000	2023-11-21 16:54:03.714000	2023-11-21 16:54:03.714000	2023-11-21 16:59:03.714000	0.003472	pick_up	2023-11-21 16:30:00	2023-11-21 17:30:00	0.041667
28	PastaFra 01_2023-11-20	Via Giuseppe Tartini, 14, 20158 Milano MI, Italy	2023-11-21 16:59:03.714000	2023-11-21 17:29:09.780000	2023-11-21 17:29:09.780000	2023-11-21 17:34:09.780000	0.003472	drop_off	2023-11-21 15:00:00	2023-11-21 18:00:00	0.125000
5	Fiori Fumagalli03_2023-11-21	Via Messina, 47, Milano, MI, Italia	2023-11-21 17:34:09.780000	2023-11-21 17:45:44.040000	2023-11-21 17:45:44.040000	2023-11-21 17:50:44.040000	0.003472	drop_off	2023-11-21 17:30:00	2023-11-21 18:00:00	0.020833
18	Pasticceria Spezia01_2023-11-20	Via Amedei, 8, 20123 Milano MI, Italy	2023-11-21 17:50:44.040000	2023-11-21 18:04:41.508000	2023-11-21 18:04:41.508000	2023-11-21 18:09:41.508000	0.003472	drop_off	2023-11-21 17:00:00	2023-11-21 18:30:00	0.062500
33	1950_450_92	Giuseppe, Via Giuseppe Ripamonti, 27, 20135 Milano MI, Italy	2023-11-21 18:09:41.508000	2023-11-21 18:17:41.208000	2023-11-21 18:17:41.208000	2023-11-21 18:22:41.208000	0.003472	drop_off	2023-11-21 05:00:00	2023-11-21 20:00:00	0.625000
25	1952_452_92	Via Santa Sofia, 27, 20122 Milano MI, Italy	2023-11-21 18:22:41.208000	2023-11-21 18:29:50.832000	2023-11-21 18:29:50.832000	2023-11-21 18:34:50.832000	0.003472	drop_off	2023-11-21 05:00:00	2023-11-21 20:00:00	0.625000
24	Walter Calzature01_2023-11-17	via Val di Fiemme, 25, 20128 Milano MI, Italy	2023-11-21 18:34:50.832000	2023-11-21 18:57:22.182000	2023-11-21 18:57:22.182000	2023-11-21 19:02:22.182000	0.003472	drop_off	2023-11-21 08:00:00	2023-11-21 20:00:00	0.500000
32	Walter Calzature03_2023-11-20	Via Monte Rosa, 17, Milano, MI, Italia	2023-11-21 19:02:22.182000	2023-11-21 19:34:02.442000	2023-11-21 19:34:02.442000	2023-11-21 19:39:02.442000	0.003472	drop_off	2023-11-21 11:30:00	2023-11-21 20:00:00	0.354167
31	Walter Calzature03_2023-11-20	Via Forze Armate, 50, Milano, MI, Italia	2023-11-21 19:39:02.442000	2023-11-21 19:46:05.640000	2023-11-21 19:46:05.640000	2023-11-21 19:51:05.640000	0.003472	drop_off	2023-11-21 11:30:00	2023-11-21 20:00:00	0.354167
41	Alveare Terzo Tempo01_2023-11-20	Via Giuseppe Frua, 26, Milano, MI, Italia	2023-11-21 19:51:05.640000	2023-11-21 19:55:00	2023-11-21 19:55:00	2023-11-21 20:00:00	0.003472	drop_off	2023-11-21 17:15:00	2023-11-21 20:00:00	0.114583

