



European
Research
Council



Multimodal network Modelling and Simulation (MnMS)

DETAILED DOCUMENTATION

Authors:

Mélanie Cortina, Louis Balzer, Cécile Bécarie*, Manon Seppecher,
Ludovic Leclercq, Christophe Tettarassar

*Corresponding author: Université Gustave Eiffel, LICIT-ECO7, cecile.becarie@univ-eiffel.fr

April 30, 2024

Introduction

MnMS (Multimodal network Modelling and Simulation) is a multimodal dynamic traffic simulator designed for a large urban scale. It results from all research activities of the ERC MAGNUM project. Further extensions related to on-demand mobility have been developed with the DIT4TraM project.

MnMS is an agent-based dynamic simulator for urban mobility. Travelers make mode and route choices considering all multimodal options on the city transportation network, including traditional modes, such as personal cars or public transportation, and new mobility services, such as ride-hailing, ride-sharing, or vehicle sharing. Vehicles motion is governed by regional multimodal MFD (Macroscopic Fundamental Diagram) curves, so all vehicles of the same type (car, bus, etc.) share the same speed within a specific region at a given time. The adoption of this traffic flow modeling framework allows to address at large urban scale timely research topics such as the management of new mobility services (operation, optimization, regulation), the design of regulatory policies taking into account the multiple stakeholders setting of today's urban transportation system, and beyond!

In this documentation, the reader will find help installing MnMS and its dependencies, a detailed description of the different modules composing the simulator, and the available example scenarios. This documentation has been made detailed on purpose so it can help both MnMS users and developers grow knowledge about the models implemented, design their own simulation study case, and eventually contribute to the platform.

Note that as the product of public scientific research, MnMS regularly evolves. The LICIT-ECO7 team does its best to update this documentation consequently, but some elements may need to be updated. Do not hesitate to contact us via institutional email or GitHub.

To improve the document's readability, we have adopted the following color legend:

- **JungleGreen**: designates an MnMS class
- **Apricot**: designates an enum item
- **Violet**: designates a parameter default value
- **Gray**: designates a class attribute or parameter
- **Goldenrod**: designates a method or function
- **Green**: file name
- **Red**: still to develop or subject to modification

Acknowledgements

The research activities that led to this simulation package have received funding from the European Research Council (ERC) under the ERC CoG 2013 MAGNUM (grant agreement No 646592). They also benefited from other funding under the European Union's Horizon 2020 research and innovation program DIT4TraM (grant agreement No 953783).

Contents

1	Getting started	4
1.1	Installing MnMS and HiPOP	4
1.2	Running tests, examples, and tutorials	4
1.3	Automatically generated documentation	5
2	Transportation network	6
2.1	Network definition	6
2.1.1	Physical network	6
2.1.2	Multi layer graph	7
2.2	Network data	8
2.3	Dynamic space sharing	9
3	Demand management	10
3.1	Demand data	10
3.1.1	Users list	10
3.1.2	Mobility services graphs	11
3.2	User states	13
3.3	Planning (path allocation)	13
3.3.1	Scheduling	14
3.3.2	Events leading to (re)planning	14
3.3.3	Planning origin	14
3.3.4	Update available mobility services	15
3.3.5	Discovering shortest paths	16
3.3.6	Shortest paths computation	18
3.3.7	Travel cost estimation	19
3.3.8	Path selection	21
3.3.9	Start of path	21
3.4	Outputs	22
3.5	Warnings	23
4	Vehicles motion management	24
4.1	Vehicle types	24
4.2	Vehicle activities	25
4.3	Traffic dynamics	25
4.3.1	Speed computation in a reservoir	25
4.3.2	Vehicle motion	26
4.3.3	Queues between reservoirs	26
4.4	Outputs	26
5	Mobility services	26
5.1	Abstract classes	26
5.2	Personal mobility service	28
5.2.1	Maintenance	29
5.2.2	Periodic maintenance	29

5.2.3	Matching	29
5.2.4	Interface methods	29
5.3	Public Transport mobility service	29
5.3.1	Maintenance	30
5.3.2	Periodic maintenance	30
5.3.3	Matching	30
5.3.4	Interface methods	30
5.4	On demand mobility service	31
5.4.1	Maintenance	31
5.4.2	Periodic maintenance	31
5.4.3	Matching	31
5.4.4	Interface methods	32
5.5	On demand mobility service with depots	33
5.5.1	Maintenance	34
5.5.2	Periodic maintenance	34
5.5.3	Matching	34
5.5.4	Interface methods	34
5.6	Vehicle sharing mobility service	34
5.6.1	Maintenance	35
5.6.2	Periodic maintenance	35
5.6.3	Matching	35
5.6.4	Interface methods	35
5.7	On demand shared mobility service	36
5.7.1	Maintenance	37
5.7.2	Periodic maintenance	37
5.7.3	Matching	37
5.7.4	Interface methods	38
5.8	Outputs	38
6	Simulation	39
6.1	Simulation launching	39
6.2	Simulation flow chart	39
6.3	Computational saving modes	39
7	Study cases	42
7.1	Lyon 6	42

1 Getting started

1.1 Installing MnMS and HiPOP

The installation of MnMS should be done from sources. The first step is to download these sources from the proper repositories. To be able to get all updates in real-time, it is recommended to clone them. There are two repositories to clone, one for MnMS and another for HiPOP dependency, the module for high-performance shortest paths computation.

```
git clone https://github.com/licit-lab/MnMS.git
git clone https://github.com/licit-lab/HiPOP.git
```

After cloning, you can eventually checkout a branch you are interested in. The two main branches of the projects are `main` (last stable version) and `develop` (most up-to-date version). Then, one should create and configure a conda environment for running simulations with:

```
cd MnMS
conda env create -f conda/env.yaml
conda activate mnms
python -m pip install -e .
cd ../HiPOP/python
python install_cpp.py
python -m pip install .
```

If each step went well without error, you should be ready to launch simulations.

1.2 Running tests, examples, and tutorials

To verify your installation, you can launch the tests within the conda environment with:

```
cd MnMS
pytest tests/.
```

Also, there exists several simple examples you can run to start to understand how to create a simulation scenario and to use the different modules of MnMS. We describe these briefly in the following. For more advanced examples, refer to section 7

- **manhattan**: a Manhattan road network, one reservoir with a constant speed covering the whole network, one layer for one mobility service of type personal car, a grid origin-destination layer that fits the roads, and only one user choosing path deterministically based on the lowest travel time.
- **on_demand_car**: a Manhattan road network, one reservoir with a constant speed covering the whole network, one layer for two mobility services of types personal car and ride-hailing with depots, a grid origin-destination layer that fits the roads, and two users choosing path deterministically based on the lowest generalized travel cost. One can observe that one user chooses the ride-hailing service while the other chooses her car, the on-demand car reposition toward the depot after having achieved its serving mission.

- **public_transport**: a line road network with a bus line on it, one reservoir with a constant speed covering the whole network, one layer for one mobility service of type public transportation, an origin-destination layer matching the roads nodes, and only one user choosing path deterministically based on the lowest travel time.
- **ridesharing**: a Manhattan road network, one reservoir with a constant speed covering the whole network, one layer for one mobility service of type ride-sharing, an origin-destination layer matching with the roads nodes, a certain number of on-demand vehicles with initial positions randomly drawn among the graph nodes, and a certain demand specified with departure rates over time.
- **vehicle_sharing**: it includes three small examples:
 - station-based vehicle sharing: a Manhattan road network, one reservoir with a constant speed covering the whole network, one vehicle sharing layer for one station-based bike sharing mobility service with two stations, a grid origin-destination layer, and three users choosing path deterministically based on the lowest travel time.
 - free-floating vehicle sharing: a Manhattan road network, one reservoir with a constant speed covering the whole network, one vehicle sharing layer for one free-floating bike sharing mobility service with one bike only, a grid origin-destination layer, and three users choosing path deterministically based on the lowest travel time.
 - station-based vehicle sharing with intermodality: a Manhattan road network, one reservoir with a constant speed covering the whole network, the car speed in this zone is lower than the bike speed, one vehicle sharing layer for one station-based bike sharing mobility service with two stations, a grid origin-destination layer, and three users choosing path deterministically based on the lowest travel time. In this scenario choosing an intermodal path with car and bike is optimal.
- **intermodal**: a line road network where a car layer extends on the left and a bus line extends on the right, one reservoir with constant speeds for cars and buses and covering the whole network, a personal car mobility service runs on the car layer and a public transportation service runs on another layer, an origin-destination layer matching with the roads nodes, and only one user who should travel from the left to the right by taking her car, then the bus line.
- **congested_mfd**: a line road network, two reservoirs, one on the left part of the roads and another on the right part of the road, speed is a function of the accumulation and an entry function is defined for both, one layer for one mobility service of type personal car, an origin-destination layer matching the roads nodes, and several users choosing path deterministically based on the lowest travel time.

NB: For all example, the simulation log and observers for users, vehicles, and paths are activated. One can check them to understand the simulation flow and the outputs available.

1.3 Automatically generated documentation

To generate an automatic documentation from the code, one should first update the conda environment with new dependencies and then build the documentation.

```
conda activate mnms
cd MnMS
conda env update -f conda/doc.yaml
mkdocs serve
```

2 Transportation network

2.1 Network definition

The multimodal transportation graph definition should be consistent with the aim of the study. In this section, we introduce the main classes allowing to define this graph and present the main principles ruling the roads, layers, and connections definition to enable flexible intermodality of users paths.

2.1.1 Physical network

The `RoadDescriptor` represents the physical nodes and sections composing the whole transportation network. It includes road intersections and segments but also dedicated transportation infrastructures separated from the actual roads and not affected by road traffic (e.g. train or metro railways). Consequently, the graph defined by the nodes and sections of the `RoadDescriptor` is not necessarily fully connected. Additionally, Public Transportation (PT) stops should be defined over the sections. Note that defining one stop per PT station is required. One PT station is associated with one PT line for one direction. See Figure 2.

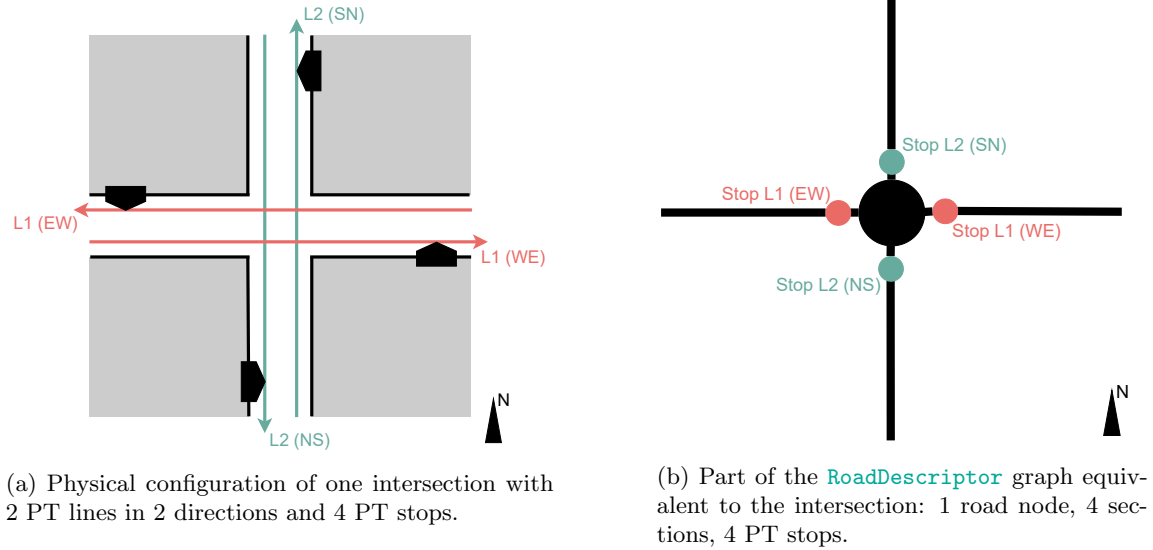


Figure 2: Example for the definition of PT stops.

2.1.2 Multi layer graph

The `MultiLayerGraph` object corresponds to the multimodal transportation graph. It gathers one or several `AbstractLayer` objects, that can be of several types, an `OriginDestinationLayer`, and a `TransitLayer` object. In the following, we describe the different types of layers.

- **AbstractLayer**: a layer corresponds to a part of the whole transportation graph. It gathers all the mobility services operating with the same type of vehicles running exactly on the same nodes and sections of the `RoadDescriptor`.
 - **NB1**: If one is looking for intermodal paths using two mobility services, it is mandatory to define them on two separate layers. Indeed, HIPOP considers one mobility service per layer, the one provided as the chosen service on this layer when we call the `parallel_k_shortest_path` function. It is possible to define several mobility services on the same layer with their respective cost functions, on the condition of not trying to get intermodal paths using these services. See Table 1.
 - **NB2**: A layer is characterized by one vehicle type. This is because when we update the speed on the links of a layer, we set the speed given by the MFD function for this vehicle type. The same speed value stands for all mobility services vehicles running on this layer.
 - **NB3**: Suppose one needs to define different behaviors for vehicles of the same type belonging to the same layer. For example, personal car and ride-hailing car are of type car but two sub-types can be defined: the ride-hailing car may have a different behavior, it can have additional activity type, pocket some money at drop-off, etc. Note that the vehicles of the two sub-types of the car vehicle type have their own behaviors but are considered as the same vehicle type in the trip-based MFD framework.

Considered modes	<ul style="list-style-type: none"> • personal car only • ride-hailing only 	<ul style="list-style-type: none"> • personal car only • ride-hailing only • personal car + ride-hailing 	
Layers definition	CarLayer	PCarLayer	RHailingLayer
Mobility services per layer	<ul style="list-style-type: none"> • PCarMS • UberMS • LyftMS 	<ul style="list-style-type: none"> • PCarMS 	<ul style="list-style-type: none"> • UberMS • LyftMS

Table 1: Example for the definition of layers and mobility services.

- **PublicTransportationLayer**: it inherits from the `AbstractLayer` class and corresponds to the specific case of PT. PT layer nodes are PT stops and links are PT line legs between two stops. Consequently, there is no stake in defining all PT types on the same layer.
 - **NB**: To be consistent with the "one vehicle type per layer" principle, it is recommended to define one PT layer per PT type (one for buses, one for metros, etc.)
- **VehicleSharingLayer**: it inherits from the `AbstractLayer` class and corresponds to the specific case of vehicle sharing. A `VehicleSharingLayer` has the particularity to be connected with the other layers through stations. The station can be physical or virtual (for

free-floating), and is dynamically create, connected and disconnected during the simulation depending if vehicles are parked in it or not.

- **OriginDestinationLayer**: This is a specific layer composed solely of origin and destination nodes. Each origin and destination node is connected to the other layers by transit links. All paths starts at an origin node of the origin-destination layer and end at a destination node of the origin-destination layer. The origin (respectively destination) defined in the demand for each user is either an origin (respectively destination) node of the origin destination layer or a point defined by its coordinates. In the latter case, the origin (respectively destination) node considered in MnMS is the nearest of the origin (respectively destination) node of the origin destination layer. The distance between the point and the node of the origin destination layer is ignored.
- **TransitLayer**: it gathers the links of type TRANSIT on which operates only one dummy mobility service which is WALK. It belongs to the user to correctly define the transit links of the **MultiLayerGraph** to enable the desired transportation modes considered by users by using the function **connect_layers**. Three types of connections may be defined:
 - **access/egress transit links**: connect the **OriginDestinationLayer** with the rest of the **MultiLayerGraph**. The function **connect_origin_destination_layer** is automatically called to proceed with a connection distance threshold when the **MultiLayerGraph** object is created.
 - **intra layer transit links**: connect two nodes belonging to the same layer.
 - **inter layer transit links**: connect two nodes belonging to different layers.

2.2 Network data

A **MultiLayerGraph** object can be saved in and loaded from a .json file with **save_graph** and **load_graph** functions. The .json file contains two main parts for the physical network and the multi layer graph. Its general structure is as follows:

```
{
  "ROADS": {
    "NODES": {
      "uz2_0": {
        "id": "uz2_0",
        "position": [12000.0,12000.0]},
      ...},
    "STOPS": {
      "Bus_NS1_0": {
        "id": "Bus_NS1_0",
        "section": "uz1_44_uz1_43",
        "relative_position": 0.0,
        "absolute_position": [10000.0,22000.0]},
      ...},
    "SECTIONS": {
      "uz2_0_uz2_1": {
```

```

        "id": "uz2_0_uz2_1",
        "upstream": "uz2_0",
        "downstream": "uz2_1",
        "length": 500,
        "zone": "RES_1-1"},
        ...},
    "ZONES": {
        "RES_0-0": {
            "id": "RES_0-0",
            "sections": ["uz1_0_uz1_1", "uz1_0_uz1_15", ...],
            "contour": [[-1.0, -1.0], [9999.0, -1.0], ...]},
        ...},
    },
    "LAYERS": [
        {"ID": "CAR",
         "TYPE": "mnms.graph.layers.SimpleLayer",
         "VEH_TYPE": "mnms.vehicles.veh_type.Car",
         "DEFAULT_SPEED": 11.5,
         "SERVICES": [...],
         "NODES": [...],
         "LINKS": [...],
         "MAP_ROADDB": {...}},
        ...],
    "TRANSIT": [
        {"ID": "CAR_uz2_164_Bus_NS6_Bus_NS6_4",
         "UPSTREAM": "CAR_uz2_164",
         "DOWNSTREAM": "Bus_NS6_Bus_NS6_4",
         "LENGTH": 0.0,
         "COSTS": {"WALK": {"length": 0.0}},
         "LABEL": "TRANSIT"},
        ...]
}

```

2.3 Dynamic space sharing

The [DynamicSpaceSharing](#) module allows to ban some links for the vehicles of some mobility services for some periods. To use it, one should use the `set_dynamic` method of the multi layer graph's `dynamic_space_sharing`. This method takes as argument a [Callable](#) which takes as inputs the multi layer graph and the current time. It should returns a list of tuples, where the first element of the tuple corresponds to the banned link id, the second element to the name of the mobility service concerned by the banning, and the third element to the number of flow time steps periods during which the banning should remain active. The functional tests located in [MnMS/tests/graph/test_dynamic_space_sharing](#) provide examples on how to use this feature. Notably, the [TestDynamicSpaceSharing](#) class shows how to define the [RoadDescriptor](#) object representing a street with a normal lane and a bus lane. The normal lane can be used by on vehicles at any time, while the bus lane can be used by non public transportation vehicles in certain

periods only.

NB: The banning leads to the rerouting of the vehicles activities concerned (i.e. that initially were supposed to pass through at least one of the banned links) while the end of a banning period does not lead to any rerouting.

3 Demand management

In this section, we describe how demand is managed within the simulation.

3.1 Demand data

3.1.1 Users list

Demand can be defined in two different ways.

- **BaseDemandManager**: takes a list of **User** objects. A user is minimally defined by an id, an origin coordinate or node, a destination coordinate or node, and a departure time. Optional attributes include: the list of available mobility services (`available_mobility_services`), the id of the mobility services graph (`mobility_services_graph`), the path this user should follow (`path`) and the chosen mobility services on this path (`forced_path_chosen_mobility_services`), the maximum duration user is ready to wait an answer from a service (`response_dt`), the maximum duration user is ready to wait for being picked up (from the moment user is matched, `pickup_dt`). Any other parameters characterizing the user can be specified with `user_parameters` as a **Callable** taking the user as input and returning a **Dict** of parameters.
- **CSVDemandManager**: this is the recommended way of defining large demand scenarios. It takes a string with the path to a .csv file specifying the demand. The .csv file should satisfy the following requirements:
 - be delimited by semicolons
 - define one user per line
 - have a header with the first four columns correspond to the mandatory fields in this order: **ID**, **DEPARTURE**, **ORIGIN**, **DESTINATION** for user id, departure time, origin coordinate or node, destination coordinate or node
 - can have optional columns correctly named:
 - * **MOBILITY SERVICES** defines the mobility services available to this user (formatted as one string with mobility services names separated by one blank space)
 - * **MOBILITY SERVICES GRAPH** defines the mobility services graph to apply for this user. Only the graph id is provided here, the whole graph is defined in a .json file and are loaded in the decision model.
 - * **PATH** defines the path (list of nodes) user will be forced to follow initially
 - * **CHOSEN SERVICES** defines the mobility service chosen per layer for the forced initial path
 - * NB1: either **MOBILITY SERVICES** or **MOBILITY SERVICES GRAPH** should be defined, not both

* NB2: either **PATH** and **CHOSEN SERVICES** should be both defined or both absent

See Figure 3 and 4 for examples.

ID	DEPARTURE	ORIGIN	DESTINATION	MOBILITY SERVICES
U0	07:00:00.00	11000.0 20000.0	12500.0 17500.0	CAR METRO BUS RIDEHAILING TRAIN
U1	07:00:01.00	8000.0 22000.0	2000.0 14000.0	CAR METRO BUS RIDEHAILING TRAIN
U2	07:00:03.00	12000.0 12500.0	26000.0 30000.0	METRO BUS RIDEHAILING TRAIN
U3	07:00:03.00	4000.0 14000.0	14500.0 14500.0	METRO BUS RIDEHAILING TRAIN
U4	07:00:04.00	24000.0 10000.0	26000.0 18000.0	CAR METRO BUS RIDEHAILING TRAIN
U5	07:00:04.00	6000.0 10000.0	22000.0 21000.0	CAR METRO BUS RIDEHAILING TRAIN

Figure 3: Example of CSV demand data with coordinates to locate origin and destination and a **MOBILITY SERVICES** optional column.

ID	DEPARTURE	ORIGIN	DESTINATION	MOBILITY SERVICES GRAPH
U0	07:00:00.00	10000.0 13000.0	28000.0 16000.0	G1
U1	07:00:04.00	17000.0 16500.0	28000.0 4000.0	G2
U2	07:00:05.00	20000.0 0.0	12000.0 6000.0	G2
U3	07:00:05.00	16000.0 18000.0	2000.0 8000.0	G1
U4	07:00:06.00	12000.0 8000.0	2000.0 24000.0	G1
U5	07:00:07.00	17000.0 20000.0	12000.0 4000.0	G1

Figure 4: Example of CSV demand data with coordinates to locate origin and destination and a **MOBILITY SERVICES GRAPH** optional column.

3.1.2 Mobility services graphs

If one wants to use the mobility services graphs, they should be loaded from a .json file in the `AbstractDecisionModel` object with the `AbstractDecisionModel.load_mobility_services_graphs_from_file` method. The .json corresponds to a nested dictionary where:

- the first level of keys corresponds to the ids of the different graphs as referred to in the demand .csv file in the **MOBILITY SERVICES GRAPH** column

- the second level corresponds to the list of currently available mobility services, the key is a string with the mobility services ids separated with a blank space, the ids can be written in any order, the string "None" refers to the initial empty list at user's departure
- the third level corresponds to the event that triggered the need for (re)planning: "DEPARTURE" for user's departure, the name of the mobility service which refused user for a match failure, the name of the custom event.