Table 3: Solution for agent 1

Id	Reference	Address	Start travel	End travel	Start task	End task	Duration	Category	Start large	End large	Complete range
11	Florarredo01_2023-11-21	Piazza della Repubblica, 32, 20124 Milano MI, Italy	2023-11-21 14:07:50.304000	2023-11-21 14:22:48.432000	2023-11-21 14:22:48.432000	2023-11-21 14:27:48.432000	0.003472	pick_up	2023-11-21 10:45:00	2023-11-21 16:00:00	0.218750
10	Florarredo01_2023-11-21	Via Gaetano de Castillia, 6A, Milano, MI, Italia	2023-11-21 14:27:48.432000	2023-11-21 14:32:30.582000	2023-11-21 14:32:30.582000	2023-11-21 14:37:30.582000	0.003472	drop_off	2023-11-21 11:00:00	2023-11-21 17:00:00	0.250000
27	Cosaporto_603053	Corso Lodi, 15, 20135 Milano MI, Italy	2023-11-21 14:37:30.582000	2023-11-21 14:55:00	2023-11-21 14:55:00	2023-11-21 15:00:00	0.003472	pick_up	2023-11-21 14:00:00	2023-11-21 15:00:00	0.041667
26	Cosaporto_603053	Via Morozzo della Rocca, 8, Milano, MI, Italia	2023-11-21 15:41:33.024000	2023-11-21 15:55:00	2023-11-21 15:55:00	2023-11-21 16:00:00	0.003472	drop_off	2023-11-21 14:00:00	2023-11-21 16:00:00	0.083333
13	Pasticceria Grecchi Luigi01_2023-11-21	Via Piero della Francesca, 7, 20154 Milano MI, Italy	2023-11-21 16:46:24.156000	2023-11-21 16:55:00	2023-11-21 16:55:00	2023-11-21 17:00:00	0.003472	pick_up	2023-11-21 16:30:00	2023-11-21 17:00:00	0.020833
17	Cosaporto_603825	Via Felice Casati, 27, 20124 Milano, Milano MI, Italia	2023-11-21 17:43:08.118000	2023-11-21 17:55:00	2023-11-21 17:55:00	2023-11-21 18:00:00	0.003472	pick_up	2023-11-21 17:00:00	2023-11-21 18:00:00	0.041667
12	Pasticceria Grecchi Luigi01_2023-11-21	Via Felice Casati, 45, 20124 Milano MI, Italy	2023-11-21 18:24:13.848000	2023-11-21 18:25:00	2023-11-21 18:25:00	2023-11-21 18:30:00	0.003472	drop_off	2023-11-21 17:00:00	2023-11-21 18:30:00	0.062500
16	Cosaporto_603825	Via Giovanni Cimabue, 41, Milano MI, Italia	2023-11-21 18:33:10.482000	2023-11-21 18:55:00	2023-11-21 18:55:00	2023-11-21 19:00:00	0.003472	drop_off	2023-11-21 17:00:00	2023-11-21 19:00:00	0.083333