See Figure 5 and Code 1 for a concrete example.

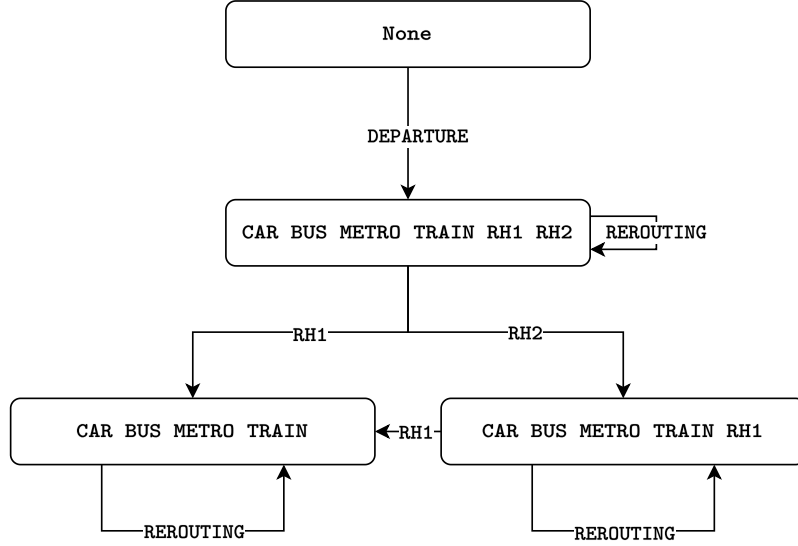


Figure 5: Example of a mobility services graph. Note that we have considered a custom event called **REROUTING**.

```

1  {
2    "G1": {
3      "None": {
4        "DEPARTURE": ["CAR", "BUS", "METRO", "TRAIN", "RH1", "RH2"]
5      },
6      "CAR BUS METRO TRAIN RH1 RH2": {
7        "RH1": ["CAR", "BUS", "METRO", "TRAIN"],
8        "RH2": ["CAR", "BUS", "METRO", "TRAIN", "RH1"],
9        "REROUTING": ["CAR", "BUS", "METRO", "TRAIN", "RH1", "RH2"]
10     },
11     "CAR BUS METRO TRAIN RH1": {
12       "RH1": ["CAR", "BUS", "METRO", "TRAIN"],
13       "REROUTING": ["CAR", "BUS", "METRO", "TRAIN", "RH1"]
14     },
15     "CAR BUS METRO TRAIN": {
16       "REROUTING": ["CAR", "BUS", "METRO", "TRAIN"]
17     }
18   }
19 }

```

Code 1: JSON file corresponding to mobility services graph represented in Figure 5.

3.2 User states

A user can be in different states along the simulation:

- **ARRIVED**: user has reached her destination and won't move again
- **WAITING_ANSWER**: user has placed a request and is waiting for a match proposition
- **WAITING_VEHICLE**: user has received and accepted a match proposition, she now waits for the vehicle
- **WALKING**: user walks on a transit link
- **INSIDE_VEHICLE**: user is riding a vehicle
- **STOP**: user has just finished walking or riding a vehicle, this is an intermediate state before walking again or requesting a vehicle
- **DEADEND**: user is stopped at a node which is not her destination and has no longer option to finish her journey

Figure 6 represents these states and the events leading to transition from one state to another.

3.3 Planning (path allocation)

Planning refers to user's mode and route choice. The planning module is called **AbstractDecisionModel** in MnMS.

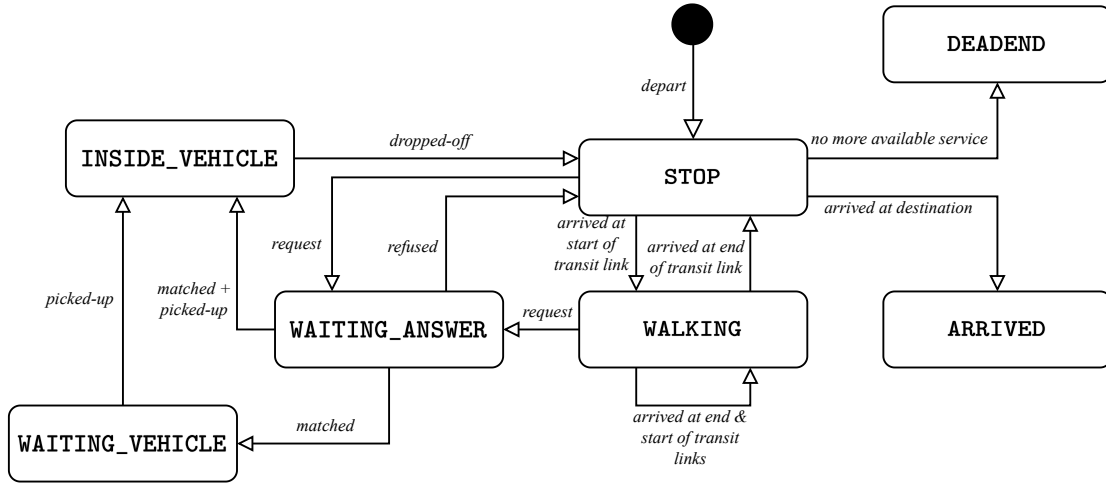


Figure 6: User states transitions.

3.3.1 Scheduling

The planning module is called in the beginning of every flow time step for all users that have undergone an event which triggered the need for (re)planning. The list `AbstractDecisionModel._users_for_planning` gathers the tuples `(user, event)`, where `user` should (re)plan following `event`.

3.3.2 Events leading to (re)planning

A traveler is added to this list of users who need to (re)plan on different types of events:

- **DEPARTURE**: user has just departed from her origin and requires an initial planning
- **MATCH_FAILURE**: user has received no matching proposition for the requested mobility service for `response_dt`, or when all matching proposed to the user during `response_dt` had an estimated pick-up time over `pickup_dt`. Note that the default `response_dt` is **2 minutes** for all users and all mobility services. The default `pickup_dt` is **5 minutes** for all users and all mobility services except services of type `PublicTransportationMobilityServices` for which `pickup_dt` is **24 hours**.
- **INTERRUPTION**: for now, this event is triggered for a user who is or should pass through a link of the `TransitLayer` leading to a free-floating station which was removed because there is no more vehicle there. One can trigger this event under other conditions by editing MnMS source code.

Note that user is removed from the `AbstractDecisionModel._users_for_planning` list when she realized the planning.

3.3.3 Planning origin

User's planning origin is:

- her current position if she is in state `STOP`, `WAITING_ANSWER`, or `WAITING_VEHICLE`, if she is in state `WALKING` and currently walking on a link that has been deleted
- her next node (the downstream node of her current link) otherwise (user is in state `WALKING` or `INSIDE_VEHICLE` since user cannot replan in state `DEADEND` or `ARRIVED`)

3.3.4 Update available mobility services

Following an event and just before searching the shortest path(s), the available mobility services of the user are updated as follows:

- a `DEPARTURE` event leads to:
 - no modification of the available mobility services list if the list is not empty and user has no mobility services graph defined
 - the initialization of this list with the value associated with `None` → `DEPARTURE` transition in the mobility services graph of the user if this graph is defined and the transition can be found
 - the initialization of this list with all mobility services defined in the simulation if (the list is empty and user has no mobility services graph defined) OR (the transition `None` → `DEPARTURE` cannot be found in user's mobility services graph)
- a `MATCH_FAILURE` event leads to:
 - the removal of the mobility service which refused the user from the list of available mobility services if the graph is not defined OR the "current available mobility services → name of refusing mobility service" transition is not present in the graph
 - applying the "current available mobility services → name of the refusing mobility service" transition present in the graph otherwise
- another (custom) event to:
 - no modification of the available mobility services list if user's graph is not defined OR the "current available mobility services → event name" transition is not present in the graph
 - applying the "current available mobility services → event name" transition present in the graph otherwise
- After these modifications, the case of the services of type `PersonalMobilityService` is treated. For each user who replans, i.e. who has undergone an event which is not `DEPARTURE`, for each service of type `PersonalMobilityService` currently available for this user:
 - If user is in `STOP`, `WAITING_ANSWER`, or `WAITING_VEHICLE` state, we check the distance between user's current node and the position of user's personal vehicle (which is user's origin if user has not already used it, and the position where it has been parked otherwise - saved in `User._parked_personal_vehicles`). If this distance is:
 - * **higher** than `AbstractDecisionModel.personal_mob_service_park_radius`, then this service is removed from user's list of available mobility services

- * **lower** than `AbstractDecisionModel.personal_mob_service_park_radius`, then the service is kept in user's list of available mobility services, and the parking location becomes user's new planning origin for all mobility services combinations containing the service. If user ends by choosing a path starting at this new planning origin, she is "teleported" there when the path starts (see Section 3.3.9).
- If user is in **WALKING** state, we check the distance between user's current position and the position of user's personal vehicle and proceed similarly when it is higher/lower than `AbstractDecisionModel.personal_mob_service_park_radius`.
- If user is in **INSIDE_VEHICLE** state, we check if user is currently within a vehicle of the service. If so, the service is kept available. If not, the service is removed from user's list of available mobility services
- **NB1**: The default value for `AbstractDecisionModel.personal_mob_service_park_radius` is **100 meters**. It should be kept small to limit the teleportation distance.
- **NB2**: If a mode combination contains several personal mobility services parked within `AbstractDecisionModel.personal_mob_service_park_radius` at different nodes, we consider arbitrarily one of them and a warning is printed in the log file.
- **NB3**: If one wants to consider personal mobility services the user can put in her pocket such as scooter, folding bike, etc., a new type of mobility service (e.g. **PocketPersonalMobilityService**) should be created to differentiate them from personal mobility services based on vehicles that need to be parked. The review done on the availability of **PersonalMobilityService** should not be done on **PocketPersonalMobilityService**.
- Finally, if we detect that user has no more available mobility services, her state is set to **DEADEND** and she is removed from the list of users who need to (re)plan.

3.3.5 Discovering shortest paths

There are two ways of discovering the user's shortest paths: a **default discovery** that require no additional input, a **guided discovery** that require to define the optional argument `considered_modes` at instantiation of the decision model. In the case of the guided discovery, note that as the definition of layers is linked with the modes considered by travelers, the list of considered modes is defined for all users at the level of the decision model. Let us call:

- $L = \{l_i\}_i$ a subgraph of the **MultiLayerGraph** that groups the layers $\{l_i\}_i$
- $M_i = \{m_{i,j}\}_j$ the set of mobility services operating on layer l_i
- M^u the set of mobility services available for user u

The shortest paths discovery is realized as follows:

- **Default discovery**: L corresponds to the whole graph in this discovery mode. We compute k shortest paths on L for each mobility services combination choice of the user. In total, there are $\prod_i |M_i \cap M^u|$ mobility services combinations.

Note that k has **3** as default value, it can be changed by setting argument `n_shortest_path` at the decision model instantiation.

The shortest paths computation is done in parallel for all users - mobility services combinations to maintain the simulation performances.

- **Guided discovery:** it requires to define `considered_modes` as a list of modes that should be considered by users. In this discovery mode, L can be any set of layers. A mode can either be:

- **not necessarily intermodal**, then represented by a tuple (L, None, k) . We compute k shortest paths on L for each mobility services combination choice of the user.
- **necessarily intermodal**, then represented by a tuple $(L, (L', L''), k)$ where $L' \subset L$, $L'' \subset L$, $L' \cap L'' = \emptyset$, L' and L'' define the sets of layers between which intermodality is mandatory. We compute k shortest paths on L for each mobility services combination choice of the user with a mandatory passage through one link in L' and one link in L'' . Several transfers between L' and L'' is not forbidden but not mandatory: looking for shortest paths PT + RideHailing + PT is not supported for example.

The shortest paths computation is done sequentially for each considered mode. For one mode, the shortest paths computation is done in parallel for all users to maintain the simulation performances.

The modes are treated in the order of the list, the shortest paths found as we go through the modes are saved. Before proceeding to the discovery for one mode, we check in the saved routes if we already have gathered the proper number of routes for each mobility services combination of this mode. If so, we pass to the next mode without proceeding to the discovery. If not, we proceed to the discovery.

The quality of the shortest paths obtained is checked with two conditions:

- the path should contain at least two nodes
- the path should not pass through a node more than once

See Table 2 for explicit examples.

Layers: MSs	CarL: • PCarMS • UberMS • LyftMS	CarL: • PCarMS • UberMS • LyftMS	PCarL: • PCarMS RHailingL: • UberMS • LyftMS
Considered modes	None default discovery	None default discovery	None default discovery
User available MSs	All	PCarMS UberMS	All
Shortest paths computed (Layer: MS chosen: nb paths)	• CarL: PCarMS: k • CarL: UberMS: k • CarL: LyftMS: k	• CarL: PCarMS: k • CarL: UberMS: k	• PCarL \cup RHailingL: PCarMS and UberMS: k • PCarL \cup RHailingL: PCarMS and LyftMS: k
Number of shortest paths computed for this user	$3*k$	$2*k$	$2*k$

Layers: MSs	PCarL: • PCarMS RHailingL: • UberMS • LyftMS	PCarL: • PCarMS RHailingL: • UberMS • LyftMS	PCarL: • PCarMS RHailingL: • UberMS • LyftMS
Considered modes	• ($\{PCarL\}, None, k_1$) • ($\{RHailingL\}, None, k_2$) • ($\{PCarL, RHailingL\},$ $(\{PCarL\}, \{RHailingL\}), k_3$)	• ($\{PCarL\}, None, k_1$) • ($\{RHailingL\}, None, k_2$) • ($\{PCarL, RHailingL\},$ $(\{PCarL\}, \{RHailingL\}), k_3$)	• ($\{PCarL\}, None, k_1$) • ($\{RHailingL\}, None, k_2$) • ($\{PCarL, RHailingL\},$ $(\{PCarL\}, \{RHailingL\}), k_3$)
User available MSs	All	PCarMS UberMS	UberMS
Shortest paths computed (Layer: MS chosen: nb paths)	• PCarL: PCarMS: k_1 • RHailingL: UberMS: k_2 • RHailingL: LyftMS: k_2 • PCarL \cup^* RHailingL: PCarMS and UberMS: k_3 • PCarL \cup^* RHailingL: PCarMS and LyftMS: k_3	• PCarL: PCarMS: k_1 • RHailingL: UberMS: k_2 • PCarL \cup^* RHailingL: PCarMS and UberMS: k_3	• RHailingL: UberMS : k_2
Number of shortest path computed for this user	$k_1 + 2 * k_2 + 2 * k_3$	$k_1 + k_2 + k_3$	k_2

Table 2: Examples of considered modes definition. MS = Mobility Service, L = Layer, \cup^* = union with mandatory intermodality

3.3.6 Shortest paths computation

The computation of k shortest paths for the origins - destinations - mobility services combination is done by calling the `parallel_k_shortest_path` function for non necessarily intermodal paths, and the `parallel_k_intermodal_shortest_path` function for necessarily intermodal paths. These are C++ functions from HiPOP wrapped in Python. On top of origins, destinations, mobility services combinations, subgraphs, cost name, numbers of paths to compute, they require the following

parameters:

- `max_diff_cost`: the maximum relative difference between the cost of the first shortest path found and the costs of the other paths to find. Default value is `0.25` meaning that the cost of the $k - 1$ other paths found should be less than 125% of the cost of the first path found.
- `max_dist_in_common`: the maximal relative distance in common between the first shortest path found and the others. Default value is `0.95` meaning that the $k - 1$ other paths found should have less than 95% common distance with the first path found.
- `cost_multiplier_to_find_k_paths`: the multiplier applied to the links costs of an accepted shortest path to find other ones. Default value is `10`.
- `max_retry_to_find_k_paths`: the maximum number of times we retry to find an acceptable shortest path in HiPOP. Default value is `50`.
- `thread_number`: the number of threads to launch in parallel. It is set to the number of CPUs of the machine where code runs by default.

NB: these parameters can be calibrated on the studied scenario. The warning `User - found only -/- shortest paths for modes combinations` - printed in the log file indicates that less paths than requested were found by HiPOP. One can try to increase `max_diff_cost`, `max_dist_in_common`, `max_retry_to_find_k_paths`, or change `cost_multiplier_to_find_k_paths` to reduce the number of times this warning is triggered.

3.3.7 Travel cost estimation

By default in MnMS, the travel cost of a path correspond to its instantaneous travel time. The travel time estimation is achieved in two stages:

- the first stage takes place in HiPOP when we call the `parallel_k_shortest_path` or the `parallel_k_intermodal_shortest_path` function. The first stage travel time is the sum of the instantaneous in-vehicle or walking travel times on each link composing the path. The first stage travel time is taken into account for the shortest paths discovery. Note that in this stage, the waiting time for getting a vehicle from any mobility service is considered null. Consequently, the waiting times do not impact the shortest paths discovery. Note also that the in-vehicle and walking travel times on each link of the `MultiLayerGraph` object are dynamically updated by the `FlowMotor` every affectation time step. The computation of shortest paths takes into account instantaneous travel times, i.e. travel times on links at the moment when the planning is done (there is no anticipation of speed variation).
- the second stage takes place outside HiPOP for each shortest path found. The second stage travel time sums the first stage travel time and the total estimated waiting time. The total estimated waiting time is obtained by consulting each mobility service used in the path via their `estimate_pickup_time_for_planning` method. See Section 5 for more details on the pickup time estimation depending on the mobility service and its dispatching strategy. The second stage travel time is used as a basis to compare paths and select one of them according to the decision model. It can be seen as the path utility.

One can define another travel cost than the travel time and let users select their path on the basis of this new travel cost. Several steps are required to do so:

- One should add the **cost functions for the first stage cost computation**. A cost function is defined on a certain layer for a certain mobility service. To add it, one can use the `add_cost_function` method of the `MultiLayerGraph` class. It requires the layer id, the name of the new cost, eventually the name of the mobility service it applies to (if no mobility service is specified, the cost function applies to all services of the layer), and a `Callable` taking the `MultiLayerGraph` object, the link id, and the dictionary of this link's official costs as input. Links official costs are: `length`, `speed`, and `travel_time`. The cost function should return a positive float because a non-negativity constraint holds on the links costs in Dijkstra algorithm.
- One should add a **waiting cost function for the second stage cost computation**. A waiting cost function is defined in the decision model. To add it, one can use the `add_waiting_cost_function` method of the decision model. It requires the name of the new cost, and a `Callable` taking the total estimated waiting time as input. The sum of the first stage travel cost and the second stage waiting cost is considered to be the path utility.
- One can eventually add an **additional cost function** to increment the new cost value with terms that cannot be taken into account in the first nor the second stage computation. An additional cost function is defined in the decision model. To add it, one can use the `add_additional_cost_function` method of the decision model. It requires the name of the new cost, and a `Callable` taking the `Path` and the `User` objects as input. The former object notably contains the list of nodes and the path cost after the second stage computation. The latter object notably contains the user's achieved path until now.
- Finally, one should specify that the new cost should be used to select users paths by filling in `cost` argument of the decision model object.

✏ Taking into account the waiting times after the shortest paths discovery may lead to sub optimal shortest paths. Figure 7 provides an example of this with an on-demand service providing service on two zones. One zone benefits from a high level of service, the other from a low level of service. As it is specified above, the shortest path discovery may lead to route 1 which requests the on-demand service in the undersupplied zone whereas route 2 would have a lower travel time. Indeed, route 2 starts by walking till the oversupplied zone, then requests the service to join the destination.

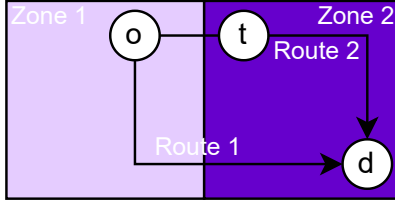


Figure 7: Example of suboptimal path when waiting times are taken into account after the shortest path discovery. The darkest violet represents a higher level of service, i.e. a lower estimated pickup time.

3.3.8 Path selection

Once the different alternatives for the user have been gathered and their travel cost evaluated, the selection of a route is done by applying the `AbstractDecisionModel.path_choice` method. For now, there are three decision models available in MnMS:

- **DummyDecisionModel**: it is a deterministic model where user choose the path with the lowest travel cost
- **LogitDecisionModel**: it is a stochastic model where the probability of choosing one path depends on the travel costs on all possible paths
- **ModeCentricLogitDecisionModel**: fit is a stochastic model where for one mode, the best path is chosen deterministically, then a logit model is called on the best path of each mode

NB1: If one wants to use a stochastic model and be able to reproduce the results, the `seed` optional argument of the `Supervisor.run` method can be provided.

NB2: If no valid path was found for a user, she turns into **DEADEND** state...

- after finishing to walk on her current link if she is in **WALKING** state and her current link has not been deleted
- after canceling her ongoing request if she is in **WAITING-ANSWER** state
- after canceling her ongoing match if she is in **WAITING-VEHICLE** state
- after finishing riding the vehicle on her current link and leave the vehicle there if she is in **INSIDE-VEHICLE** state


3.3.9 Start of path

The planning execution is generic: it deals with users that have just departed from their origin, users that have been refused, users that are currently following their path. The effect of the (re)planning depends on user's current state. If user's state is:

- **STOP**: user starts the chosen path instantaneously, user's current node should corresponds to the chosen path first node, OR the distance between user's current node and the chosen path first node should be lower than `AbstractDecisionModel.personal_mob_service_park_radius`. In the latter case, user is "teleported" at chosen path first node.

- **WALKING**: if user is walking on a link that have been deleted or if the distance between user's current position and the chosen path first node is lower than `AbstractDecisionModel.personal_mob_service_park_radius`, user starts the chosen path instantaneously by being "teleported" at chosen path first node. Otherwise, user finishes to walk the current link and her path is updated by concatenating current node and chosen path which should start at the downstream node of the current link.
- **WAITING_ANSWER**: user starts the chosen path instantaneously, user's current node should corresponds to the chosen path first node, OR the distance between user's current node and the chosen path first node should be lower than `AbstractDecisionModel.personal_mob_service_park_radius`. In the latter case, user is "teleported" at chosen path first node. User's path is updated and her ongoing request is cleaned or updated consequently.
- **WAITING_VEHICLE**: user starts the chosen path instantaneously, user's current node should corresponds to the chosen path first node, OR the distance between user's current node and the chosen path first node should be lower than `AbstractDecisionModel.personal_mob_service_park_radius`. In the latter case, user is "teleported" at chosen path first node. User's path is updated and the plan of the vehicle she was matched is updated consequently.
- **INSIDE_VEHICLE**: user's path is updated by concatenating current node and chosen path which should start at the downstream node of the current link so that user finishes to travel the vehicle on the current link and then starts her new path. Vehicle plan is updated consequently.