Table 4: Solution for agent 2

Id	Reference	Address	Start travel	End travel	Start task	End task	Duration	Category	Start large	End large	Complete range
22	Bibilab02_2023-11-20	Via Mario Pannunzio, 4, 20156 Milano MI, Italy	2023-11-21 14:45:01.128000	2023-11-21 15:05:26.316000	2023-11-21 15:05:26.316000	2023-11-21 15:10:26.316000	0.003472	pick_up	2023-11-21 15:00:00	2023-11-21 16:00:00	0.041667
15	Au Nome de la Rose01_2023-11-21	Piazza Riccardo Wagner, 20145 Milano MI, Italy	2023-11-21 15:10:26.316000	2023-11-21 15:25:00	2023-11-21 15:25:00	2023-11-21 15:30:00	0.003472	pick_up	2023-11-21 12:00:00	2023-11-21 15:30:00	0.145833
20	Bibilab02_2023-11-20	Via Seprio, 2, Milano, MI, Italia	2023-11-21 16:01:15.474000	2023-11-21 16:03:45.450000	2023-11-21 16:03:45.450000	2023-11-21 16:08:45.450000	0.003472	drop_off	2023-11-21 15:15:00	2023-11-21 17:00:00	0.072917
21	Bibilab02_2023-11-20	Via Caminadella, 18, 20123 Milano MI, Italy	2023-11-21 16:08:45.450000	2023-11-21 16:16:28.104000	2023-11-21 16:16:28.104000	2023-11-21 16:21:28.104000	0.003472	drop_off	2023-11-21 15:15:00	2023-11-21 17:00:00	0.072917
14	Au Nome de la Rose01_2023-11-21	Via Moneta, 1, Milano, MI, Italia	2023-11-21 16:21:28.104000	2023-11-21 16:25:00	2023-11-21 16:25:00	2023-11-21 16:30:00	0.003472	drop_off	2023-11-21 12:15:00	2023-11-21 16:30:00	0.177083
40	Alveare del Naviglio01_2023-11-20	Via Don Rodrigo, 3, 20142 Milano MI, Italy	2023-11-21 17:42:33.612000	2023-11-21 17:56:41.700000	2023-11-21 17:56:41.700000	2023-11-21 18:01:41.700000	0.003472	pick_up	2023-11-21 17:00:00	2023-11-21 18:30:00	0.062500
39	Alveare del Naviglio01_2023-11-20	Via Nicola Romeo, 5, 20142 Milano MI, Italia	2023-11-21 18:01:41.700000	2023-11-21 18:13:25.428000	2023-11-21 18:13:25.428000	2023-11-21 18:18:25.428000	0.003472	drop_off	2023-11-21 17:15:00	2023-11-21 20:00:00	0.114583
38	Alveare Basiglio01_2023-11-20	Via Borgo del Majno, 1a, 20080 Basiglio, MI, Italy	2023-11-21 18:18:25.428000	2023-11-21 18:44:40.554000	2023-11-21 18:44:40.554000	2023-11-21 18:49:40.554000	0.003472	pick_up	2023-11-21 17:30:00	2023-11-21 19:00:00	0.062500
35	Alveare Basiglio01_2023-11-20	Vicolo Torretta, 18, 27010 Siziano PV, Italia	2023-11-21 18:49:40.554000	2023-11-21 19:09:19.770000	2023-11-21 19:09:19.770000	2023-11-21 19:14:19.770000	0.003472	drop_off	2023-11-21 17:45:00	2023-11-21 20:00:00	0.093750
36	Alveare Basiglio01_2023-11-20	Via dei Pini, 2l, 20072 Pieve Emanuele MI, Italy	2023-11-21 19:14:19.770000	2023-11-21 19:25:17.508000	2023-11-21 19:25:17.508000	2023-11-21 19:30:17.508000	0.003472	drop_off	2023-11-21 17:45:00	2023-11-21 20:00:00	0.093750
37	Alveare Basiglio01_2023-11-20	Via dei Pini, 4, 20090 Pieve Emanuele MI, Italia	2023-11-21 19:30:17.508000	2023-11-21 19:31:23.262000	2023-11-21 19:31:23.262000	2023-11-21 19:36:23.262000	0.003472	drop_off	2023-11-21 17:45:00	2023-11-21 20:00:00	0.093750
34	Alveare Basiglio01_2023-11-20	Via Pietro Nenni, 40, 20085 Locate di Triulzi MI, Italia	2023-11-21 19:36:23.262000	2023-11-21 19:55:00	2023-11-21 19:55:00	2023-11-21 20:00:00	0.003472	drop_off	2023-11-21 17:45:00	2023-11-21 20:00:00	0.093750

Table 5: Solution for agent 3

From the results shown in table 3, 4 and 5, we can see that first of all there are a few key differences between this implementation and our solution:

- Our solution presupposes that all vehicles begin their travel at the same exact time, so we do not make an optimization on the time at which the vehicles should take off.
- We added some fake jobs in the list to solve the issue of the "orphan" drop offs (without a related pick up with the same reference).
- Our solution does not give us an analogous of "Start task" and "End task", but we just imply the 5 minutes for the pick up or drop off task between a travel and the next in "Start travel" and "End travel".
- Our solutions also include the travel time to return to the warehouse.
- Our solution also does not have the categories "Duration" and "Complete range".