NB: For now, there exist only two strategies to update the plan of a vehicle.

- For `PublicTransportationMobilityService` vehicles, the plan is updated according to the line definition. The order in which line stops are visited is fixed.
 - For vehicles of other mobility service types, when a user cancels her ride or modifies her drop node, the order in which customers are served in the plan is maintained.
-  Other strategies could be added in the future, for e.g. we could optimize the plan for on-demand vehicles, allowing the vehicle to change the order in which customers are served, with regard to specified objective and constraints.

3.4 Outputs

Some information about the planing can be written in a .csv file. To enable this paths writing, there exists two arguments to pass to the `AbstractDecisionModel` constructor:

- `outfile`: if passed to the constructor, the chosen paths per user per planing phase are written in the file specified
- `verbose_file`: if `True` and `outfile` specified, all the computed shortest paths are written, not only the ones chosen

The outputs file includes the following columns: `ID` (id of the user), `EVENT` (event which triggered the need for replanning), `TIME` (time at which event was triggered), `COST` (cost of the path), `PATH` (list of links of the path), `LENGTH` (total distance of the path), `SERVICES` (list of mobility services

used in this path), CHOSEN (boolean specifying if this path has been chosen by user or not).

Additionally, a `UserObserver` object can be attached to the demand manager with the `add_user_observer` method to follow the activity of users. It outputs a .csv file with the following columns: TIME (current time), ID (id of the user), LINK (link where the user is), POSITION (coordinates of the user), DISTANCE (distance traveled by user so far), STATE (state of the user), VEHICLE (vehicle in which the user is).

3.5 Warnings

The demand management module can lead to several warning alerts in the log file of the simulation. In this section we document these warning messages.

- `User - already undergone an event triggering a (re)planning, ignore new event -:` triggered when a user has undergone several events leading to the need for (re)planning during the same flow time step.
- `Cannot find transition None->DEPARTURE in - mobility services events graph, all services available:` triggered when a mobility services graph does not contain the None - DEPARTURE transition, default behavior is applied, i.e. user has access to all mobility services defined in the scenario.
- `Cannot find transition - in - mobility services events graph, - removed:` triggered when a mobility services graph does not contain the proper transition after a `MATCH_FAILURE` event, default behavior is applied, i.e. the service which declined user's requests is removed from the list of user's available mobility services.
- `Cannot find transition - in - mobility services events graph, list unchanged:` triggered when a mobility services graph does not contain the proper transition after an event which is not `DEPARTURE` nor `MATCH_FAILURE`, default behavior is applied, i.e. no modification of user's list of available mobility services.
- `- has no more available mobility service to continue her path:` triggered when a user turns into `DEADEND` state because has access to no more mobility service.
- `User - was walking on link - when this link got deleted, user will look for an alternative path from upstream node of this link...:` triggered when user undergone an `INTERRUPTION` event due to the deletion of the link she is currently traveling on.
- `User - have several personal mob services available with different planning origins check what to do, for now we take the first arbitrarily ! (user intersection = -):` triggered when user looks for a path on a subgraph with several personal mobility services associated with different planning origins (personal vehicles parking locations), the first mobility service and its associated planning origin is chosen arbitrarily.
- `User - has found no path during the (re)planning, her current TRANSIT link no longer exist, turns DEADEND immediately.:` triggered when user turns to `DEADEND` state because found no path during the (re)planning after an `INTERRUPTION` event as she was walking on a link now deleted.

- User - has found no path during the (re)planning, will finish to walk the current TRANSIT link and turn DEADEND: triggered when user turns to DEADEND state because found no path during the (re)planing after an INTERRUPTION event as she was walking.
- User - has found no path during the (re)planning, cancels ongoing request and turns to DEADEND.: triggered when user turns to DEADEND state because found no path during the (re)planing after an INTERRUPTION event in WAITING_ANSWER state.
- User - has found no path during the (re)planning, cancels ongoing match and turns to DEADEND.: triggered when user turns to DEADEND state because found no path during the (re)planing after an INTERRUPTION event in WAITING_VEHICLE state.
- User - has found no path during the (re)planning, will alight vehicle - at next node and turn DEADEND.: triggered when user turns to DEADEND state because found no path during the (re)planing after an INTERRUPTION event in INSIDE_VEHICLE state.
- User - has found no path during the (re)planning, turns to DEADEND.: triggered when user turns to DEADEND state because found no path during the (re)planing after a DEPARTURE or MATCH_FAILURE event or INTERRUPTION event in state STOP.
- User - found only -/- shortest paths for modes combinations -: indicates that less paths than requested were found by HiPOP.
- One shortest path computed for user - is not valid because contains several occurrences of the same node: -: triggered when user has found one path which passess through the same node several times, this path is not taken into account.
- One shortest path computed for user - is not valid because contains less than 2 nodes: -: triggered when user has found a path with no node or only one node, this path is not taken into account.
- Zero path computed for user - for mode combination -, -: triggered when no path was found by HiPOP for one user - mode combination.

4 Vehicles motion management

MnMS is based on a macroscopic traffic modeling framework to compute the speeds in different regions of the network, but vehicles are individually moved on the roads. In this section, we describe how vehicles are modeled and moved.


4.1 Vehicle types

A **Vehicle** object is characterized by its location on the graph or node, its capacity, the mobility service it belongs to, and a boolean indicating if it is personal or not. Six types of vehicles are defined for now in MnMS: **Car** and **Bike** which can be personal, **Bus**, **Tram**, **Metro**, **Train** which cannot be personal. Note that here is not default value for the capacity of one type of vehicle. Note also that all vehicle types have the same methods, but one can easily add new or modify existing methods for one specific vehicle type.

4.2 Vehicle activities

A **Vehicle** object has no state, but a plan, which is a list of activities the vehicle should achieve. A **VehicleActivity** is characterized by a destination node on the graph, a path to follow to reach this destination, eventually a user concerned, a boolean specifying if it leads to a vehicle movement or not. There exist four types of vehicle activities in MnMS for now:

- One static activity:
 - **VehicleActivityStop**: vehicle is stopped
- And three moving activities:
 - **VehicleActivityRepositioning**: vehicle is empty and repositioning toward a certain node of the graph
 - **VehicleActivityPickup**: vehicle is joining the desired pickup point of a user
 - **VehicleActivityServing**: vehicle is joining the desired drop-off point of one of its passengers

 New activity types could be defined, for example a charging activity for electric vehicles, or a waiting passenger activity for vehicles belonging to mobility services where user can book in advance.

4.3 Traffic dynamics

The traffic dynamics are handled either by the **MFDFlowMotor** or the **CongestedMFDFlowMotor**. These modules inherit from the **AbstractMFDFlowMotor** class which allows to compute the traffic flow speed according to the accumulations of vehicles within defined reservoirs and moves vehicles on the network accordingly.

4.3.1 Speed computation in a reservoir

The mean speed v_m^r of every vehicle type m for each reservoir r is computed with the Macroscopic Fundamental Diagram framework (MFD) [1, 2] to represent the dynamics of the congestion. More specifically, MnMs is based on the trip-based variant [5, 3] to account for different trip lengths. Each reservoir is characterized by an MFD speed function which links the number of vehicles currently moving in this reservoir for each vehicle type to the mean speed of each vehicle type. At time t , the mean speed $v_m(t)$ of vehicle type m is

$$v_m^r(t) = V_m^r(\{n_{m'}(t) \forall m' \in \mathcal{M}\}), \quad (1)$$

with $n_{m'}^r(t)$ the number of vehicles of type m' currently driving in reservoir r and \mathcal{M} the set of vehicle type considered in the simulation. $\{n_{m'} \forall m' \in \mathcal{M}\} \mapsto \{V_m^r \forall m' \in \mathcal{M}\}$ is the multi-modal MFD speed function describing the congestion dynamics and the interactions between the modes in region r .

4.3.2 Vehicle motion

At the start of an activity, the vehicle gets initiated with a distance to travel in one or several reservoirs. For each time step (dt_{flow}), all circulating vehicles move by a distance equal to the time step multiplied by the corresponding speed. They change the reservoir or reach the destination of the ongoing activity once the remaining travel distance reaches zero. Then, the activity is considered to be done, and the next activity in plan starts. If there is no more activity in vehicle's plan, a dummy `VehicleActivityStop` at current node is created.

4.3.3 Queues between reservoirs

While the `MFDFlowMotor` models intra reservoirs congestion only, the `CongestedMFDFlowMotor` also models inter reservoirs congestion:

- In the version without spillover, a vehicle traveling through different reservoirs directly appears in the following reservoir once it completes its travel in the former reservoir on its itinerary. Issues may arise when a vehicle travels from an uncongested reservoir to a congested one, as (i) we would expect the vehicle to undergo delays when crossing the boundary between the reservoirs as the congestion spillover adjacent reservoir and (ii) the congested reservoir may tend dangerously close to gridlock, which is rarely reached in real observations.
- In the version with spillover, one should define an input function to specify the maximum flow allowed to enter the reservoir according to the current accumulation. If the demand flow is higher, incoming vehicles are stored in a queue and do not appear in either the origin or the destination reservoirs. The entry time of the next vehicle in the queue is updated accordingly. To represent the spillover, the MFD of the origin reservoir is scaled down to have the maximum accumulation equal to the maximum accumulation without queue n_{max} minus the queue accumulation n_{queue} to represent the capacity drop [6]. It means the speed MFD is then $n \mapsto V \left(n \frac{n_{\text{max}}}{n_{\text{max}} - n_{\text{queue}}} \right)$.

 The `CongestedMFDFlowMotor` only works for monomodal reservoirs for now.

4.4 Outputs

Some information about the traffic dynamics can be written in a .csv file. To enable this, the argument `outfile` should be specified when the flow motor object is created. The output file includes the following columns: `AFFECTATION_STEP` (the affectation step number), `FLOW_STEP` (the flow step number), `TIME` (current time), `RESERVOIR` (id of the reservoir), `VEHICLE_TYPE` (vehicle type concerned), `SPEED` (speed of this vehicle type in this reservoir), `ACCUMULATION` (number of vehicles of this type in this reservoir).

5 Mobility services

5.1 Abstract classes

All mobility services inherit from the `AbstractMobilityService` class. A mobility service is minimally characterized by its ID, the layer on which it runs, the capacity of the vehicles composing

its fleet, the frequency of matching phase and periodic maintenance. The behavior of a mobility service always follows the flow chart represented in the upper part of Figure 8. The lower part of the Figure shows the interface of a mobility service. It minimally contains the `estimate_pickup_time_for_planning` method which provides an estimation of the pick up time at a certain node, useful for the user's (re)planning.