Aside from this, still the solutions have enough similarities to be comparable, so we will now present the results from our implementation:

Id	Reference	Address	Start travel	End travel	Category
Special	Warehouse	Via Olona, 2, 20123 Milano MI, Italy	2023/11/21, 10:00	2023/11/21, 10:00	
Special	Walter Calzature03_2023-11-20	Via Olona, 2, 20123 Milano MI, Italy	2023/11/21, 10:00	2023/11/21, 10:05	pick_up
Special	Walter Calzature01_2023-11-17	Via Olona, 2, 20123 Milano MI, Italy	2023/11/21, 10:05	2023/11/21, 10:10	pick_up
Special	Alveare Terzo Tempo01_2023-11-20	Via Olona, 2, 20123 Milano MI, Italy	2023/11/21, 10:10	2023/11/21, 10:15	pick_up
24	Walter Calzature01_2023-11-17	via Val di Fiemme, 25, 20128 Milano MI, Italy	2023/11/21, 10:15	2023/11/21, 10:32	drop_off
11	Florarredo01_2023-11-21	Piazza della Repubblica, 32, 20124 Milano MI, Italy	2023/11/21, 10:56	2023/11/21, 11:08	pick_up
10	Florarredo01_2023-11-21	Via Gaetano de Castillia, 6A, Milano, MI, Italia	2023/11/21, 11:08	2023/11/21, 11:15	drop_off
8	La teiera Eclettica01_2023-11-21	La Teiera Eclettica, Via Melzo, 30, Milano, MI, Italia	2023/11/21, 11:15	2023/11/21, 11:24	pick_up
31	Walter Calzature03_2023-11-20	Via delle Forze Armate, 50, Milano, MI, Italia	2023/11/21, 11:40	2023/11/21, 11:55	drop_off
32	Walter Calzature03_2023-11-20	Via Monte Rosa, 17, Milano, MI, Italia	2023/11/21, 12:00	2023/11/21, 12:30	drop_off
7	La teiera Eclettica01_2023-11-21	Via Piero della Francesca, 74, Milano, MI, Italia	2023/11/21, 14:21	2023/11/21, 14:35	drop_off
41	Alveare Terzo Tempo01_2023-11-20	Via Giuseppe Frua, 26, Milano, MI, Italia	2023/11/21, 17:45	2023/11/21, 18:14	drop_off
Special	Warehouse	Via Olona, 2, 20123 Milano MI, Italy	2023/11/21, 18:14	2023/11/21, 18:24	