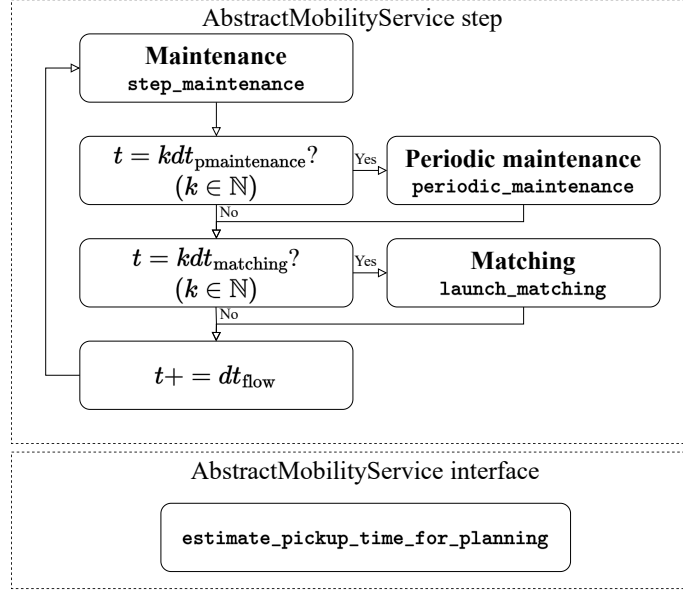


Figure 8: `AbstractMobilityService` flow chart and main interfaces.

Depending on the mobility service type, additional parameters can be defined, the content of the maintenance, periodic maintenance, matching phases, and pickup time estimation can be different, and additional steps can be added to the flow. Sub abstract classes inheriting from the `AbstractMobilityService` class have been defined to group mobility service types with common features. Figure 9 represents the inheritance relationships between the different mobility services classes. For now we have three sub abstract classes and six operational mobility services.

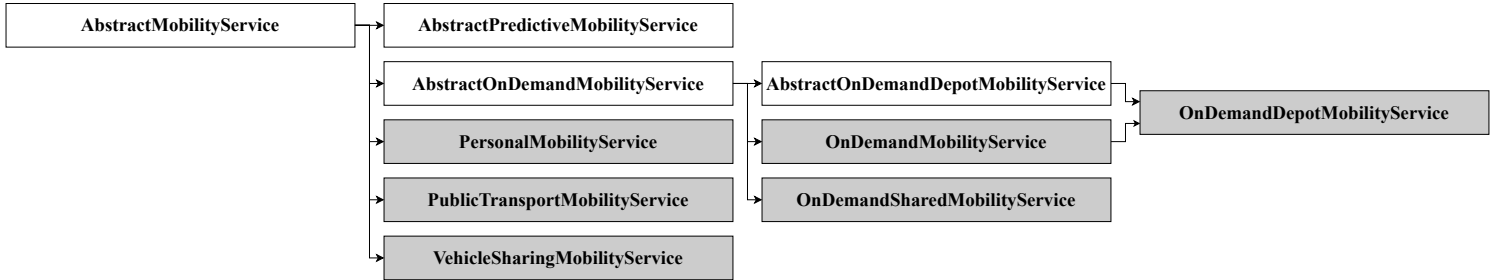


Figure 9: Inheritance relationships for the different mobility service classes. White bricks correspond to abstract classes and gray bricks to non abstract classes.

The three sub abstract classes are the following:

- **AbstractPredictiveMobilityService**: This class applies to mobility services which have an horizon of prediction for the upcoming demand. In this class, the flow chart is slightly modified to include a rebalancing step realized based on the prediction horizon (see Figure 10).
- **AbstractOnDemandMobilityService**: The class applies to all types of on demand mobility services. It provides two additional interface functions:
 - `create_waiting_vehicle` to initialize the on demand vehicles on the layer the service runs on.
 - `add_zoning` to initialize a zoning for the operation of the service.
- **AbstractOnDemandDepotMobilityService**: The class applies to all types of on demand mobility services with depots for their vehicles. On top of `create_waiting_vehicle` and `add_zoning` functions, it also provides a `add_depot` function to define the depots parameters (location, capacity).

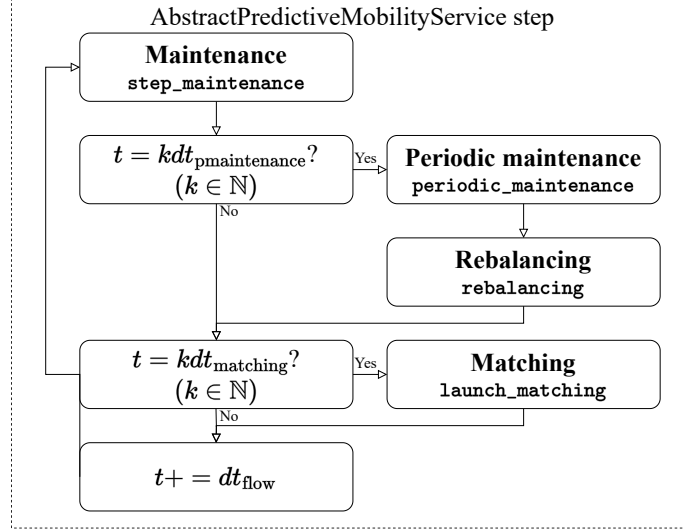


Figure 10: **AbstractPredictiveMobilityService** flow chart.

In the following, we describe the six operational mobility services in detail.

5.2 Personal mobility service

The **PersonalMobilityService** is the simplest mobility service. It corresponds to the personal vehicles. It has no more parameters and interface methods than the minimally required ones. Figure 11 shows its parameters.

PersonalMobilityService
id: str = "PersonalVehicle"
veh_capacity: integer = 1 dt_matching: integer = 0 dt_periodic_maintenance: integer = 0
estimate_pickup_time_for_planning

Figure 11: Parameters characterizing a `PersonalMobilityService` object at instantiating. List on the top corresponds to accessible parameters, list in the middle corresponds to inaccessible parameters, list on the bottom corresponds to interface methods.

5.2.1 Maintenance

The maintenance phase deletes the personal vehicles of users who arrived at their destination.

5.2.2 Periodic maintenance

There is no periodic maintenance for the personal mobility service.

5.2.3 Matching

A user requesting the personal mobility service is immediately matched with a null pick-up time with either:

- a new vehicle if user has not yet traveled with a personal vehicle during her journey
- the personal vehicle she used earlier in her journey otherwise

5.2.4 Interface methods

Pickup time estimation for planning The estimated pickup time for users planning is obviously null at any node.

5.3 Public Transport mobility service

The `PublicTransportMobilityService` corresponds to any type of public transport working with well defined lines and timetables. Figure 12 shows its parameters and interface methods. Note that lines with timetables belong to the layer on which the public transport service is defined.

PublicTransportMobilityService
id: str veh_capacity: integer = 50
dt_matching: integer = 0 dt_periodic_maintenance: integer = 0
estimate_pickup_time_for_planning

Figure 12: Parameters characterizing a `PublicTransportMobilityService` object at instantiating. List on the top corresponds to accessible parameters, list in the middle corresponds to inaccessible parameters, list on the bottom corresponds to interface methods.

5.3.1 Maintenance


The maintenance phase creates the vehicles at the first stops of the public transport lines ahead of their departure time and launches these vehicles service according to the timetables. It also deletes the vehicles once they arrived at the last stop of their line.

5.3.2 Periodic maintenance

There is no periodic maintenance for the public transport mobility service.

5.3.3 Matching

A user requesting the public transport mobility service is immediately matched with the vehicle that is expected to be the next to stop at user's desired pick-up node. The pick-up time is estimated considering the distance separating the selected vehicle and the requesting user divided by the vehicle's current speed. If this estimation is wrong, it has no real impact as the `pickup_dt` for all mobility services of type `PublicTransportationMobilityServices` is 24 hours for all users. When a public transport vehicle is matched with a user, the user pick-up and drop-off activities are inserted in the vehicle's plan so as to maintain the order in which lines stops should be visited.

 For now, the vehicle that is expected to be the next to stop at user's desired pick-up node means that we check vehicles of the proper line in the order in which vehicles have been created according to the timetables. If those vehicles are affected by the traffic, the FIFO rule is not enforced on the line, and a vehicle may arrive before at the stop than a vehicle which departed from the terminus earlier.

 For now, there is no check of the vehicle capacity constraint.

5.3.4 Interface methods

Pickup time estimation for planning The estimated pickup time for users planning is half the headway of the line serving the desired pickup node.

5.4 On demand mobility service

The `OnDemandMobilityService` corresponds to a free-floating ride-hailing service (without depot). Figure 13 shows its parameters and interface methods.

OnDemandMobilityService
<code>id: str</code> <code>dt_matching: integer</code> <code>dt_periodic_maintenance: integer = 0</code> <code>default_waiting_time: float = 0.</code> <code>matching_strategy: str = 'nearest_idle_vehicle_in_radius_fifo'</code> <code>radius: integer = 10000</code> <code>detour_ratio: float = 1.343</code>
<code>veh_capacity: integer = 1</code>
<code>estimate_pickup_time_for_planning</code> <code>create_waiting_vehicle</code> <code>add_zoning</code>

Figure 13: Parameters characterizing a `OnDemandMobilityService` object at instantiating. List on the top corresponds to accessible parameters, list in the middle corresponds to inaccessible parameters, list on the bottom corresponds to interface methods.

5.4.1 Maintenance

The maintenance phase proceeds to the computation of the estimated waiting time(s) while the `estimate_pickup_time_for_planning` method only reads the result of this computation. See section 5.4.4 for more details.

5.4.2 Periodic maintenance

There is no periodic maintenance for the on demand mobility service.

5.4.3 Matching

For now, four matching strategies are available in MnMS.

- `nearest_idle_vehicle_in_radius_fifo`: this strategy treats the customers in order of request (first to request, first to be treated). The nearest (in time) idle vehicle located within a certain radius around the desired pickup point is matched with the user. If there is no vehicle within the specified radius, user is not matched but her request is kept in the requests list and will be treated again during the next round of matching.
- `nearest_idle_vehicle_in_radius_batched`: this strategy treats jointly all the currently open (unmatched) requests. It solves the matching problem with the set of idle vehicles following a total pickup time minimization rule with a bipartite graph matching algorithm. An idle vehicle located outside the radius around a request's pickup node is not considered to serve this request.
- `nearest_vehicle_in_radius_fifo`: this strategy treats the customers in order of request (first to request, first to be treated). The nearest (in time) vehicle located within a certain radius around the desired pickup point at the end of its plan is matched with the user. If no

vehicle is or finishes its plan within the specified radius, user is not matched but her request is kept in the requests list and will be treated again during the next round of matching.

- **nearest_vehicle_in_radius_batched**: this strategy treats jointly all the currently open (un-matched) requests. It solves the matching problem with the set of vehicles following a total pickup time minimization rule with a bipartite graph matching algorithm. A vehicle finishing its plan outside the radius around a request's pickup node is not considered to serve this request.

Note that the estimated waiting times are recomputed at the end of each matching phase.

5.4.4 Interface methods

Pickup time estimation for planning Note that the pickup time estimation contains the time taken to obtain a match and the time taken for the matched vehicle to join the pickup point. To estimate the pickup time taken into account at the moment of users' planning, the market is considered in a stationary state, customers' pickup locations are supposed to follow a uniform distribution, and idle vehicle's locations are supposed to follow a spatial Poisson process. The pickup time estimation is realized for the whole layer the on-demand mobility service is running on if no zoning is specified for the service, and for each zone otherwise. Note that if the zoning defined does not cover the entire network, the pickup time estimation is realized for all links which do not belong to a zone considering the whole layer. Also, if a node belongs to several zones, the estimated pickup time at this node is the mean of the estimated pickup times of these zones. Figure 14 illustrates this case. Based on the work of [4]:

- In **oversupply** mode, i.e. when the number of idle vehicles is greater than the number of open requests, the estimated pickup time w is given by Equation 2.

$$w = \frac{\tau}{2} + \frac{\zeta}{2v\sqrt{\rho_s}} \quad (2)$$

where τ is the matching time step, ζ is the detour ratio (the distance on the actual road network to straight line distance, chosen equal to 1.343), v is the mean speed of vehicles, ρ_s is the idle vehicle density.