Table 6: Ant colony algorithm with time constraints solution for agent 1

Id	Reference	Address	Start travel	End travel	Category
Special	Warehouse	Via Olona, 2, 20123 Milano MI, Italy	2023/11/21, 10:00	2023/11/21, 10:00	
Special	Dolcemente01_2023-11-20	Via Olona, 2, 20123 Milano MI, Italy	2023/11/21, 10:00	2023/11/21, 10:05	pick_up
Special	Cafezal02_2023-11-21	Via Olona, 2, 20123 Milano MI, Italy	2023/11/21, 10:05	2023/11/21, 10:10	pick_up
Special	1952_452_92	Via Olona, 2, 20123 Milano MI, Italy	2023/11/21, 10:10	2023/11/21, 10:15	pick_up
25	1952_452_92	Via Santa Sofia, 27, 20122 Milano MI, Italy	2023/11/21, 10:15	2023/11/21, 10:21	drop_off
23	Dolcemente01_2023-11-20	Via Privata Giovanni Ventura, 5, Milano, MI, Italia	2023/11/21, 10:21	2023/11/21, 10:35	drop_off
9	Dolcemente01_2023-11-20	Via Mario Pichi, 8, Milano, MI, Italia	2023/11/21, 11:12	2023/11/21, 11:29	drop_off
15	Au Nome de la Rose01_2023-11-21	Piazza Riccardo Wagner, 20145 Milano MI, Italy	2023/11/21, 12:11	2023/11/21, 12:22	pick_up
14	Au Nome de la Rose01_2023-11-21	Via Moneta, 1, Milano, MI, Italia	2023/11/21, 12:22	2023/11/21, 12:31	drop_off
3	Cafezal02_2023-11-21	Via Larga, 13, Milano, MI, Italia	2023/11/21, 12:31	2023/11/21, 12:38	drop_off
27	Cosaporto_603053	Corso Lodi, 15, 20135 Milano MI, Italy	2023/11/21, 14:07	2023/11/21, 14:15	pick_up
26	Cosaporto_603053	Via Morozzo della Rocca, 8, Milano, MI, Italia	2023/11/21, 14:15	2023/11/21, 14:26	drop_off
13	Pasticceria Grecchi Luigi01_2023-11-21	Via Piero della Francesca, 7, 20154 Milano MI, Italy	2023/11/21, 16:33	2023/11/21, 16:41	drop_off
19	Pasticceria Spezia01_2023-11-20	Via Spezia, 26, Milano, MI, Italia	2023/11/21, 16:41	2023/11/21, 16:54	pick_up
40	Alveare del Naviglio01_2023-11-20	Via Don Rodrigo, 3, 20142 Milano MI, Italy	2023/11/21, 17:03	2023/11/21, 17:10	pick_up
18	Pasticceria Spezia01_2023-11-20	Via Amedei, 8, 20123 Milano MI, Italy	2023/11/21, 17:10	2023/11/21, 17:21	drop_off
12	Pasticceria Grecchi Luigi01_2023-11-21	Via Felice Casati, 45, 20124 Milano MI, Italy	2023/11/21, 17:21	2023/11/21, 17:32	drop_off
39	Alveare del Naviglio01_2023-11-20	Via Nicola Romeo, 5, 20142 Milano MI, Italia	2023/11/21, 17:32	2023/11/21, 17:48	drop_off
38	Alveare Basiglio01_2023-11-20	Via Borgo del Maino, 1a, 20080 Basiglio Basiglio MI, Italy	2023/11/21, 17:48	2023/11/21, 18:04	pick_up
37	Alveare Basiglio01_2023-11-20	Via dei Pini, 4, 20090 Pieve Emanuele MI, Italia	2023/11/21, 18:04	2023/11/21, 18:15	drop_off
36	Alveare Basiglio01_2023-11-20	Via dei Pini, 2l, 20072 Pieve Emanuele MI, Italy	2023/11/21, 18:15	2023/11/21, 18:20	drop_off
34	Alveare Basiglio01_2023-11-20	Via Pietro Nenni, 40, 20085 Locate di Triulzi MI, Italia	2023/11/21, 18:20	2023/11/21, 18:32	drop_off
35	Alveare Basiglio01_2023-11-20	Vicolo Torretta, 18, 27010 Siziano PV, Italia	2023/11/21, 18:32	2023/11/21, 18:46	drop_off
Special	Warehouse	Via Olona, 2, 20123 Milano MI, Italy	2023/11/21, 18:46	2023/11/21, 19:13	