- In **undersupply** mode, i.e. when the number of idle vehicles is lower than the number of open requests, the estimated pickup time w is given by Equation 3.

$$w = \frac{\rho_d A}{Q} - \frac{\tau}{2} + \frac{\zeta}{v\sqrt{\pi\rho_d}} \quad (3)$$

where ρ_d is the open requests density, A is the service area, Q is the mean requests arrival rate.

- When there are no idle vehicle nor open requests and before the first maintenance phase, the `default_waiting_time` parameter is used.

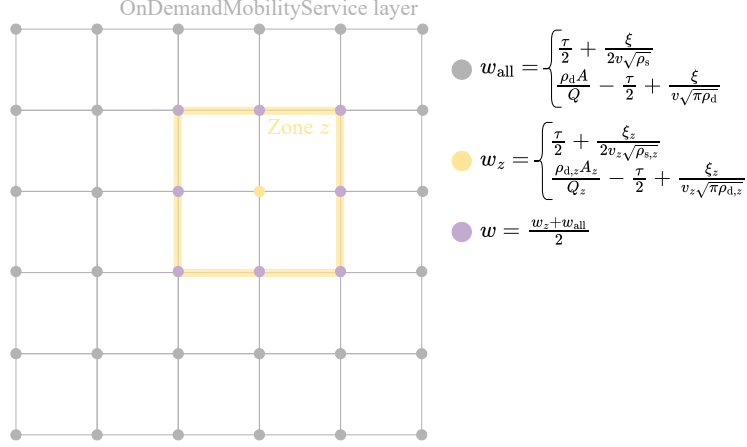


Figure 14: Case when the on demand service zoning does not cover the whole network.

Other interface methods There exist two specific interface methods:

- **create_waiting_vehicle**: to create/initiate vehicles in this service's fleet. It takes the node where the vehicle should be created as input.
- **add_zoning**: to define a zoning for the service. For now, the zoning is only used for waiting times estimation, but it can be used to define specific matching or rebalancing strategies. It takes a list of **LayerZone** objects as input. All links of one **LayerZone** object should belong to the service's layer graph.

5.5 On demand mobility service with depots

The **OnDemandDepotMobilityService** corresponds to a ride-hailing service with depots. Figure 15 shows its parameters and interface methods.

OnDemandDepotMobilityService
id: str dt_matching: integer dt_periodic_maintenance: integer = 0 default_waiting_time: float = 0. matching_strategy: str = 'nearest_idle_vehicle_in_radius_fifo' radius: integer = 10000 detour_ratio: float = 1.343
veh_capacity: integer = 1
estimate_pickup_time_for_planning create_waiting_vehicle add_zoning add_depot

Figure 15: Parameters characterizing a **OnDemandDepotMobilityService** object at instantiating. List on the top corresponds to accessible parameters, list in the middle corresponds to inaccessible parameters, list on the bottom corresponds to interface methods.

5.5.1 Maintenance

The maintenance phase makes each stopped vehicle reposition itself toward the closest non full depot. It also proceeds to the computation of the estimated waiting time(s) while the `estimate_pickup_time_for_planning` method only reads the result of this computation.

5.5.2 Periodic maintenance

There is no periodic maintenance for the on demand depot mobility service.

5.5.3 Matching

The matching strategies of this mobility service are the same as the ones of `OnDemandMobilityService`.

5.5.4 Interface methods

Pickup time estimation for planning The pickup time estimation of this mobility service is the same as the one of `OnDemandMobilityService`.

Other interface methods There exist three specific interface methods:

- `create_waiting_vehicle`: to create/initiate vehicles in this service's fleet. It takes the node where the vehicle should be created as input.
- `add_depot`: to create/initiate a depot eventually full of vehicles. It takes the node where to create the depot, its capacity, and a boolean specifying if it should be filled with vehicles as input.
- `add_zoning`: to define a zoning for the service. For now, the zoning is only used for waiting times estimation, but it can be used to define specific matching or rebalancing strategies. It can be called without argument, then the zoning is built based on the Voronoi diagram of depots. It can also be called with a list of `LayerZone` objects as input.

5.6 Vehicle sharing mobility service

The `VehicleSharingMobilityService` corresponds to either a station-based or a free-floating vehicle sharing service. Figure 16 shows its parameters.

VehicleSharingMobilityService
id: str free_floating_possible: bool dt_matching: integer dt_periodic_maintenance: integer = 0 critical_nb_vehs: integer = 10 alpha: float = 600 beta: float = 0.1
veh_capacity: integer = 1
estimate_pickup_time_for_planning create_station init_free_floating_vehicles

Figure 16: Parameters characterizing a `VehicleSharingMobilityService` object at instantiating. List on the top corresponds to accessible parameters, list in the middle corresponds to inaccessible parameters, list on the bottom corresponds to interface methods.

5.6.1 Maintenance

The maintenance phase makes sure that each vehicle stopped at the location of an existing station is associated with this station. If the service authorizes free-floating, a vehicle stopped at a node where no station exist leads to the creation of a free-floating station.

5.6.2 Periodic maintenance

There is no periodic maintenance for the vehicle sharing mobility service.

5.6.3 Matching

Customers are treated in order of request. User is matched with one available vehicle parked at the station where user is. If no vehicle is available at this station, user is not matched but her request is kept in the service requests list to be eventually matched during the following matching rounds.

5.6.4 Interface methods

Pickup time estimation for planning The estimation of the pickup time for users planning depends on the type of vehicle sharing service:

- For station-based vehicle sharing services, the waiting time is estimated with a BPR-like function.

$$w = \alpha \left(1 - \left(\frac{n_s}{n_C} \right)^\beta \right) \quad (4)$$

where n_s is the number of vehicles currently available at station s , n_C is the critical number of vehicles, i.e., the number of vehicles available at station below which the estimated waiting time starts to increase (default is 10), α is the maximum estimated pickup time (default is 600), β is a parameter controlling the shape of the estimated pickup time curve (default is 0.1). See Figure 17 for typical shapes.

- For free-floating vehicle sharing services, the waiting time is considered null, we do not take into account any risk for the free-floating vehicle to be taken by another traveler. However, note that a free-floating station is disconnected from the rest of the graph and removed when its vehicle is taken by a user. This leads to an **INTERRUPTION** event for all users who planned to pass through the deleted free-floating station. They replan during the next planing phase while considering only available free-floating vehicles/stations.

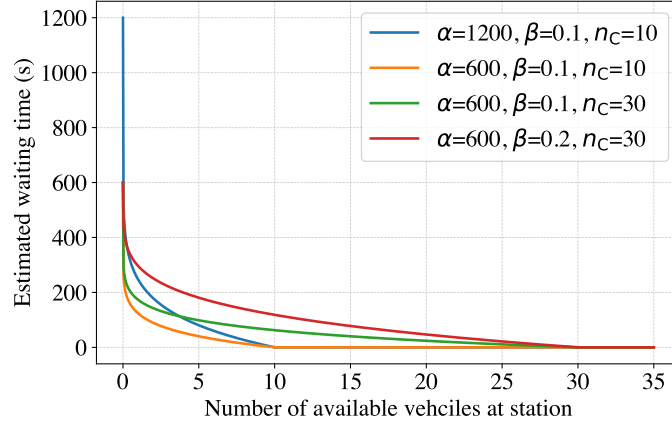


Figure 17: Typical estimated pickup time curves for the station-based vehicle sharing service.

Other interface methods This mobility service has one specific interface method:

- **create_station**: to create a station with a certain number of vehicles in it. This method takes the station name, its location (either a **RoadDescriptor** or a **VehicleSharingLayer** graph node), its capacity (default is 30), the number of vehicles to create in it (default is 0), and a boolean specifying if the station is free-floating or not.
- **init_free_floating_vehicles**: to create a certain number of free-floating vehicles at a certain node and the associated station.

5.7 On demand shared mobility service

The **OnDemandSharedMobilityService** corresponds to a ridesharing service. Figure 18 shows its parameters and interface methods.

OnDemandSharedMobilityService
id: str veh_capacity: integer dt_matching: integer dt_periodic_maintenance: integer default_waiting_time: float = 0 matching_strategy: str = "smallest_disutility_vehicle_in_radius_fifo" replanning_strategy: str = "all_pickups_first_fifo" radius: float = 10000 detour_ratio: float = 1.343
estimate_pickup_time_for_planning create_waiting_vehicle add_zoning

Figure 18: Parameters characterizing a `OnDemandSharedMobilityService` object at instantiating. List on the top corresponds to accessible parameters, list in the middle corresponds to inaccessible parameters, list on the bottom corresponds to interface methods.

5.7.1 Maintenance

To support the matching, and especially the verification of users' maximum detour ratio constraints, this service keeps in memory a piece of information about the users who got a match. This information includes the distance user has traveled at the moment when she got matched, the initial distance she planned to ride this service (D_i^{init} where i designates the user index), and the distance she traveled onboard the vehicle she was matched with so far (D_i where i designates the user index). The maintenance phase deals with the update of this information: it updates the distances users matched with this service have run onboard the vehicle they were matched with and deletes the information of users who have already been dropped off by a vehicle of this service.

Additionally, it computes the estimated waiting time(s) for a request in each zone of this service while the `estimate_pickup_time_for_planning` method only reads the result of this computation. See section 5.7.4 for more details.

5.7.2 Periodic maintenance

There is no periodic maintenance for the on demand shared mobility service.

5.7.3 Matching

Matching strategies For now, there is only one matching strategy for this service.

- `smallest_disutility_vehicle_in_radius_fifo`: this strategy treats customers in order of request (first to request, first to be treated). Among vehicles currently located within a radius around the requesting user and able to be matched with a new request, the one leading to the smallest matching disutility under maximum detour ratios constraints is chosen to be matched with the user. To define the disutility of a match between a request r and a vehicle v , noted $-u_{r,v}$, let us note R_v the requests that have been already matched with v and not yet served/dropped off; P_{R_v} the plan of v , i.e., the order of pickup and drop-off activities the vehicle has to achieve; $d(i, P_{R_v})$ the remaining distance user i has to travel onboard the

vehicle in plan P_{R_v} (null if user i does not appear in this plan), ζ_i the maximum detour ratio for accepted by user i . In this strategy, we have:

$$-u_{r,v} = \sum_{i \in R_v \cup \{r\}} -u(i, P_{R_v}, P_{R_v \cup \{r\}}) \quad (5)$$

$$-u(i, P_{R_v}, P_{R_v \cup \{r\}}) = \begin{cases} d(i, P_{R_v \cup \{r\}}) - d(i, P_{R_v}), & \text{if } \frac{D_i + d(i, P_{R_v \cup \{r\}})}{D_i^{\text{init}}} \leq \zeta_i \\ +\infty, & \text{otherwise} \end{cases} \quad (6)$$

The replanning strategy dictates the way $P_{R_v \cup \{r\}}$ is computed and the condition to consider whether a vehicle in radius is able to be matched with a new request or not.

NB: The disutility can be modified directly in the `OnDemandSharedMobilityService` class code, if one wants to consider other aspects than distance.

Replanning strategies A replanning strategy designates the way new activities are inserted in a vehicle's plan and its old activities are re-ordered consequently. For now, there is only one replanning strategy.

- `all_pickups_first_fifo`: it inserts all new pickup activities in the order in which they are passed right after the last pickup activity of vehicle's current plan, and all serving activities in the order in which they are passed in the end of vehicle's current plan. Consequently, a vehicle is able to be matched with a new request when the sum of the number of its passengers and the number of its expected passengers (regarding its current plan) is strictly below its capacity.

5.7.4 Interface methods

Pickup time estimation for planning The pickup time estimation of this mobility service is the same as the one of `OnDemandMobilityService`.

Other interface methods The other interface methods are the same as the ones of `OnDemandMobilityService`.

5.8 Outputs

A `VehicleObserver` object can be attached to a mobility service with the `attach_vehicle_observer` method to follow the activity of its vehicles. It outputs a .csv file with the following columns: **TIME** (current time), **TYPE** (type of the vehicle), **LINK** (link on which the vehicle is), **POSITION** (coordinates of the vehicle), **SPEED** (speed of the vehicle), **STATE** (activity type currently occupying the vehicle), **DISTANCE** (distance traveled by the vehicle so far), **PASSENGERS** (passengers currently within the vehicle).

6 Simulation

6.1 Simulation launching

Launching a simulation requires a well-built `Supervisor` object, some mandatory parameters including the simulation start time (t_{start} or `tstart`), end time (t_{end} or `tend`), the time step for vehicles and users motion (dt_{flow} or `flow_dt`), the affectation factor (dt_{aff} or `affectation_factor`), and eventually some optional parameters including the threshold on speed variation for updating the costs on the graph links (`update_graph_threshold`), and the simulation seed (`seed`).


NB: It is recommended to start the simulation one affectation time step earlier the time at which the first user departs from home to let the costs on the graph links be updated according to the MFD functions.

6.2 Simulation flow chart

Figure 19 represents the simulation flow chart.

6.3 Computational saving modes

There are several ways to decrease the simulation time in MnMS.

- `update_graph_threshold`: this argument can be passed to the `run` method of the `Supervisor` class. It specifies a threshold on the speed variation below which costs on the `MultiLayerGraph` object are not updated. In this way, less time is spent on updating the graph for negligible variation of the costs.
- `save_routes_dynamically_and_reapply`: this argument can be passed to the decision model at instantiating. When it is set to `True`, the shortest paths computed along the simulation are saved. If k shortest paths should be computed for a certain origin, destination, available layers-mobility services, and intermodality, we verify if these paths has already been computed and saved. If so, they are reused. Their travel cost is recomputed based on current costs, and the decision model is applied to let the user select one of them.
- `load_shortest_paths`: there exist two functions in `src/mnms/tools/preprocessing.py` to pre compute the shortest paths between all pairs of nodes of a layer's graph, `compute_all_shortest_paths_floyd_warshall` and `compute_all_shortest_paths`. You can use one of them depending on the size of the graph (for the biggest graph, `compute_all_shortest_paths` should be the most efficient function). Once pre-computed and dumped into a json file, they can be loaded into the proper layer with the method `load_shortest_paths`. For now, such loaded shortest paths are used only in the `request_nearest_idle_vehicle_in_radius_fifo` method of the `OnDemandMobilityService` class to compute the pick up paths quickly. But this feature can also be used in future in other matching strategies, other mobility services, or even the `AbstractDecisionModel` class.
-  `dynamic_flow_dt`: the idea is to adapt the flow time step depending on the vehicle and its situation so as to save computation time in moving the vehicles. For all personal vehicles, the flow time step can be increased when the vehicle is moving within a reservoir and is not about to finish its activity. The adaptive time step should be deployed for the vehicles belonging to

mobility services that do not require precise vehicle location, depends on the percentage of distance traveled compared to the total distance to travel, and on the distance from current reservoir's border.

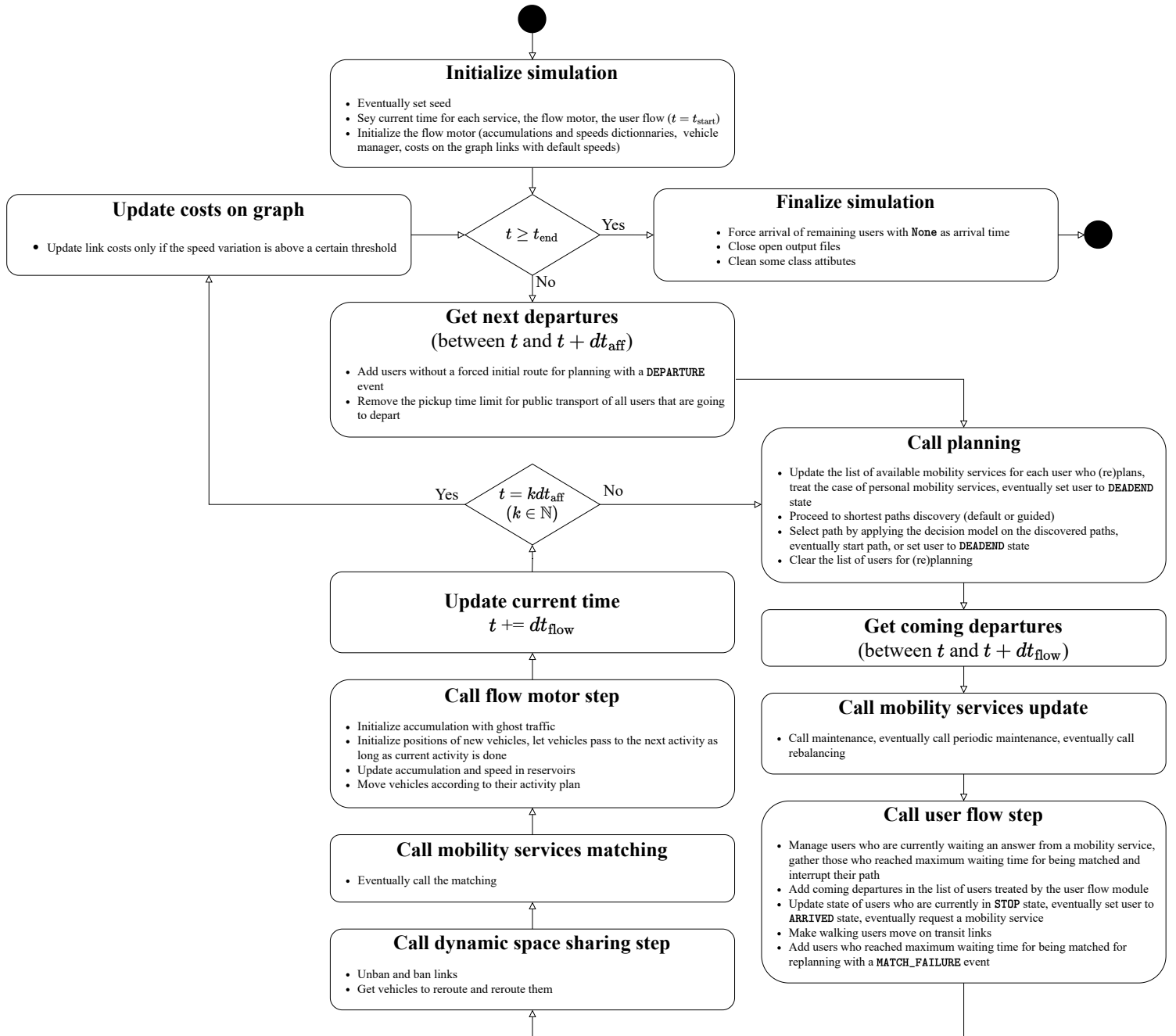


Figure 19: Simulation detailed flow chart.

7 Study cases

7.1 Lyon 6

This study case is the simplest one based on a real network. It extends on the 6th district of Lyon city, France. It has only one transportation layer for one personal vehicle mobility service. The origin-destination layer matches this layer's nodes. There is one reservoir including all links with a varying speed with accumulation.

There are two versions of this scenario, one with an assumed links classification and another one without any classification. The classification recognizes primary and secondary roads as represented in Figure 20.

To influence users paths based on this classification, the study case uses the cost functions to apply a certain multiplier to the travel time depending on the link class. The new cost considered to compute shortest paths is called `weighted_travel_time`. Figure 21 compares the number of visits on links with and without the application of these multiplier.



Figure 20: Classification of links for the Lyon 6 study case.

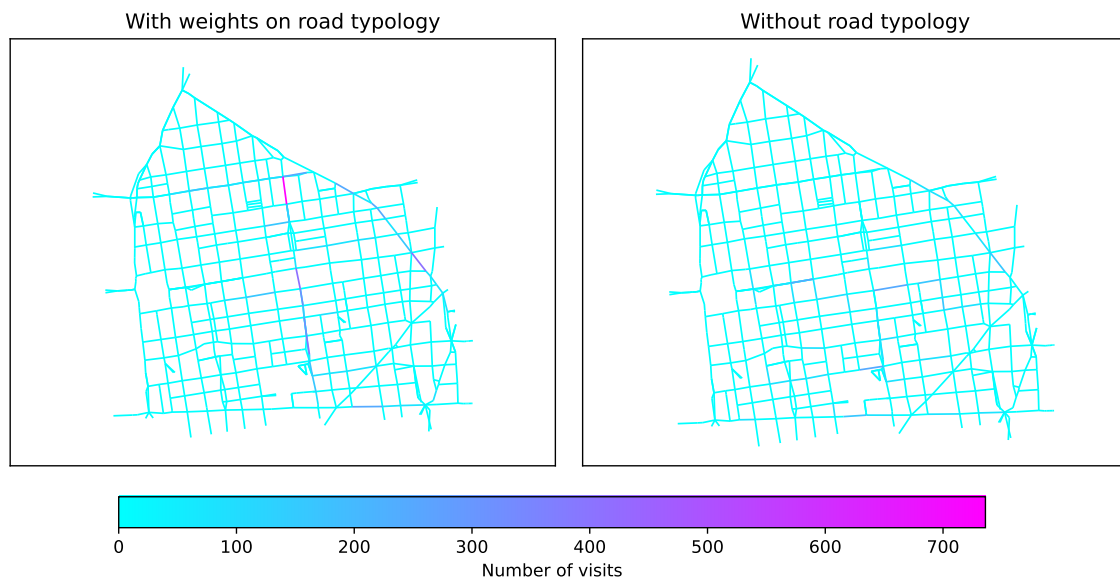


Figure 21: Comparison with and without application of a multiplier depending on link class (the multiplier is 1 for primary links, and 2 for secondary links in this example).

References

- [1] Carlos F. Daganzo. Urban gridlock: Macroscopic modeling and mitigation approaches. *Transportation Research Part B: Methodological*, 41(1):49–62, 1 2007.
- [2] Nikolas Geroliminis and Carlos F. Daganzo. Existence of urban-scale macroscopic fundamental diagrams: Some experimental findings. *Transportation Research Part B: Methodological*, 42(9):759–770, 11 2008.
- [3] Raphaël Lamotte and Nikolas Geroliminis. The morning commute in urban areas with heterogeneous trip lengths. *Transportation Research Part B: Methodological*, 117:794–810, 11 2018.
- [4] Xinwei Li, Jintao Ke, Hai Yang, Hai Wang, and Yaqian Zhou. A general matching function for ride-sourcing services. *SSRN Electronic Journal*, 2021.
- [5] Guilhem Mariotte, Ludovic Leclercq, and Jorge A. Laval. Macroscopic urban dynamics: Analytical and numerical comparisons of existing models. *Transportation Research Part B: Methodological*, 101:245–267, 7 2017.
- [6] Wei Ni and Michael Cassidy. City-wide traffic control: Modeling impacts of cordon queues. *Transportation Research Part C: Emerging Technologies*, 113:164–175, 4 2020.