Table 7: Ant colony algorithm with time constraints solution for agent 2

Id	Reference	Address	Start travel	End travel	Category
Special	Warehouse	Via Olona, 2, 20123 Milano MI, Italy	2023/11/21, 10:00	2023/11/21, 10:00	
Special	1950_450_92	Via Olona, 2, 20123 Milano MI, Italy	2023/11/21, 10:00	2023/11/21, 10:05	pick_up
Special	Clori03_2023-11-21	Via Olona, 2, 20123 Milano MI, Italy	2023/11/21, 10:05	2023/11/21, 10:10	pick_up
33	1950_450_92	Via Giuseppe Ripamonti, 27, 20135 Milano MI, Italy	2023/11/21, 10:10	2023/11/21, 10:18	drop_off
4	Clori03_2023-11-21	Via Larga, 19, Milan, Metropolitan City of Milan, Italy	2023/11/21, 12:09	2023/11/21, 12:18	drop_off
1	Frida's Corso Genova02_2023-11-21	Corso Genova, 6, 20123 Milano MI, Italy	2023/11/21, 12:18	2023/11/21, 12:27	pick_up
30	PastaFra 01_2023-11-20	Via Filippo Argelati, 7, 20143 Milano MI, Italy	2023/11/21, 13:05	2023/11/21, 13:14	pick_up
29	PastaFra 01_2023-11-20	Via Enrico Annibale Butti, 18, 20158 Milano MI, Italia	2023/11/21, 14:00	2023/11/21, 14:15	drop_off
6	Fiori Fumagalli03_2023-11-21	Piazza Guglielmo Oberdan, 2, Milano, MI, Italia	2023/11/21, 14:15	2023/11/21, 14:26	pick_up
2	Fiori Fumagalli03_2023-11-21	Foro Buonaparte, 22, Milano, MI, Italia	2023/11/21, 14:26	2023/11/21, 14:35	drop_off
28	PastaFra 01_2023-11-20	Via Giuseppe Tartini, 14, 20158 Milano MI, Italy	2023/11/21, 15:06	2023/11/21, 15:16	drop_off
22	Bibilab02_2023-11-20	Via Mario Pannunzio, 4, 20156 Milano MI, Italy	2023/11/21, 15:16	2023/11/21, 15:28	pick_up
21	Bibilab02_2023-11-20	Via Caminadella, 18, 20123 Milano MI, Italy	2023/11/21, 15:28	2023/11/21, 15:41	drop_off
20	Bibilab02_2023-11-20	Via Seprio, 2, Milano, MI, Italia	2023/11/21, 15:41	2023/11/21, 15:51	drop_off
0	Frida's Corso Genova02_2023-11-21	Via Ettore Ponti, 58, Milano, MI, Italia	2023/11/21, 15:51	2023/11/21, 16:01	drop_off
17	Cosaporto_603825	Via Felice Casati, 27, 20124 Milano, Milano MI, Italia	2023/11/21, 17:11	2023/11/21, 17:26	pick_up
16	Cosaporto_603825	Via Giovanni Cimabue, 41, Milano MI, Italia	2023/11/21, 17:26	2023/11/21, 17:46	drop_off
5	Fiori Fumagalli03_2023-11-21	Via Messina, 47, Milano, MI, Italia	2023/11/21, 17:53	2023/11/21, 18:15	drop_off
Special	Warehouse	Via Olona, 2, 20123 Milano MI, Italy	2023/11/21, 18:15	2023/11/21, 18:26	

Table 8: Ant colony algorithm with time constraints solution for agent 3

The main difference, as we can see, is that our solution has a lot of idling time.

Agent	Total time (hours)	Travel time (hours)	Pick up/drop off time (hours)	Idling time (hours)	Idling time percentage
1	8.4	1.64	0.91	5.85	69.64%
2	9.21	2.23	1.83	5.15	55.91%
3	8.43	1.89	1.41	5.13	60.85%

Table 9: Ant colony solution times

Probably if we were to run the time simulation with a starting time set later than 10:00, we could minimize these dead times.

Besides that, our solution focuses a lot on trying to get similar working times for all the agents, and this is visible from the low variance we have with the times defined by our solutions.

It could be absolutely possible to further improve our solution by using dynamic time starts and introducing a penalty for the idling time of the agent.

11 Final thoughts

The idea of implementing the ant colony algorithm for a pathfinding problem is surely efficient, since the algorithm can give us near optimal solutions in short amount of times (premise that we have no urgency to get the weight matrix and query the geocode). But, when we add time management to the mix, we can see some problems:

- The solutions space critically decrease, leaving us with a bunch of solutions, not nearly approximable to the $N!$ solutions of the problem without any constraints.
- We lose some of the strengths of the ant colony algorithm: we can't freely explore the graph in order to find a global minimum, but we are constrained by some external factors, making the ant colony algorithm concept less useful.
- Inserting the time management in the ant colony algorithm is a non trivial task, that requires us to cover a lot of cases non existing in the vanilla algorithm, making us question how convenient it is to do this modification.
- As we already mentioned, with these modifications, we are going more towards a discrete time simulation, that would probably be a better solution at this point rather than using a metaheuristic algorithm; maybe even a reinforcement learning solution might be more appropriate.

Yet, it is still possible to refine our solution with further studies, especially trying to improve the load balancing of the weight for each agent.